БИБЛИОТЕКА ПРОГРАММИСТА

М. Ву, Т. Девис, Дж. Нейдер, Д. Шрайнер

OpenGL

4-е издание Руководство по программированию

- Базовые и дополнительные методы компьютерной графики
- Повышение производительности программ
- Расширения, создание кросс-платформных приложений
- Последние обновления языка OpenGL

Addison-Wesley

ПИТЕР®

БИБЛИОТЕКА ПРОГРАММИСТА

М. Ву, Т. Девис, Дж. Нейдер, Д. Шрайнер

OpenGL

4-е издание Руководство по программированию

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2006

ББК 32.973-018.3
УДК 004.92
Ш85

Ву М., Девис Т., Нейдер Дж., Шрайнер Д.
Ш85 OpenGL. Руководство по программированию. Библиотека программиста. 4-е издание. — СПб.: Питер, 2006. — 624 с.: ил.

ISBN 5-94723-827-6

Это 4-е издание признанного бестселлера, посвященного OpenGL и его библиотеке инструментов. В книге описаны все возможности OpenGL и самые значительные приложения, содержится описание базовых методов компьютерной графики, таких как построение и воспроизведение трехмерных моделей, интерактивный просмотр объектов с различных точек наблюдения, использование тонирования, освещения и эффектов текстурирования. Представлено углубленное описание дополнительных методов компьютерной графики: наложение текстур, сглаживание, «туман» и имитация других атмосферных эффектов, сплайны, конвейерная обработка изображений и другие ключевые темы, такие как повышение производительности программ, расширения OpenGL и создание кросс-платформных приложений.

ББК 32.973-018.3
УДК 004.92

Права на издание получены по соглашению с Addison-Wesley Longman.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

© 2004 by Silicon Graphics, Inc.
© Перевод на русский язык, ЗАО Издательский дом «Питер», 2006
© Издание на русском языке, оформление, ЗАО Издательский дом «Питер», 2006

ISBN 0321173481 (англ.)
ISBN 5-94723-827-6

Краткое содержание

Об этой книге 15
Благодарности 21
Глава 1. Введение в OpenGL 22
Глава 2. Управление состоянием и рисование геометрических объектов 43
Глава 3. Визуализация 96
Глава 4. Цвет 145
Глава 5. Освещение 160
Глава 6. Смешивание, сглаживание, туман и отклонение 198
Глава 7. Списки отображения 238
Глава 8. Вывод пикселов, битовых образов, шрифтов и изображений 259
Глава 9. Наложение текстур 309
Глава 10. Буфер кадров 386
Глава 11. Мозаичное представление и поверхности второго порядка 416
Глава 12. Вычислители и NURBS 438
Глава 13. Выбор и обратная связь 466
Глава 14. Теперь, когда вы знаете многое 492
Приложения 519
Приложение А. Порядок выполнения операций 520
Приложение Б. Параметры состояния 525
Приложение В. OpenGL и оконные системы 553
Приложение Г. Основы GLUT: инструментальный набор пакета разработчика OpenGL 572
Приложение Д. Вычисление векторов нормалей 577
Приложение Е. Однородные координаты и матричные преобразования 582
Приложение Ж. Приемы программирования 588
Приложение З. OpenGL и инвариантность 593
Словарь терминов 595
Алфавитный указатель 610

Содержание

Об этой книге 15
Что нового в этом издании 17
Что необходимо знать перед прочтением 18
Как получить примеры программ 19
Обучающие программы по OpenGL от Нэйта Робинса 20
Используемые стили 20
От издателя перевода 20

Благодарности 21

Глава 1. Введение в OpenGL 22
Что такое OpenGL 23
Немного программного кода 26
Синтаксис команд OpenGL 27
OpenGL как конечный автомат 29
Конвейер рисования OpenGL 30
Списки отображения 31
Вычислители 31
Операции над вершинами 31
Компоновка примитивов 31
Операции над пикселами 32
Компоновка текстур 32
Растеризация 32
Операции над фрагментами 33
Библиотеки, связанные с OpenGL 33
Включаемые файлы 34
GLUT, набор инструментов OpenGL 35
Анимация 38
Обновление с задержкой 40
Движение = Перерисовка + Обмен 40

Глава 2. Управление состоянием
и рисование геометрических объектов 43
Основы рисования 45
Очистка окна 45
Установка цвета 48
Ускорение завершения рисования 49
Управление системой координат 50
Описание точек, линий и многоугольников 51
Что такое точки, линии и многоугольники? 51
Задание вершин 55

Рисование графических примитивов в OpenGL . . . 56
Управление основными состояниями . . . 60
Отображение точек, линий и многоугольников . . . 61
Подробно о точках . . . 61
Подробно о линиях . . . 62
Подробно о многоугольниках . . . 66
Векторы нормали . . . 72
Массивы вершин . . . 73
Шаг 1: активация массивов . . . 75
Шаг 2: помещение данных в массивы . . . 76
Шаг 3: разыменование и рисование . . . 78
Смешанные массивы . . . 84
Группы атрибутов . . . 86
Несколько советов по построению полигональных моделей поверхностей . . . 89
Пример: построение икосаэдра . . . 90

Глава 3. Визуализация . . . 96

Обзор: аналогия с камерой . . . 98
Пример: рисование куба . . . 101
Команды преобразования общего назначения . . . 105
Модельно-видовые преобразования . . . 107
Размышления о преобразованиях . . . 107
Модельные преобразования . . . 110
Видовые преобразования . . . 114
Преобразования проецирования . . . 119
Перспективное проецирование . . . 120
Ортогональное проецирование . . . 122
Отсечение объемом видимости . . . 124
Преобразование окна просмотра . . . 124
Определение окна просмотра . . . 124
Измененная координата глубины . . . 126
Проблемы преобразований . . . 127
Управление стеком матриц . . . 129
Стек модельно-видовых матриц . . . 132
Стек матриц проекций . . . 132
Дополнительные плоскости отсечения . . . 133
Примеры объединения нескольких преобразований . . . 135
Солнечная система . . . 135
Сочлененная рука робота . . . 138
Обратные, или имитационные, преобразования . . . 141

Глава 4. Цвет . . . 145

Цветовосприятие . . . 146
Вычисление цвета . . . 147
RGBA в сравнении с индексированным цветом . . . 150
Режим RGBA . . . 150
Режим индексации цвета . . . 152
Выбор между режимами RGBA и индексации цвета . . . 153
Переключение между режимами . . . 154
Задание цветовой и полутоновой моделей . . . 155
Задание цвета в режиме RGBA . . . 155

Задание цвета в режиме индексации цвета . . . 156
Задание модели закрашивания . . . 157

Глава 5. Освещение . . . 160

Удаление невидимых поверхностей . . . 162
Реальный мир и освещение в OpenGL . . . 163
Фоновый, рассеянный, отраженный и излучаемый свет . . . 164
Цвета материала . . . 165
Значения RGB для источников света и материалов . . . 166
Пример: рисование освещенной сферы . . . 166
Задание векторов нормалей для каждой вершины каждого объекта . . . 168
Создание, размещение и включение одного или нескольких источников света . . . 169
Выбор модели освещения . . . 169
Задание свойств материала для объектов в сцене . . . 169
Замечания на память . . . 170
Создание источников света . . . 170
Цвет . . . 171
Положение и ослабление . . . 172
Прожекторы . . . 173
Несколько источников света . . . 175
Управление положением и направлением источников света . . . 176
Выбор модели освещения . . . 181
Общее фоновое освещение . . . 181
Локальная или бесконечно удаленная точка обзора . . . 182
Двухстороннее освещение . . . 182
Вторичный отраженный цвет . . . 183
Включение источников света . . . 183
Задание свойств материала . . . 184
Рассеянное и фоновое отражение . . . 185
Зеркальное отражение . . . 186
Излучение . . . 186
Изменение свойств материала . . . 186
Режим цвета материала . . . 188
Математика освещения . . . 191
Излучение материала . . . 192
Масштабированный общий фоновый свет . . . 192
Вклады от источников света . . . 192
Вторичный отраженный цвет . . . 194
Освещение в режиме индексации цвета . . . 195
Математика освещения в режиме индексации цвета . . . 196

Глава 6. Смешивание, сглаживание, туман и отклонение . . . 198

Смешивание . . . 200
Факторы влияния источника и получателя . . . 201
Включение смешивания . . . 203
Объединение пикселов на основе выражений . . . 203
Область применения смешивания . . . 205
Пример смешивания . . . 207
Трехмерное смешивание с использованием буфера глубины . . . 209
Сглаживание . . . 212

Сглаживание точек или линий . . . 214
Сглаживание геометрических примитивов с помощью мультивыборки . . . 219
Альфа-компонент и покрытие мультивыборки . . . 222
Сглаживание границ многоугольников . . . 223
Туман . . . 224
Применение тумана . . . 225
Уравнения тумана . . . 227
Координаты тумана . . . 230
Параметры точки . . . 233
Отклонение многоугольников по глубине . . . 235

Глава 7. Списки отображения . . . 238

Для чего нужны списки отображения . . . 239
Пример списка отображения . . . 240
Философия проектирования списков отображения . . . 243
Создание и выполнение списка отображения . . . 245
Присвоение имени списку отображения и создание списка . . . 246
Что сохраняется в списке отображения? . . . 247
Выполнение списка отображения . . . 249
Иерархические списки отбражения . . . 249
Управление индексацией списков отображений . . . 251
Выполнение составных списков отображения . . . 251
Управление параметрами состояния с помощью списков отображения . . . 256
Инкапсуляция изменений режима . . . 257

Глава 8. Вывод пикселов, битовых образов, шрифтов и изображений . . . 259

Битовые образы и шрифты . . . 261
Текущая позиция растра . . . 262
Вывод битового образа . . . 264
Выбор цвета для битового образа . . . 265
Шрифты и списки отображения . . . 265
Описание и использование полного шрифта . . . 267
Изображения . . . 269
Чтение, запись и копирование пикселов . . . 269
Конвейерная обработка изображений . . . 275
Упаковка и распаковка пикселов . . . 278
Режимы хранения пикселов . . . 279
Операции при передаче пикселов . . . 282
Табличные преобразования пикселов . . . 284
Растяжение, сжатие и зеркальное отражение изображений . . . 285
Чтение и рисование прямоугольных областей . . . 288
Процесс вывода пиксельного прямоугольника . . . 288
Советы по увеличению быстродействия вывода пикселов . . . 292
Расширение Imaging Subset . . . 292
Таблицы цветов . . . 294
Свертки . . . 297
Матрица цвета . . . 303
Гистограмма . . . 305
Метод минимакса . . . 307

Глава 9. Наложение текстур 309
Общий обзор и пример 314
Действия при наложении текстур 314
Пример программы 316
Описание текстуры 318
Текстуры-посредники 322
Замещение изображения текстуры или его части 323
Одномерные текстуры 326
Трехмерные текстуры 328
Сжатые изображения текстур 332
Использование границ текстуры 334
MIP-отображение: детализация на множестве уровней 334
Фильтрация 343
Объекты текстур 346
Присвоение имен объектам текстур 346
Создание и использование объектов текстур 347
Удаление объектов текстур 349
Рабочий набор резидентных текстур 350
Функции текстур 352
Присвоение координат текстур 355
Вычисление подходящих координат текстуры 356
Повторяемые и обрезанные текстуры 358
Автоматическая генерация координат текстур 361
Создание контуров 362
«Сферические» карты 366
«Кубические» текстуры 368
Мультитекстурирование 370
Комбинирование функций текстур 374
Интерполирующие функции комбинирования 378
Применение вторичного цвета после текстурирования 379
Вторичный цвет при отключенном освещении 380
Вторичный отраженный цвет при отключенном освещении 380
Стек матриц текстур 380
Глубина текстур 382
Создание «теневой» карты 382
Генерация координат текстур и их отображение 384

Глава 10. Буфер кадров 386
Буферы и их применение 388
Буферы цвета 389
Очистка буферов 390
Выбор буферов цвета для записи и чтения 391
Маскирование буферов 393
Тесты и операции над фрагментами 394
Тест на отсечение 394
Альфа-тест 395
Тест трафарета 396
Тест глубины 401
Смешивание, размытие и логические операции 402
Буфер накопления 404
Сглаживание сцены 405

Размытие при движении . . . 410
Глубина резкости . . . 410
Плавные тени . . . 414
«Дрожание» . . . 414

Глава 11. Мозаичное представление и поверхности второго порядка . . . 416
Мозаичное представление многоугольников . . . 417
Создание объекта разбиения . . . 418
Процедуры обратной связи . . . 419
Свойства мозаичного разбиения . . . 423
Описание многоугольника . . . 427
Удаление объекта разбиения . . . 429
Советы по увеличению производительности разбиения . . . 429
Сообщения об ошибках GLU . . . 430
Обратная совместимость . . . 430
Поверхности второго порядка: сферы, цилиндры и диски . . . 431
Управление квадратичными объектами . . . 432
Управление атрибутами квадратичных объектов . . . 433
Квадратичные примитивы . . . 434

Глава 12. Вычислители и NURBS . . . 438
Необходимые знания . . . 440
Вычислители . . . 440
Одномерные вычислители . . . 441
Двухмерные вычислители . . . 445
Применение вычислителей для текстур . . . 450
Интерфейс NURBS библиотеки GLU . . . 452
Пример простого NURBS-объекта . . . 452
Управление NURBS-объектом . . . 456
Создание NURBS-кривой или поверхности . . . 459
Усечение NURBS-поверхностей . . . 463

Глава 13. Выбор и обратная связь . . . 466
Выбор . . . 467
Основные шаги . . . 468
Создание стека имен . . . 469
Буфер выбора . . . 470
Пример выбора . . . 471
Выбор с помощью указателя мыши . . . 474
Рекомендации по написанию программы, использующей механизм выбора . . . 483
Обратная связь . . . 485
Массив обратной связи . . . 486
Использование маркеров в режиме обратной связи . . . 487
Пример обратной связи . . . 488

Глава 14. Теперь, когда вы знаете многое . . . 492
Обработка ошибок . . . 494
Какую версию я использую? . . . 496

Версия библиотеки инструментов . . . 497
Версии расширений для оконных систем . . . 497
Расширения стандарта . . . 497
Расширения стандарта для Microsoft Windows (WGL) . . . 499
Полупрозрачность . . . 499
Простой эффект «растворения» изображения . . . 500
Использование фонового буфера для выбора объектов . . . 501
Простая трансформация изображения . . . 502
Послойное отображение . . . 504
Сглаживание символов . . . 504
Рисование округлых точек . . . 506
Интерполяция изображений . . . 506
Создание устойчивых изображений . . . 507
Рисование закрашенных вогнутых многоугольников с помощью буфера трафарета . . . 508
Нахождение перекрывающихся областей . . . 510
Тени . . . 511
Удаление невидимых линий . . . 512
Удаление невидимых линий со смещением многоугольника . . . 513
Удаление невидимых линий с помощью буфера трафарета . . . 513
Текстурирование . . . 514
Рисование изображений с буферизацией глубины . . . 515
Области Дирихле . . . 515
«Жизнь» в буфере трафарета . . . 516
Альтернативное применение glDrawPixels() и glCopyPixels() . . . 518

Приложения . . . 519

Приложение А. Порядок выполнения операций . . . 520
Общий обзор . . . 521
Векторные операции . . . 522
Операции над вершинами . . . 522
Сборка примитивов . . . 522
Операции над пикселами . . . 523
Память текстур . . . 523
Операции над фрагментами . . . 524
Прочее . . . 524

Приложение Б. Параметры состояния . . . 525
Команды запроса . . . 526
Параметры состояния OpenGL . . . 527
Текущие значения и связанные данные . . . 528
Массив вершин . . . 529
Преобразования . . . 531
Окрашивание . . . 532
Освещение . . . 533
Растеризация . . . 534
Мультивыборка . . . 536
Текстурирование . . . 536
Операции над пикселами . . . 541
Управление буфером кадров . . . 542

Пикселы . . . 543
Вычислители . . . 547
Рекомендации . . . 548
Параметры, зависящие от реализации . . . 548
Параметры глубины, зависящие от реализации . . . 551
Прочее . . . 551

Приложение В. OpenGL и оконные системы . . . 553
Обращение к новым функциям OpenGL . . . 554
GLX: расширение OpenGL для X Window System . . . 554
Инициализация . . . 555
Управление визуализацией . . . 556
Прототипы GLX . . . 558
AGL: расширение OpenGL для Apple Macintosh . . . 560
Инициализация . . . 561
Визуализация и контексты . . . 561
Управление контекстом визуализации OpenGL . . . 561
Экранная визуализация . . . 562
Внеэкранная визуализация . . . 562
Полноэкранная визуализация . . . 562
Перестановка буферов . . . 562
Обновление буферов визуализации . . . 562
Использование шрифтов Apple Macintosh . . . 562
Обработка ошибок . . . 563
Прототипы AGL . . . 563
PGL: расширение OpenGL для IBM OS/2 Warp . . . 564
Инициализация . . . 565
Управление визуализацией . . . 565
Прототипы PGL . . . 566
WGL: расширение OpenGL для Microsoft Windows 95/98/NT/ME/2000/XP . . . 567
Инициализация . . . 568
Управление визуализацией . . . 568
Прототипы WGL . . . 570

Приложение Г. Основы GLUT: инструментальный набор пакета разработчика OpenGL . . . 572
Инициализация и создание окон . . . 573
Управление окнами и событиями . . . 574
Загрузка карты цвета . . . 575
Инициализация и отображение трехмерных объектов . . . 575
Управление фоновыми процессами . . . 576
Запуск программ . . . 576

Приложение Д. Вычисление векторов нормалей . . . 577
Нахождение нормалей к аналитическим поверхностям . . . 578
Нахождение нормалей к полигональным поверхностям . . . 580

Приложение Е. Однородные координаты и матричные преобразования . . . 582
Однородные координаты . . . 583

Преобразование вершин 583
Преобразование нормалей 583
Матричные преобразования 584
Параллельный перенос 584
Масштабирование 585
Поворот 585
Перспективная проекция 586
Ортогональная проекция 586

Приложение Ж. Приемы программирования 588
Правильное обращение с OpenGL 589
Приемы увеличения производительности 590
Советы по применению GLX 592

Приложение З. OpenGL и инвариантность 593

Словарь терминов 595

Алфавитный указатель 610

Моей семье — Фелисити, Максу, Саре и Скауту.
Джеки Нейдер (JLN)

Моей семье — Эллин, Рикки и Люси.
Том Дэвис (TRD)

Тому Деппнеру и Энди ван Даму, которые направили меня по этому пути.
Мэйсон Ву (MW)

Моей семье — Викки, Бонни, Бобу, Сквигалзу, Фантому, Тоби и Токседо.
Дэйв Шрейнер (DRS)

Памяти Фила Карлтона

Об этой книге

Графическая система OpenGL — это программный интерфейс к графическому оборудованию (GL означает Graphics Library — графическая библиотека). Она позволяет создавать интерактивные программы, которые формируют движущиеся цветные изображения в трехмерном пространстве. С помощью OpenGL вы можете управлять технологиями машинной графики, создавая изображения реальных объектов или воплощая другие плоды вашей фантазии. Это руководство объясняет, как создать OpenGL-программу, результатом работы которой является желаемый визуальный эффект.

Книга состоит из 14 глав. Первые пять глав рассказывают, как строить правильно раскрашенные и освещенные объекты в сцене.

- Глава 1, **«Введение в OpenGL»**, коротко описывает возможности OpenGL. Она также содержит простейшую OpenGL-программу и излагает важнейшие программные аспекты, необходимые для понимания следующих глав.
- Глава 2, **«Управление состоянием и рисование геометрических объектов»**, посвящена вопросам создания геометрического описания трехмерного объекта, рисуемого в сцене.
- Глава 3, **«Визуализация»**, описывает процесс преобразования этих трехмерных моделей перед тем, как они будут нарисованы на двухмерном экране. Вы можете управлять преобразованиями для получения желаемого результата.
- Глава 4, **«Цвет»**, объясняет, как определить цвет и метод заливки, используемые при рисовании объекта.
- Глава 5, **«Освещение»**, раскрывает, как управлять условиями освещенности вокруг объекта, и описывает реакцию объекта на освещение (то есть как он отражает или поглощает свет). Освещение — это важная тема, поскольку объекты обычно не выглядят трехмерными, пока их не осветят.

Остальные главы рассматривают, как оптимизировать или ввести дополнительные усложнения в вашу трехмерную сцену. Вероятно, вы не сможете оценить все преимущества дополнительных возможностей до тех пор, пока хорошо не освоите OpenGL. Ссылки на эти главы даются в соответствующих местах книги.

- Глава 6, **«Смешивание, сглаживание, туман и отклонение»**, описывает технологию создания реалистичных изображений: *альфа-смешивание* (для создания эффекта прозрачности), *сглаживание* (для устранения зубчатых граней), *атмосферные эффекты* (для имитации тумана или смога) и *смещение многоугольников* (для удаления визуальных артефактов при высокой освещенности граней залитых многоугольников).

- Глава 7, **«Списки отображения»**, описывает, как сохранить набор команд OpenGL для их повторного использования. Это необходимо для увеличения скорости OpenGL-программ.

- Глава 8, **«Вывод пикселов, битовых образов, шрифтов и изображений»**, обсуждает работу с двухмерными объектами, такими как битовые образы или изображения. Типичное применение битовых образов — описание символов шрифтов.

- Глава 9, **«Наложение текстур»**, показывает, как наложить одно-, двух- или трехмерные изображения, называемые текстурами, на трехмерные объекты. Множество удивительных эффектов достигается наложением текстур.

- Глава 10, **«Буфер кадров»**, описывает все возможные буферы, существующие в OpenGL, и их использование. Буферы применяются для удаления невидимых поверхностей, маскирования, размытия и фокусировки резкости.

- Глава 11, **«Мозаичное представление и поверхности второго порядка»**, показывает, как использовать функции разбиения и квадрирования в GLU (OpenGL Utility Library).

- Глава 12, **«Вычислители и NURBS»**, учит работе с дополнительной техникой эффективного построения кривых и поверхностей.

- Глава 13, **«Выбор и обратная связь»**, объясняет, как используется механизм выбора в OpenGL для выбора объекта на экране. Кроме того, глава рассказывает о механизме обратной связи, позволяющем осуществлять доступ к информации, которую OpenGL выводит при прорисовке и которую можно использовать при рисовании сцены.

- Глава 14, **«Теперь, когда вы знаете многое»**, дает информацию о нестандартном использовании OpenGL для получения интересных результатов. Эти приемы выработаны за годы с момента появления OpenGL и его предшественницы IRIS Graphics Library от Silicon Graphics.

Кроме того, книга содержит несколько приложений, которые могут вам пригодиться.

- Приложение А, **«Порядок выполнения операций»**, дает технический обзор выполнения операций OpenGL, краткое их описание в порядке, в котором они выполняются при запуске программы.
- Приложение Б, **«Параметры состояния»**, содержит список параметров, используемых в OpenGL, и описывает их значения.
- Приложение В, **«OpenGL и оконные системы»**, кратко обозревает библиотеку, управляющую оконными операциями. В нем обсуждаются оконные интерфейсы X Window System, Apple MacIntosh, IBM OS/2 и Microsoft Windows.
- Приложение Г, **«Основы GLUT: инструментальный набор пакета разработчика OpenGL»**, рассматривает библиотеку, управляющую операциями оконной системы. Библиотека GLUT — машинно-независимая, и она позволяет создавать более короткий и понятный программный код.
- Приложение Д, **«Вычисление векторов нормалей»**, объясняет, как вычислить векторы нормалей для различных типов геометрических объектов.
- Приложение Е, **«Однородные координаты и матричные преобразования»**, акцентирует внимание на математических операциях, скрытых за матричными преобразованиями.
- Приложение Ж, **«Приемы программирования»**, содержит несколько основанных на замыслах проектировщиков OpenGL советов по программированию, которые вы можете счесть полезными.
- Приложение З, **«OpenGL и инвариантность»**, уточняет, когда и где при выполнении OpenGL-программ должны генерироваться точные значения пикселов согласно спецификации OpenGL.

Наконец, небольшой словарь определяет ключевые термины, встречающиеся в этом руководстве.

Что нового в этом издании

Четвертое издание *OpenGL Programming Guide* включает в себя новый и обновленный материал, относящийся к OpenGL версий 1.3 и 1.4:

- Новые возможности OpenGL 1.3:
  - мультивыборка;
  - строковый порядок матриц;
  - расширенное текстурирование, включая «кубические» текстуры, сжатые текстурные изображения, новые методы режима текстурирования (`GL_ADD` и `GL_COMBINE`) и новый метод сжатия текстур (`GL_CLAMP_TO_BORDER`).
- Новые возможности OpenGL 1.4:
  - многократное рисование массива вершин;
  - спецификация положения растра, основанная не на внешних, а на оконных координатах;

  - новая функциональность смешивания (blending), в том числе разделение RGB и альфа-смешивание;
  - координаты тумана, которые могут быть однозначно связаны с вершинами;
  - параметры точки, управляющие атрибутами отрисовки точечных примитивов;
  - вторичный цвет для приложений с посттекстуризацией;
  - новые возможности текстурирования, такие как уровень детализации для MIP-текстур, автоматическое восстановление MIP-текстур, режим текстурирования `GL_MIRRORED_REPEAT` и глубина текстур;
  - операции, которые позволяют «оборачивать» трафарет вокруг объекта, а не только привязывать его к минимальному и максимальному значениям.
- Ядро OpenGL было обновлено и приведено в соответствие с набором возможностей мультитекстурирования.
- Исправлены ошибки.

Что необходимо знать перед прочтением

Для того чтобы понимать содержание книги, нужно уметь программировать на языке C и иметь начальные знания по математике (геометрии, тригонометрии, линейной алгебре, дифференциальному исчислению и дифференциальной геометрии). Даже если у вас небольшой опыт в компьютерной графике или его нет совсем, вам должно быть понятно большинство разделов книги. Конечно, машинная графика — неисчерпаемая тема, поэтому вы можете дополнительно прочитать следующие материалы:

- *Computer Graphics: Principles and Practice by James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes (Addison-Wesley, 1990)*. Эта книга — энциклопедия компьютерной графики. В нее входит огромное количество информации, но лучше читать ее, уже обладая опытом работы в этой области.
- *3D Computer Graphics by Andrew S. Glassner (The Lyons Press, 1994)*. Книга дает короткое нетехническое введение в компьютерную графику. Она больше ориентирована на визуальные эффекты, чем на технические проблемы их получения.[1]

Другое замечательное место, где вы можете получить практически любую информацию, — официальный веб-сайт OpenGL. На этом сайте находятся документация, консультации и новости. Это то место, откуда стоит начинать искать ответ на любой вопрос, возникающий при работе с OpenGL. Сайт расположен по адресу: http://www.opengl.org/.

[1] Рекомендуем на русском языке: *Хилл Ф.* OpenGL. Программирование компьютерной графики. Для профессионалов. — СПб.: Питер, 2002. — *Примеч. ред.*

Если вы начинаете программировать с использованием OpenGL, вам наверняка пригодится руководство *OpenGL Reference Manual*, подготовленное *Architecture Review Board* (изданное *Addison–Wesley*), которое было создано как дополнение к данной книге. Это справочное руководство смотрит на операции OpenGL с технической стороны; от информации, описывающей геометрические объекты, до получения изображения на экране. В нем также имеется полное описание всего набора команд OpenGL — аргументы функций, значения по умолчанию и коды возврата. Многие реализации OpenGL предоставляют интерактивный доступ к аналогичному материалу в виде руководства или другого справочного материала, который является более современным. Также много информации можно найти в Интернете; посмотрите на уже упомянутом официальном веб-сайте OpenGL.

OpenGL — по-настоящему аппаратно-независимая спецификация интерфейса программирования, работающая на различном оборудовании. Это руководство рассказывает, как программировать, используя любую реализацию OpenGL. Однако поскольку реализации могут сильно различаться, например по производительности и по дополнительным возможностям, возможно, вам захочется найти соответствующую документацию. Кроме того, наверное, вы пожелаете воспользоваться дополнительными относительно вашей реализации инструментами, пакетами разработчика, другими программными и отладочными средствами, виджетами, примерами программ и презентациями.

Как получить примеры программ

В книге содержится множество примеров программ, показывающих использование различных технологий OpenGL. Эти программы созданы с помощью *OpenGL Utility Toolkit* (GLUT) Марка Килгарда (Mark Kilgard). Документацию по GLUT его же авторства можно найти в *OpenGL Programming for the X Window System* (издательство Addison—Wesley, 1996). За информацией по работе с GLUT обращайтесь к разделу «Библиотеки, связанные с OpenGL» главы 1 и приложению Г. Если у вас есть доступ в Интернет, вы можете получить исходные коды примеров программ и GLUT на анонимных FTP-серверах.

Исходные коды примеров программ из этой книги находятся по следующим адресам (выберите нужный файл в зависимости от архиватора, установленного в вашей системе):

ftp://ftp.sgi.com/opengl/opengl14.tar.Z

ftp://ftp.sgi.com/opengl/opengl14.tgz

ftp://ftp.sgi.com/opengl/opengl14.zip

Используйте соответствующие инструменты для распаковки и извлечения исходного кода из полученных файлов.

Исходные коды Марка Килгарда для GLUT (для Microsoft Windows или X Windows) доступны по адресу:

http://www.opengl.org/developers/documentation/glut/index.html

Многие реализации OpenGL также содержат примеры. Эти исходные коды, скорее всего, являются лучшими для вашей реализации, оптимизированными под нее. Где их взять, вы узнаете из документации для вашей версии OpenGL.

Обучающие программы по OpenGL от Нэйта Робинса

Нэйт Робинс (Nate Robins) написал комплект программ, обучающих основам программирования в OpenGL. Они предоставляют пользователю возможность изменения аргументов функций и наглядную демонстрацию результата изменения. В их число входят примеры по преобразованиям, освещенности, туману и текстурированию. Эти прекрасные небольшие примеры созданы с помощью уже упомянутой GLUT. Их исходные тексты можно получить на странице:

http://www.xmission.com/~nate/tutors.html

Используемые стили

В данном руководстве используются следующие стили:

- **Жирный** — названия команд и операций, матриц.
- *Курсив* — переменные, аргументы, названия параметров, размеры, элементы матриц, первое использование ключевого термина.
- Примеры кодов выделены в тексте моноширинным шрифтом.

Так выделено краткое описание команд.

В описании команд фигурные скобки используются для указания типов данных. В следующем примере `glCommand` имеет четыре суффикса: `s`, `i`, `f` и `d`, соответствующие типам GLshort, GLint, GLfloat и GLdouble. В прототипе функции `glCommand`, `TYPE` заменяет тип данных, определяемый суффиксом:

```
void glCommand{sifd}(TYPE x1, TYPE y1, TYPE x2, TYPE y2)
```

От издателя перевода

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты: comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты программ, приведенные в книге, а также цветные рисунки к книге вы найдете по адресу: http://www.piter.com/download.

На веб-сайте издательства http://www.piter.com представлена подробная информация о других наших книгах.

Благодарности

Успешное развитие OpenGL стало возможным благодаря работе множества людей. Участники *OpenGL Architecture Review Board* помогают двигать OpenGL вперед. Их вклад оценен по достоинству.

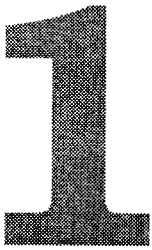
Множество примеров программ написаны Стэйсом Питерсоном (Stace Peterson). Полезные пояснения и дополнения были внесены Мэриэнн Симмонс (Maryann Simmons), Патриком Броуном (Patrick Brown), Аланом Коммайком (Alan Commike), Брэдом Грэнтхэмом (Brad Grantham), Бобом Куэне (Bob Kuehne), Джоном Личем (Jon Leech), Бенджаменом Липчаком (Benjamin Lipchak), Марком Олано (Marc Olano) и Вики Шрейнер (Vicki Shreiner).

Также очень помог редакторский и издательский коллектив Addison–Wesley. Особая благодарность Мэри О'Брайен (Mary O'Brien), Джону Фуллеру (John Fuller) и Бренде Маллиган (Brenda Mulligan).

Цветные рисунки[1] были пересмотрены для этой редакции. Цветные рисунки с 1-го по 9-й были созданы Тэдом Бейером (Thad Beier), Сетом Кацем (Seth Katz) и Мейсоном Ву (Mason Woo). Цветные рисунки с 10-го по 20-й, 22-й и 23-й были получены с помощью программ Мэйсона Ву. Цветной рисунок 21 изготовил Пол Хэберли (Paul Haeberli). Вкладка 24 сделана Сирилом Кардасевичем (Cyril Kardassevitch) из *Institute de Recherche en Informatique de Toulouse*. Цветной рисунок 25 создан Юкари Ито (Yukari Ito) и Кейсуке Кирии (Keisuke Kirii) из *Nihon SGI*. Цветной рисунок 26 сделан Джоном Когги (John Coggi) и Дэвидом Стодденом (David Stodden) из Аэрокосмической компании (*The Aerospace Company*). Цветной рисунок 27 был сделан Рейнером Гебелем (Rainer Goebel) из *Max Planck Institute for Brain Research*. Цветной рисунок 28 был сделан Стефаном Брабеком (Stefan Brabec) и Вольфгангом Гейдрихом (Wolfgang Heidrich) из *Max Planck Institute for Computer Science*. Цветной рисунок 29 сделан Микко Бломквистом (Mikko Blomqvist) из *Mediaclick OY*. Цветной рисунок 30 сделан Берндом Луцем (Bernd Lutz) из *Fraunhofer IGD*. И наконец, цветные рисунки 31 и 32 — это экранные снимки из игр Quake-серии, придуманных *id Software*.

---

[1] Цветные рисунки к книге см. на сайте издательства http://www.piter.com/download. — *Примеч. ред.*



Введение в OpenGL

После прочтения этой главы вы сможете:

- Ориентироваться в основных терминах OpenGL
- Различать различные уровни сложности построения изображения
- Понимать структуру программ OpenGL
- Распознавать синтаксис команд OpenGL
- Различать последовательности операций, выполняющих построение изображения
- Разбираться в основных терминах, имеющих отношение к анимации графики с помощью OpenGL

Эта глава — вводная. Она состоит из следующих основных частей:

- **Что такое OpenGL.** Раскрывает, что такое OpenGL, что и как можно и чего нельзя сделать с его помощью.
- **Немного программного кода.** Содержит небольшие OpenGL-программы и объясняет их работу. Здесь же поясняются несколько основных терминов компьютерной графики.
- **Синтаксис команд OpenGL.** Представляет некоторые соглашения и условные обозначения, используемые в командах OpenGL.
- **OpenGL как конечный автомат.** Описывает использование параметров состояния в OpenGL и команд запроса, включения и выключения состояний.
- **Конвейер рисования OpenGL.** Показывает типичные последовательности операций для обработки геометрических данных и изображений.
- **Библиотеки, связанные с OpenGL.** Описывает наборы связанных с OpenGL подпрограмм, в том числе содержит подробный обзор платформно-независимого пакета разработчика GLUT (Graphics Library Utility Toolkit).
- **Анимация.** Раскрывает в общих терминах процесс создания подвижных изображений на экране.

Что такое OpenGL

OpenGL — это программный интерфейс к графическому оборудованию. Этот интерфейс содержит порядка 250 отдельных команд (около 200 в ядре OpenGL и еще 50 в библиотеке OpenGL Utility Library), которые используются для задания объектов и операций, необходимых для создания интерактивных приложений трехмерной графики.

OpenGL разработан как аппаратно-независимый интерфейс для работы на различных аппаратных платформах. Поэтому в OpenGL не включены команды для управления окнами и для организации пользовательского ввода. Вся такая работа ведется через операционную систему. Аналогично, OpenGL не имеет высокоуровневых команд для описания трехмерных сложных моделей: автомобилей, частей человеческого тела, самолетов или молекул. Задача OpenGL — помочь вам построить вашу модель из небольшого набора *графических примитивов*: точек, линий и многоугольников.

Высокоуровневые же средства предоставляются библиотеками, являющимися надстройками над OpenGL. Для моделирования кривых и поверхностей предназначена библиотека GLU (OpenGL Utility Library), имеющая множество инструментов, таких как вычислители и NURBS. Библиотека GLU является стандартной составляющей любой реализации OpenGL.

Теперь, когда вы знаете, что OpenGL не умеет делать, пришла пора выяснить, что же делать можно. Посмотрите на цветные рисунки[1], иллюстрирующие типичное использование OpenGL. На них изображена сцена, *построенная* (так называемый *рендеринг*) компьютером в порядке постепенного усложнения задачи. Ниже объясняется, как получен этот рисунок.

- Цветной рисунок 1 показывает сцену в виде *каркасной* модели — как будто сплетенной из проволоки. Каждая линия соответствует границе примитива (многоугольника). Например, поверхность стола состоит из треугольников. Впечатление такое, как если бы мы разрезали пирог на ломтики.

  Заметим, что в таких условиях можно видеть даже те части объектов, которые были бы скрыты в случае полноценной — залитой — поверхности. Например, можно видеть холмы за окном, большая часть которых в этой модели скрыта стеной комнаты. Глобус выглядит почти сплошным, так как он состоит из сотни цветных блоков, и взгляду доступны линии каркаса для всех граней всех блоков, даже тех, что образуют обратную сторону глобуса. Пример глобуса показывает, как можно создавать сложные объекты путем сборки из простых (низкоуровневых) объектов.

- Цветной рисунок 2 показывает *объемное* представление этой же каркасной сцены. Заметим, что линии, расположенные дальше от наблюдателя, бледнее, как и в реальной жизни, за счет чего появляется визуальное восприятие *глубины*. Для получения эффекта глубины OpenGL использует также атмосферные эффекты (*туман*).

- Цветной рисунок 3 демонстрирует версию каркасной сцены с устраненной ступенчатостью. Эффект *ступенчатости* (aliasing) возникает в результате растеризации векторных примитивов на *пикселах*, составляющих прямоугольную сетку. Такая ступенчатость наиболее заметна на линиях, близких к горизонтали или вертикали.

- Цветной рисунок 4 демонстрирует лишенную полутонов, неосвещенную версию сцены (без соответствующих эффектов освещения). Объекты в сцене показаны сплошными — для закраски каждого многоугольника используется только один цвет. Потому они выглядят «плоскими», а не округлыми.

- На цветном рисунке 5 нарисована уже *освещенная, с полутонами* версия сцены. Видно, что сцена более реалистична и трехмерна за счет того, что полутона находятся в гармонии с источниками света в комнате, в сравнении с просто объемными объектами.

- На цветном рисунке 6 к сцене добавлены тени и текстуры. *Тени* — это нечетко определенная возможность OpenGL (нет специальной команды «тень»), но вы можете их создавать по технологии, описанной в главах 9 и 14. *Текстурирование* позволяет накладывать двухмерные изображения на трехмерные объекты. В рассматриваемой сцене наиболее ярким примером текстурирования являет-

[1] Цветные рисунки к книге см. на сайте издательства http://www.piter.com/download. — *Примеч. ред.*

ся верхняя часть поверхности стола. Пол и поверхность стола имитированы текстурами (под дерево), аналогично обоям и верхней части игрушки (на столе).

- Цветной рисунок 7 показывает *размывку движением* (motion-blurred) объектов в сцене. Сфинкс (или собака, согласно мотивациям Роршаха) кажется застывшим в движении вперед, оставляет смазанный след на пути своего движения.
- Цветной рисунок 8 показывает сцену с обложки книги с другой точки, доказывая, что картинка действительно является снимком с трехмерных объектов.
- Цветной рисунок 9 возвращает нас к туману, который применен на цветном рисунке 2 для получения эффекта задымленности воздуха. Заметьте, что, в отличие от цветного рисунка 2, здесь эффект более впечатляет.
- Цветной рисунок 10 демонстрирует эффект *глубины резкости*, неспособность объектива фотокамеры сделать четкими все объекты. Камера фокусируется только на небольшой части сцены. Объекты, находящиеся дальше или ближе, чем эта область, выглядят слегка размытыми.

Ниже кратко перечислены основные графические операции, выполняемые OpenGL при построении изображения на экране. (См. раздел «Конвейер рисования OpenGL».)

1. Построение моделей из графических примитивов и создание таким образом математического описания объектов. (Примитивами в OpenGL являются точки, линии, многоугольники, изображения и битовые образы.)
2. Расстановка объектов в трехмерном пространстве и выбор выгодной точки обзора всей сцены.
3. Вычисление цвета всех объектов. Цвета могут быть заданы приложением, определяться условиями освещения, получаться наложением текстур на объекты или быть комбинацией этих трех действий.
4. Преобразование математического описания объектов и информации об их цвете в пикселы на экране. Данный процесс называется *растеризацией*.

Во время этих шагов OpenGL может выполнять другие операции, например удаление невидимых частей объектов. Кроме того, после растеризации сцены, но перед ее прорисовкой, при необходимости можно выполнить операции с массивом пикселов.

В некоторых реализациях (совместимых с X Windows) OpenGL работает и в случаях, когда компьютер, на экране которого выводится изображение, не является тем компьютером, на котором выполняется строящая это изображение программа. Такая возможность рассчитана на сетевое соединение компьютеров. В этом случае компьютер, где запускается программа и выполняются команды OpenGL, называется *клиентом*, а компьютер, который получает эти команды и выполняет рисование, называется *сервером*. Формат для передачи команд OpenGL (*протокол*) от клиента к серверу всегда один и тот же. Поэтому программы OpenGL могут работать в *сетях*, даже если клиент и сервер представляют собой компьютеры

разных типов. Если программа OpenGL работает только на одном компьютере, он является одновременно клиентом и сервером.

Немного программного кода

Так как OpenGL позволяет множество различных вещей, программы OpenGL могут быть достаточно сложными и запутанными. Тем не менее общая структура программ проста: инициализация состояний, управляющих рисованием, и указание объектов для рисования.

Перед тем как рассмотреть некоторые примеры, определим несколько терминов. *Отображение*, иначе *рендеринг* (воспроизведение изображения на экране, rendering — этот термин вы уже встречали), — это процесс, с помощью которого компьютер создает изображения на основе моделей. Эти модели, или объекты, вырабатываются из графических примитивов — точек, линий и многоугольников, определяемых *вершинами* (vertices).

Окончательно прорисованное изображение состоит из пикселов на экране. *Пиксел* — это минимальный элемент изображения. Информация о пикселах (например, их цвет) хранится в битовых плоскостях. *Битовая плоскость* — это область памяти, в которой каждому пикселу соответствует один бит информации. Бит, например, может определять красную составляющую цвета. В свою очередь, битовые плоскости образуют видеобуфер (буфер кадров), который хранит всю информацию, необходимую графическому дисплею для управления цветом и яркостью всех пикселов экрана.

Теперь посмотрите, как выглядит программа OpenGL. Программа из листинга 1.1 рисует белый прямоугольник на черном фоне, как показано на рис. 1.1.

Листинг 1.1. Пример кода OpenGL

```
# include <whateverYouNeed.h>
main() {
    InitializeAWindowPlease();
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

    glBegin(GL_POLYGON);
    glVertex3f(0.25, 0.25, 0.0);
    glVertex3f(0.75, 0.25, 0.0);
    glVertex3f(0.75, 0.75, 0.0);
    glVertex3f(0.25, 0.75, 0.0);
    glEnd();
    glFlush();
    UpdateTheWindowAndCheckForEvents();
}
```

Первая строка процедуры `main()` создает окно и выводит его на экран: функция `InitializeAWindowPlease()` обращается к функциям оконной системы, а не

к вызовам OpenGL — это очевидное расположение функции инициализации. Следующие две строки — команды OpenGL для очистки окна (закрашивания его черным цветом): `glClearColor()` устанавливает цвет очистки окна, а `glClear()` выполняет закраску. После установки цвета очистки все последующие вызовы `glClear()` очищают окно этим цветом. Цвет очистки может быть изменен другим вызовом функции `glClearColor()`. Функция же `glColor3f()` устанавливает цвет, используемый для рисования объектов, в данном случае — белый. Все объекты, рисуемые после этой строки, будут белыми, пока цвет не будет изменен другой командой.

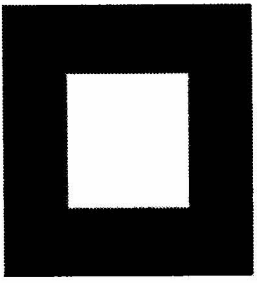


Рис. 1.1. Белый прямоугольник на черном фоне

Команда `glOrtho()`, задающая *систему координат* OpenGL, определяет вид конечного изображения и способ его отображения на экране. Вызовы, заключенные в операторные скобки `glBegin()` и `glEnd()`, определяют объекты рисования — в данном примере это многоугольник с четырьмя вершинами. «Углы» многоугольника заданы функциями `glVertex3f()`. Как видно из значений аргументов, определяющих координаты (x, y, z), многоугольником является прямоугольник, расположенный на плоскости $z = 0$.

Наконец, команда `glFlush()` гарантирует, что команды рисования будут сразу выполнены, а не сохранены в *буфере* в ожидании дополнительных команд OpenGL. Процедура `UpdateTheWindowAndCheckForEvents()` («местозаполнитель», как и `InitializeAWindowPlease()`) управляет содержимым окна и начинает обработку событий.

Фактически эта часть кода OpenGL не структурирована хорошо. Вы можете спросить: «А что случится, если я попробую переместить окно или изменить его размеры?» или «Нужно ли устанавливать систему координат каждый раз, когда я рисую прямоугольник?» Позже в этой главе мы покажем, как выполнить замену функций `InitializeAWindowPlease()` и `UpdateTheWindowAndCheckForEvents()` на более эффективный код.

Синтаксис команд OpenGL

Как видно из только что представленной программы, для команд OpenGL используется префикс `gl`, а первые буквы каждого слова имени команды являются заглавными (например, `glClearColor()`). Константы в OpenGL начинаются с `GL_`,

записываются прописными буквами, а отдельные слова в них выделяются подчеркиванием (например, GL_COLOR_BUFFER_BIT).

Наверное, вы обратили внимание, что к некоторым именам команд добавляются суффиксы (например, 3f, как в glColor3f() и glVertex3f()). На самом деле слова Color в имени достаточно для того, чтобы определить команду, которая устанавливает текущий цвет. Однако существует несколько таких команд, для различных типов аргументов. В частности, цифра 3 в суффиксе указывает, что команда имеет три аргумента; в другой версии команды Color используются четыре аргумента. Буква f в суффиксе указывает, что аргументы — типа floatingpoint (вещественные числа). Таким образом OpenGL приобретает возможность принимать данные пользователя в угодных последнему форматах данных.

Для некоторых команд OpenGL используются аргументы восьми различных типов данных. Буквы-суффиксы этих типов данных соответствуют стандарту ISO для языка C, они приведены в табл. 1.1 вместе с определением типов OpenGL. Используемая вами реализация OpenGL может в точности не повторять эту схему, как, например, в реализации C++ или Ада.

Таблица 1.1. Суффиксы команд и типы аргументов

Суффикс	Тип данных	Типичное соответствие в языке C	Тип данных в OpenGL
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int или unsigned long	GLuint, GLenum, GLbitfield

Таким образом, команды

```
glVertex2i(1, 3);
glVertex2f(1.0, 3.0);
```

эквивалентны, за исключением того, что первая определяет координаты вершин как переменные 32-битного целого типа, а вторая — как числа вещественного типа с одинарной точностью.

ПРИМЕЧАНИЕ

В различных реализациях OpenGL соглашения о соответствии между типами данных C и OpenGL не обязательно одинаковы. Если вы будете использовать в своем приложении типы данных, определяемые OpenGL, у вас не возникнет проблем с переносом кода между различными реализациями.

Некоторые команды OpenGL заканчиваются буквой v, которая показывает, что команда принимает указатель на вектор (или массив) значений, а не несколько различных аргументов. Для многих команд допускается задание аргументов

и в виде векторов, и в виде набора аргументов, но для отдельных команд возможны только либо набор значений, либо только векторная форма. В следующих строках показано использование векторной и невекторной версий команд для задания цвета:

```
glColor3f(1.0, 0.0, 0.0);

GLfloat color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);
```

Наконец, в OpenGL определен тип `GLvoid`. Наиболее часто он применяется в командах, аргументами которых являются указатели на массив значений.

В данном руководстве (за исключением реальных примеров кода) команды OpenGL обозначаются только базовым именем, а звездочка означает, что существует несколько версий данной команды. Например, `glColor*()` подразумевает все версии команды, используемой для установки текущего цвета. Если будет нужно акцентировать внимание на конкретной версии команды, к имени будет добавлен необходимый суффикс. Так, `glVertex*v()` обозначает все векторные версии команды определения вершин.

OpenGL как конечный автомат

OpenGL — это конечный автомат. Вы переводите его в различные состояния (или режимы), в которых он остается до момента следующего изменения. Как вы уже видели, текущий цвет — это *параметр состояния*. Вы можете его устанавливать в белый, красный или какой-либо другой цвет, и после этого все объекты будут отображаться этим цветом до тех пор, пока вы его не смените каким-то другим. Текущий цвет — один из многих параметров состояния OpenGL. Другие отвечают за текущее модельно-видовое преобразование, тип линий и многоугольников, методы отображения многоугольников, режим упаковки пикселов, расположение и характеристики источников света, свойства материала объектов. Многие параметры состояния связаны с режимами, которые включаются и выключаются командами `glEnable()` и `glDisable()`.

Каждый параметр состояния или режим имеет значение по умолчанию, и в любой момент можно запросить у системы текущее состояние. Обычно для этого используется одна из шести команд: `glGetBooleanv()`, `glGetDoublev()`, `glGetFloatv()`, `glGetIntegerv()`, `glGetPointerv()` и `glIsEnabled()`. Команда выбирается в соответствии с типом данных параметра. Для некоторых параметров состояния используются более специфичные команды-запросы (такие как `glGetLight*()`, `glGetError()` или `glGetPolygonStipple()`). Кроме того, можно сохранить набор параметров состояния в стеке командой `glPushAttrib()` или `glPushClientAttrib()`, временно их изменить и позднее восстановить значения командой `glPopAttrib()` или `glPopClientAttrib()`. При временном изменении состояний использование этих команд предпочтительнее, так как они более эффективны.

В приложении Б приведен полный список параметров состояния, которые можно запросить. Для каждого параметра в приложении также перечислены соответствующие команды `glGet*()`, тип возвращаемого значения и значения по умолчанию.

Конвейер рисования OpenGL

Большинство реализаций OpenGL имеет сходную последовательность операций, серию этапов обработки, называемую конвейером рисования OpenGL. Этот порядок, как показано на рис. 1.2, не регламентирует строго действия OpenGL, а, скорее, представляет собой руководство к тому, что делает OpenGL.

Если вы новичок в трехмерной графике, предстоящее описание может оказаться сравнимо с питьем воды из пожарного гидранта. Вы можете проглядеть текст и рисунок сейчас и возвращаться к ним позже, по мере прочтения каждой главы этой книги.

Для обработки данных OpenGL взял на вооружение подход, примененный Генри Фордом в автомобилестроении. Векторные данные (вершины, линии и многоугольник) подвергаются конвейерной обработке рядом блоков, включая блоки оценки и операции с вершинами, а растровые данные (пикселы, изображения и битовые образы) обрабатываются отдельно. Оба типа данных проходят одни и те же финальные стадии (растеризацию и операции над фрагментами) перед тем, как окончательный набор пикселов будет записан в буфер кадров.

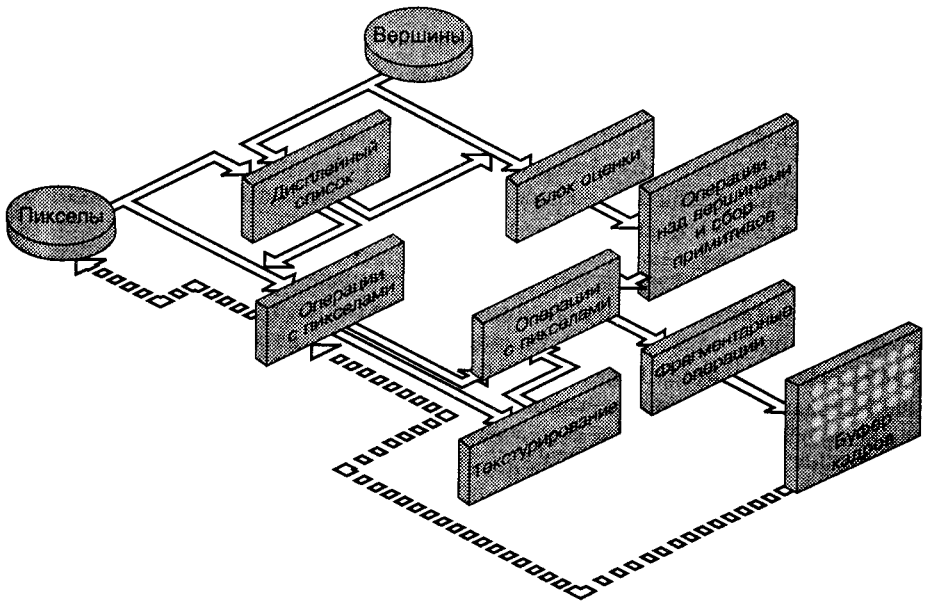


Рис. 1.2. Порядок операций

Теперь рассмотрим некоторые ключевые стадии конвейера рисования OpenGL более подробно.

Списки отображения

Все данные, как векторные, так и растровые, могут быть сохранены в списках отображения для текущего или последующего использования. (Альтернативой хранению данных в списках отображения является непосредственное использование данных, также известное как *непосредственный режим.*) При выполнении списка отображения хранящаяся в нем информация посылается на обработку так же, как если бы она посылалась на обработку приложением в непосредственном режиме. (См. главу 7 для получения более подробной информации о списках отображения.)

Вычислители

Все геометрические примитивы в конечном итоге описываются вершинами. Параметрические кривые и поверхности могут быть первоначально описаны контрольными точками и многочленами — *базисными функциями. Вычислители* (evaluators) предоставляют способ для получения (генерации) вершин, применяемый для представления поверхности, заданной контрольными точками.

Метод полиномиального отображения вырабатывает нормали к поверхности, координаты текстур, цвета и значения пространственных координат из набора контрольных точек. (См. главу 12 для получения более подробной информации.)

Операции над вершинами

Следующей стадией обработки являются операции над вершинами, преобразующие вершины в примитивы. Некоторая информация о вершинах (например, пространственные координаты) преобразуется с использованием матриц размерностью 4 × 4. Пространственные координаты проецируются из положения в трехмерном пространстве в положение на экране. (См. главу 3 для получения более подробной информации о матричных преобразованиях.)

Если привлекаются дополнительные возможности, эта стадия расширяется. При использовании текстурирования могут генерироваться и преобразовываться координаты текстур. Если задействуется освещенность, выполняются расчеты значений цвета исходя из данных преобразованных вершин, нормалей к поверхности, положения источников света, свойств материалов и другой информации.

Компоновка примитивов

Отсечение, важнейшая часть сборки примитивов, — это удаление частей геометрических объектов, которые выходят за границы полупространства, определенного плоскостью. Точечное отсечение обычно отбрасывает вершины; отсечение линий или многоугольников может добавлять дополнительные вершины, в зависимости от того, как линия или многоугольник отсекаются.

В некоторых случаях это происходит за счет разделения по перспективе, когда удаленные объекты кажутся меньшими, чем близкие. Следующие на очереди — точка наблюдения и операции глубины (координата z). Если включена отбраковка и в качестве примитива используется многоугольник, отсечение может быть выполнено за счет отбраковывающего теста. В зависимости от режима отображения многоугольника, он может быть нарисован точками или линиями. (См. раздел «Подробно о многоугольниках» главы 2.)

Результатом этого этапа являются законченные графические примитивы, вершины которых преобразованы и отсечены; с вершинами связаны цвет, глубина и иногда координаты текстур, а также рекомендации к растеризации.

Операции над пикселами

В то время как векторные данные проходят по одному пути в конвейере рисования OpenGL, растровые данные следуют другим маршрутом. Пикселы из массива системной памяти сначала распаковываются из одного из множества различных форматов в некоторое число компонентов. Затем данные масштабируются, трансформируются и обрабатываются как пиксельные карты. Результаты накапливаются, а затем записываются в память текстур или отправляются на этап растеризации. (См. раздел «Конвейерная обработка изображений» главы 8.)

Если растровые данные считываются из видеобуфера, выполняются операции при пересылке пикселов (масштабирование, смещение, отображение и ограничивание). Затем результаты упаковываются согласно формату и возвращаются в массив в системной памяти.

Существуют специальные пиксельные операции для копирования информации из одной части видеобуфера в другую или в память текстур. Операции при пересылке пикселов выполняются однократно перед тем, как данные записываются в память текстур или возвращаются в видеобуфер.

Компоновка текстур

Приложения OpenGL могут накладывать текстурные изображения на графические объекты с целью придания последним большего реализма. Если используется несколько изображений текстур, представляется разумным поместить их в текстурные объекты для того, чтобы переключаться между изображениями стало просто.

Некоторые реализации OpenGL имеют специальные средства для увеличения производительности текстурирования. Например, может существовать специальная, высокопроизводительная память текстур. Если такая память присутствует, текстурным объектам могут назначаться приоритеты для контроля над использованием этого ограниченного и очень ценного ресурса.

Растеризация

Растеризация — это преобразование векторных и растровых данных во *фрагменты*. Каждому квадратному фрагменту соответствует пиксел в видеобуфере.

Учитываются прерывистые линии и многоугольники, ширина линии, размер точек, модель закраски и вычисление покрытия для поддержки сглаживания, так как вершины связаны в линии, или вычисляются внутренние пикселы для закрашенных многоугольников. С каждым фрагментом связаны значения цвета и глубины.

Операции над фрагментами

Перед записью значений в видеобуфер выполняется ряд операций, которые могут изменить или даже отбросить фрагменты. Все эти операции могут быть разрешены или запрещены (включены или выключены).

Первая ожидаемая операция — текстурирование, когда *тексел* (элемент текстуры) генерируется из памяти текстур для каждого фрагмента и применяется к фрагменту. Затем может выполняться вычисление тумана, затем тест отсечения, альфа-тест, тест трафарета и тест буфера глубины (буфер глубины связан с удалением невидимых поверхностей). Неудачное завершение теста может прервать процесс фрагментации. Далее производятся смешивание, размытие, логические операции и наложение битовой маски (см. главы 6 и 10). Наконец сформированные фрагменты рисуются в соответствующем буфере, где из них формируются пикселы и определяется их окончательное местоположение.

Библиотеки, связанные с OpenGL

OpenGL предоставляет мощный, но простой набор команд, и все высокоуровневое рисование должно быть определено в терминах этих команд. К тому же программы OpenGL используют базовые механизмы оконной системы. Несколько библиотек, приведенных ниже, позволяют упростить решение программных задач:

- The OpenGL Utility Library (GLU) содержит несколько команд, использующих низкоуровневые команды OpenGL, которые позволяют устанавливать значения видовых матриц и матриц проекций, выполнять разбивку на многоугольники и рендеринг поверхностей. Эта библиотека входит в состав любой реализации OpenGL. Составляющие GLU описаны в *OpenGL Reference Manual.* Наиболее полезные команды GLU описаны в данной книге, в соответствующих разделах, как, например, в главе 11 и в разделе «Интерфейс NURBS библиотеки GLU» главы 12. Все команды GLU начинаются с `glu`.
- Для каждой оконной системы существуют библиотеки, расширяющие функциональные возможности OpenGL по поддержке данной оконной системы. Для компьютеров, использующих систему X Windows, таким расширением является библиотека GLX (OpenGL Extension to the X Windows System). Команды GLX начинаются с префикса `glX`. Для Microsoft Windows такой библиотекой является WGL. Все команды WGL начинаются с `wgl`. Для OS/2 от IBM используется PGL (Presentation Manager to OpenGL), и ее команды на-

чинаются с `pgl`. Для Apple используется библиотека AGL, команды которой начинаются с `agl`.

Все эти расширения для оконной системы более подробно описаны в приложении B. Кроме того, команды GLX представлены в *OpenGL Reference Manual.*

- GLUT (The OpenGL Utility Toolkit) — это независимый от оконной системы набор инструментов, написанных Марком Килгардом (Mark Kilgard) для того, чтобы устранить различия между различными API по управлению окнами. Библиотека GLUT рассматривается в следующем разделе и более подробно описана в книге Марка Килгарда «*OpenGL Programming for the X Window System*» (издательство *Addison–Wesley*, 1996). Команды GLUT легко опознаются по префиксу `glut`. Если вас интересуют исходные коды для GLUT, см. раздел «Как получить примеры программ» этой книги.

Включаемые файлы

Обычно в заголовочные данные каждого файла OpenGL-приложения включается файл `gl.h`. Почти все приложения OpenGL используют GLU, вышеупомянутую библиотеку утилит OpenGL, которая тянет за собой заголовочный файл `glu.h`. Итак, почти все исходные тексты программ OpenGL начинаются так:

```
#include <GL/gl.h>
#include <GL/glu.h>
```

ПРИМЕЧАНИЕ

При работе с Microsoft Windows до присоединения gl.h и glu.h требуется включить windows.h, так как некоторые макросы, используемые в версиях gl.h и glu.h для Microsoft Windows, определены в windows.h.

Библиотеки OpenGL постоянно изменяются. Различные производители графического оборудования добавляют новые возможности, которые могут быть не учтены в `gl.h`. Для их использования в OpenGL имеется дополнительный заголовочный файл `glext.h`. Этот файл содержит последние дополнения, и его можно получить на веб-сайте производителя оборудования или на официальном сайте OpenGL (http://www.opengl.org). Как и любой заголовочный файл, он включается следующей командой:

```
#include "glext.h"
```

Вы, должно быть, заметили, что имя файла заключено в кавычки, а не в скобки. Это неспроста. Так как `glext.h` предоставляет доступ к новым возможностям, введенным производителями видеокарт, вам лучше всего будет взять последнюю версию из Интернета для того, чтобы успешно компилировать программы. Кроме того, вы можете столкнуться с проблемой отсутствия разрешения на запись файла `glext.h` в системный каталог заголовочных файлов (например, /usr/include) в системах UNIX.

Если вы используете дополнительные интерфейсные библиотеки, такие как GLX, AGL, PGL или WGL, вам нужно включить дополнительные заголовочные

файлы. Например, для GLX необходимо добавить следующие строки в программный код:

```
#include <X11/Xlib.h>
#include <GL/glx.h>
```

В Microsoft Windows для библиотеки WGL достаточно указать

```
#include <windows.h>
```

Если вы используете GLUT, вам нужно добавить строку

```
#include <GL/glut.h>
```

ПРИМЕЧАНИЕ

Подключение glut.h гарантирует, что gl.h и glu.h также подключены, поэтому подключение всех трех библиотек является избыточным. Кроме того, glut.h до включения gl.h и glu.h определяет все используемые макросы, все зависимые от операционной системы. Поэтому, чтобы уменьшить вашу программу, использующую GLUT, подключите только glut.h.

Большинство программ OpenGL также обращаются к стандартным системным библиотекам C, то есть требуют заголовочные файлы, не связанные с графикой, например:

```
#include <stdlib.h>
#include <stdio.h>
```

В текстах наших примеров определения заголовочных файлов отсутствуют по причине обычной удобочитаемости.

GLUT, набор инструментов OpenGL

Как известно, библиотека OpenGL содержит команды отрисовки, причем она разрабатывалась платформо-независимой. Поэтому в ней нет команд для открытия окон или обработки событий от клавиатуры или мыши. К сожалению, невозможно написать полноценную графическую программу без, как минимум, открытия окна, а более интересные приложения требуют пользовательского ввода или других сервисных функций операционной системы или оконной системы. Во многих случаях законченные программы представляют больший интерес, поэтому в данной книге используется GLUT для упрощения работы с окнами, определения ввода и т. д. Если у вас есть реализации OpenGL и GLUT, примеры из данной книги должны запускаться без внесения каких-либо изменений.

Кроме того, в отличие от команд рисования OpenGL, которые позволяют строить только простые геометрические примитивы (точки, линии и многоугольники), GLUT предоставляет возможности для создания более сложных трехмерных объектов, таких как сфера, тор и чайник. Поэтому снимки экранов результатов выполнения программ представляют определенный интерес. (GLU также имеет команды для создания некоторых трехмерных объектов, таких как сфера, цилиндр и конус.)

Управление окнами

Пять команд выполняют задачи, необходимые для определения окна:

- `glutInit`(int *argc*, char ***argv*) инициализирует GLUT и обрабатывает любые параметры командной строки (для X-систем это могут быть такие параметры, как `-display` и `-geometry`). Команда `glutInit()` должна быть вызвана до любых других команд GLUT.
- `glutInitDisplayMode`(unsigned int *mode*) определяет, какую цветовую модель, RGBA или индексную, следует использовать. Также она может определять режим одиночной или двойной буферизации окна. (Если вы работаете в режиме цветовых индексов, вам необходимо загрузить какую-нибудь цветовую карту с помощью команды `glutSetColor()`.) Наконец, с помощью этой команды можно определить, связано ли окно с глубиной, трафаретом, множественной выборкой и/или буфером накопления. Например, если требуется создать окно с двойной буферизацией, моделью цвета RGBA и буфером глубины, вы можете вызвать функцию следующего вида `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)`.
- `glutInitWindowPosition`(int *x*, int *y*) определяет положение левого верхнего угла окна.
- `glutInitWindowSize`(int *width*, int *size*) определяет размер окна в пикселах.
- `int glutCreateWindow`(char **string*) создает окно с содержимым OpenGL. Она возвращает уникальный идентификатор нового окна.

ПРИМЕЧАНИЕ

Пока не будет вызвана функция glutMainLoop(), окно отображено не будет!

Обратная связь

`glutDisplayFunc`(void (**func*)(void)) — первая и наиболее важная функция обратной связи. Когда GLUT определяет, какое содержимое окна нужно перерисовать, выполняется функция обратной связи, зарегистрированная с помощью `glutDisplayFunc()`. Поэтому все команды для перерисовки сцены следует размещать в функции обратной связи.

Если программа изменяет содержимое окна, иногда вызывают функцию `glutPostRedisplay()`, которая заставляет функцию `glutMainLoop()` вызвать зарегистрированную функцию обратной связи при первом удобном случае.

Запуск программы

Последнее, что необходимо сделать, — вызвать функцию `glutMainLoop` (void). После этого будут показаны все созданные окна вместе с их содержимым. Начнется обработка событий, и сработает привязка функции обратной связи. После того как цикл обработки начнет работать, он никогда не прекратится!

Листинг 1.2 показывает, как с помощью GLUT создать простую программу, показанную в листинге 1.1. Заметьте, что код был переформирован. Для улучшения производительности те операции, которые вызываются один раз (установка цвета фона и системы координат), размещены в процедуре `init()`. Операции рисования (и возможной перерисовки) сцены находятся в процедуре `display()`, которая зарегистрирована как функция обратной связи GLUT.

Листинг 1.2. Простая программа, использующая GLUT: hello.c

```
void display(void)
{
/*  очищаем все пикселы  */
    glClear(GL_COLOR_BUFFER_BIT);
/* рисуем белый многоугольник (прямоугольник) с углами в (0,25, 0,25, 0,0)
и (0.75, 0.75, 0.0) */
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_POLYGON);
    glVertex3f(0.25, 0.25, 0.0);
    glVertex3f(0.75, 0.25, 0.0);
    glVertex3f(0.75, 0.75, 0.0);
    glVertex3f(0.25, 0.75, 0.0);
    glEnd();
/*  не ждем!начинаем обработку буферизованных команд OpenGL */
    glFlush();
}
void init(void) {
/*  выбираем цвет очистки (фона)       */
    glClearColor(0.0, 0.0, 0.0, 0.0);
/*  определяем значения обзора  */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0); }
/*
Определяем размеры, положение и режим отображения окна
(одиночный буфер и RGBA).  Открываем окно с надписью "hello"
в заголовке.  Вызываем функции инициализации.
Регистрируем функцию обратной связи.
Входим в цикл и обрабатываем события.
*/
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("hello");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
/* Стандарт ISO C требует от функции main
возврата значения типа int. */ }
```

Обработка событий

Вы можете использовать следующие функции для задания реакции на определенные события:

- `glutReshapeFunc` (void (**func*)(int *w*, int *h*)) — реакция на изменение размера окна;

- `glutKeyboardFunc` (void (**func*)(unsigned char *key*, int *x*, int *y*)) и `glutMouseFunc` (void (**func*)(int *button*, int *state*, int *x*, int *y*)) — позволяют обрабатывать нажатия клавиш на клавиатуре и движение мыши и нажатие ее кнопок;
- `glutMotionFunc` (void (**func*)(int *x*, int *y*)) — отклик на перемещение мыши при нажатой кнопке.

Управление фоновым процессом

Вы можете определить функцию, которая будет вызываться, когда никакие события не возникают — например, при зацикливании, с помощью команды `glutIdleFunc`(void (**func*)(void)). Аргументом функции является указатель на функцию обработки. Для отключения обработки необходимо задать в качестве параметра значение `NULL` (нуль).

Рисование трехмерных объектов

GLUT включает в себя ряд функций для построения трехмерных объектов:

cone (конус)	icosahedron (икосаэдр)	teapot (чайник)
cube (куб)	octahedron (октаэдр)	tetrahedron (тетраэдр)
dodecahedron (додекаэдр)	sphere (сфера)	torus (тор)

Вы можете рисовать эти объекты в виде каркасов или залитыми, оттененными, с заданием векторов нормалей. Например, функциями для рисования куба и сферы являются:

void `glutWireCube`(GLdouble *size*);

void `glutSolidCube`(GLdouble *size*);

void `glutWireSphere`(GLdouble *radius*, GLint *slices*, GLint *stacks*);

void `glutSolidSphere`(GLdouble *radius*, GLint *slices*, GLint *stacks*).

Все перечисленные модели рисуются отцентрированными относительно начала координат (см. приложение Г для получения более подробной информации обо всех прототипах этих функций).

Анимация

Одна из наиболее захватывающих вещей компьютерной графики — это создание движущихся изображений. Не важно, на кого рассчитана анимация: на инженера, пытающегося рассмотреть все части разработанного механизма, или пилота, который учится летать на самолете с помощью симулятора, или на страстного поклонника компьютерных игр, ясно, что она является важнейшей частью компьютерной графики.

В кинотеатре движение создается путем последовательного проецирования кадров на экран — 24 раза/с. Каждый кадр помещается под линзу, затвор открывается, и кадр показывается. Затвор моментально закрывается и остается закрытым, пока фильм движется к следующему кадру, и т. д. Несмотря на то что вы видите 24 различных кадра каждую секунду, ваш мозг объединяет их в плавную анимацию. (Старые фильмы Чарли Чаплина снимались с частотой 16 кадров/с, и в них заметна смена кадров.) Экран в компьютерных графических системах обычно обновляется (перерисовывает изображение) примерно от 60 до 76 раз/с, а в некоторых даже около 120 раз/с. Несомненно, что 60 раз/с плавнее, чем 30, и, конечно, 120 лучше, чем 60. Однако частота обновления больше 120 раз/с, в зависимости от восприятия, может вызвать ухудшение качества.

Ключевым моментом в проецировании движения является то, что каждый кадр показывается полностью. Допустим, вы пытаетесь создать компьютерную анимацию фильма из миллиона кадров с помощью следующей программы:

```
open_window();
for (i = 0;  i < 1000000;  i++)   {
    clear_the_window();
    draw_frame(i);
    wait_until_a_24th_of_a_second_is_over();
}
```

Если вы добавите время, затрачиваемое вашей системой на очистку экрана и рисование типичного кадра, эта программа даст тем более плохие результаты, чем ближе время, затраченное на очистку и прорисовку, к 1/24 с. Предположим, что прорисовка занимает около 1/24 с. Элементы, рисуемые первыми, видны в течение всей 1/24 с и представляют собой сплошное изображение на экране; элементы же, рисуемые в конце, мгновенно стираются перед выводом следующего кадра. В лучшем случае получится призрачное изображение, так как большую часть 1/24 с глаз будет видеть фон, а не элементы, которые должны были строиться в конце. Проблема в том, что эта программа показывает не законченный кадр, а его построение.

Большинство реализаций OpenGL обеспечивают возможности технологии двойной буферизации — аппаратной или программной, которая предоставляет два полноценных цветовых буфера. Один отображается, в то время как второй подготавливается. Когда рисование кадра завершено, буферы меняются местами, то есть тот, который был виден, используется для рисования и наоборот. Это похоже на кинопроектор с двумя зацикленными кадрами: пока один проецируется на экран, художник стирает и перерисовывает невидимый кадр. Пока художник достаточно быстр, зритель не различит этот способ и способ, когда все кадры уже нарисованы, и проектор просто отображает кадры друг за другом. С двойной буферизацией каждый кадр показывается, только когда рисование завершено, и зритель никогда не видит нарисованный частично кадр.

Измененная версия программы, которая отображает «сглаженную» анимацию с помощью двойной буферизации, может выглядеть так:

```
open_window_in_double_buffer_mode();
for (i = 0; i < 1000000; i++) {
```

```
    clear_the_window();
    draw_frame(i);
    swap_the_buffers();
}
```

Обновление с задержкой

В некоторых реализациях OpenGL, кроме простой смены видимого и рисуемого буферов, команда `swap_the_buffers()` задерживает обновление экрана до окончания показа предыдущего буфера. Эта функция также позволяет показывать буфер с самого начала целиком. Полагая, что частота обновления 60 раз/с, мы получаем, что максимальная частота смены кадров составляет 60 кадров/с (*fps*), и, если все ваши кадры смогут быть очищены и нарисованы менее чем за 1/60 с, ваша анимация будет смотреться гладко.

Чаще всего кадр оказывается слишком сложным, чтобы быть нарисованным за 1/60 с, и поэтому каждый кадр отображается больше одного раза. Если, например, на рисование кадра затрачивается 1/45 с, вы получите 30 fps, и графика задерживается на 1/30 – 1/45 = 1/90 с на каждый кадр, или на одну треть времени.

Кроме того, частота смены кадров — это константа, которая может принимать различные значения во время выполнения. Например, при частоте обновления монитора 1/60 с значения частоты смены кадров могут быть 60 fps, 30 fps, 20 fps, 15 fps, 12 fps и т. д. (60/1, 60/2, 60/3, 60/4, 60/5, ...). Это значит, что, когда вы пишете программу и добавляете новые возможности (например, это авиасимулятор, и вы добавляете пейзаж местности), сначала каждая новая возможность, которую вы привносите, не влияет на производительность — вы все еще получаете 60 fps. Затем, внезапно частота падает до 30 fps, так как система уже не успевает рисовать все за 1/60 с, а это первое возможное очередное значение обмена буферов. Точно так же, если время рисования кадра становится больше 1/30 с, скорость анимации падает с 30 до 20 fps.

Если сложность сцены близка к одному из периодов (1/60 с, 2/60 с, 3/60 с и т. д.), то из-за случайных изменений некоторые кадры следуют несколько быстрее, а некоторые немного медленнее. То есть частота смены кадров непостоянная, и это может стать заметным визуально. Поэтому, если вы не можете упростить сцену с целью ускоренной подачи кадров, бывает лучше добавить небольшую задержку, чтобы замедлить и выровнять частоту смены кадров. Если разброс большой, могут потребоваться более сложные методы.

Движение = Перерисовка + Обмен

Структура реальной программы, создающей анимацию, не слишком отличается от этого описания. Обычно легче перерисовать целиком буфер из рабочей области для каждого кадра, чем определить, какую часть нужно перерисовать. Это особенно верно для приложений наподобие трехмерных авиасимуляторов, когда небольшое изменение положения самолета изменяет расположение всего, находящегося за окном.

В большинстве анимаций объекты в сцене просто перерисованы с различными изменениями — перемещена *точка обзора* наблюдателя, или машина немного сдвинута на дороге, или объект слегка повернут. Если существенные перерасчеты требуются не для операций перерисовки, частота смены кадров падает. Однако нужно иметь в виду, что время после вызова функции `swap_the_buffer()` часто используется для подобного рода вычислений.

В OpenGL не включена команда `swap_the_buffer()`, так как эта возможность может отсутствовать на аппаратном уровне, и во многих случаях она очень зависима от оконной системы. Например, при наличии прямого доступа к системе X Window, вы можете использовать следующую команду GLX:

```
void glXSwapBuffers(Display *dpy, Windows window).
```

(Подобные команды для других оконных систем описаны в приложении B).

Если используется библиотека GLUT, команда выглядит следующим образом:

```
void glutSawpBuffers(void).
```

Листинг 1.3 демонстрирует использование `glutSwapBuffers()` при рисовании вращающегося квадрата, как показано на рис. 1.3. Это также пример применения GLUT для управления устройством ввода и включения и выключения функций задержки. Здесь кнопки мыши включают и выключают вращение.

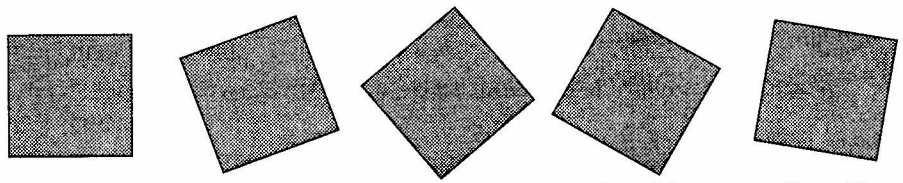


Рис. 1.3. Вращение квадрата с использованием двойной буферизации

Листинг 1.3. Программа с использованием двойной буферизации: double.c

```
static GLfloat spin = 0.0;

void init(void) {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
}

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glRotatef(spin, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
    glRectf(-25.0, -25.0, 25.0, 25.0);
    glPopMatrix();
    glutSwapBuffers();
}

void spinDisplay(void) {
    spin = spin + 2.0;
```

продолжение

Листинг 1.3 (*продолжение*)

```
    if (spin > 3 60.0)
        spin = spin   - 3 60.0;
    glutPostRedisplay();
}

void reshape(int w, int h) {
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
}

void mouse(int button, int state, int x, int y) {
    switch (button) {
            case GLUT_LEFT_BUTTON:
                    if (state == GLUT_DOWN)
                        glutIdleFunc(spinDisplay); break;
            case GLUT_MIDDLE_BUTTON:
                    if (state == GLUT_DOWN)
                        glutIdleFunc(NULL); break;
            default:
                    break;
    }
}
/*
Необходим режим двойной буферизации
Регистрация функции обратной связи на ввод от мыши
*/
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutMainLoop();
    return 0;
}
```

Управление состоянием и рисование геометрических объектов

Прочитав эту главу, вы сможете:

- Очистить окно и закрасить его произвольным цветом
- Принудительно ускорить завершение любого незаконченного рисования
- Рисовать с использованием любых графических примитивов — точек, линий и многоугольников в двух- или трехмерном пространстве
- Переключать состояния и запрашивать значения параметров состояния
- Управлять отображением графических примитивов — например, рисовать пунктирные линии или очерченные многоугольники
- Определять векторы нормалей в соответствующих точках поверхностей закрашенных объектов
- Хранить геометрические данные в массивах вершин для уменьшения числа вызовов функций
- Сохранять и восстанавливать несколько параметров состояния одновременно

Даже самые сложные и интересные изображения, нарисованные в OpenGL, состоят из небольшого числа простых графических элементов. Это не является неожиданностью — посмотрите, какие совершенные картины создал Леонардо да Винчи только с помощью карандашей и кисти.

На самом высоком уровне абстракции существует только три основных операции рисования: очистка окна, рисование геометрических объектов и рисование растровых объектов. Растровые объекты: двухмерные изображения, битовые образы и символы шрифтов — рассматриваются в главе 8. В этой главе вы узнаете, как очистить экран и нарисовать геометрические объекты: точки, прямые линии и плоские многоугольники.

Вы могли подумать: «Подождите минуту. Я видел много компьютерной графики в фильмах и по телевизору, и всегда это были великолепно оттененные кривые линии и поверхности. Как такое получается, если OpenGL способен рисовать только прямые линии и плоские многоугольники?» На цветном рисунке 6 есть круглый стол, а объекты на столе состоят из кривых поверхностей. Все кривые линии и поверхности, которые вы видите, получены с помощью большого числа небольших плоских многоугольников и прямых линий, почти так же, как поверхность глобуса можно представить совокупностью огромного числа прямоугольных блоков. Она не кажется гладкой тогда, когда блоки относительно велики в сравнении с глобусом, но с увеличением числа блоков ее качество улучшается. Позже в этой главе мы покажем, как образовывать кривые линии и поверхности из большого набора маленьких графических примитивов.

Эта глава состоит из следующих основных разделов:

- **Основы рисования.** Показывает, как очистить окно и принудительно ускорить рисование. Также дает основную информацию об управлении цветами геометрических объектов и описание системы координат.
- **Описание точек, линий и многоугольников.** Рассматривает задание графических примитивов и методы их рисования.
- **Управление основными состояниями.** Показывает, как включать и выключать некоторые состояния (режимы) и опрашивать параметры состояний.
- **Отображение точек, линий и многоугольников.** Рассматривает методы управления примитивами, например определение размера точки, типа линии, закраски многоугольника.
- **Векторы нормали.** Обсуждает методику задания векторов нормалей для геометрических объектов и кратко описывает, для чего нормали нужны.
- **Массивы вершин.** Показывает, как размещать большой набор геометрических данных всего лишь в нескольких массивах и как с помощью всего лишь нескольких вызовов функций формировать изображение описываемого объекта. Сокращение числа вызовов функций может увеличить производительность отрисовки.
- **Группы атрибутов.** Показывает, как запрашивать текущие значения параметров состояний и как сохранять и восстанавливать одновременно несколько связанных параметров состояний.

- **Несколько советов по построению полигональных моделей поверхностей.** Раскрывает технические тонкости, связанные с кусочно-линейной аппроксимацией поверхностей.

Читая эту главу дальше, вы должны всегда помнить, что в OpenGL, если не указан иной порядок действий, при вызове команды рисования рисунок изменяется всегда. Это может показаться очевидным, однако в некоторых системах сначала оформляется список объектов и действий для рисования. Когда данный список подготовлен, аппаратуре дается команда на рисование элементов списка. Первый вариант называется *непосредственным графическим режимом* (*графическим режимом с прямым доступом*) и принят по умолчанию в OpenGL. Кроме режима с прямым доступом, вы можете сохранить некоторые команды в список (называемый *списком отображения*) для последующего рисования. Режим прямого доступа обычно легче для разработки программ, но списки отображения в большинстве более эффективны. Возможно, вам захочется их использовать, тогда читайте внимательно главу 7.

В OpenGL версии 1.1 были введены массивы вершин.

В OpenGL версии 1.2 было добавлено масштабирование нормалей к поверхностям (`GL_RESCALE_NORMAL`). Кроме того, для функции `glDrawRangeElements()` была добавлена поддержка массивов вершин.

В OpenGL версии 1.3 в ядре появилась базовая поддержка координат текстур для набора текстур. До этого мультитекстурирование было дополнительным расширением OpenGL.

В OpenGL версии 1.4 стало возможным хранение координат тумана и дополнительных цветов в массиве вершин, а также рисование примитивов по массиву вершин функциями `glMultiDrawArrays()` и `glMultiDrawElements()`.

Основы рисования

В этом разделе рассматриваются методы очистки окна при подготовке к рисованию, установки цвета объектов, которые необходимо нарисовать, и принудительного ускорения рисования. Напрямую ни один из этих методов не изменяет геометрические объекты, но любая программа рисования геометрических объектов связана с ними.

Очистка окна

Рисование на экране компьютера отличается от рисования на бумаге тем, что бумага изначально белая и мы можем сразу ее использовать для дела. На компьютере память, хранящая изображение, обычно заполнена последним нарисованным изображением, поэтому, как правило, нужно перед началом рисования очистить ее фоновым цветом. Фоновый цвет определяется программно. Для текстового процессора это, скорее всего, будет белый цвет (цвет бумаги), на котором уже будет отображен текст. Если рисуется вид из космического корабля, перед

рисованием звезд, планет и чужих кораблей экран нужно очистить черным цветом. Иногда очистка совсем не нужна, например, если это изображение внутренней части комнаты и все графическое окно занимают стены.

Если рассматривать вопрос только с этой точки зрения, вы можете спросить, почему мы говорим об очистке экрана вместо того, чтобы просто не нарисовать прямоугольник требуемого цвета размером с окно? Во-первых, специальная команда очистки окна может быть более эффективна, чем обычная команда рисования. Кроме того, как вы увидите из главы 3, OpenGL позволяет устанавливать систему координат, позицию наблюдателя и направление обзора, соответственно, бывает трудно определить необходимые размер и положение прямоугольника для очистки экрана. Наконец, на многих компьютерах графическое оборудование включает в себя несколько буферов, кроме буфера, содержащего информацию об отображаемых пикселах. Эти буферы также время от времени нуждаются в очистке, и удобно иметь команду для одновременной очистки различных их сочетаний. (См. главу 10 для обсуждения всех возможных буферов.)

Также необходимо знать, как цвета пикселов хранятся в графическом оборудовании в виде так называемых *битовых плоскостей*. Есть два метода хранения. Первый — каждое значение красной, зеленой, синей и альфа- (RGBA) составляющих пиксела может напрямую храниться в битовой плоскости, или второй — в битовой плоскости содержится единое значение индекса, связанное с цветом в цветовой таблице. Цветовой режим RGBA используется чаще, поэтому он фигурирует в большинстве примеров этой книги. (См. главу 4 для получения более подробной информации о цветовых режимах.) До 6-й главы на все значения альфа-составляющей можно не обращать внимания.

Например, данные строки очищают экран и заливают его черным цветом в режиме RGBA:

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

Первая строка устанавливает черный цвет очистки, а следующая команда очищает окно текущим цветом очистки. Аргумент функции `glClear()` определяет надлежащий буфер. В данном случае программа очистит только цветовой буфер, в котором хранится изображение, выводимое на экран. Обычно цвет очистки устанавливается один раз, в начале приложения, а затем выполняются очистки так часто, как это необходимо. OpenGL хранит текущий цвет очистки как параметр состояния и не требует его у вас при каждой очистке буфера.

Главы 4 и 10 рассматривают, как использовать остальные буферы. Сейчас вам достаточно знать, что очистить их довольно просто. Например, для очистки цветового буфера и буфера глубины используется следующая последовательность команд:

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

В этом примере использование `glClearColor()` аналогично предыдущему, функция `glClearDepth()` определяет значение, устанавливаемое для каждого

пиксела буфера глубины, а функция `glClear()` принимает в аргументе битовую операцию ИЛИ (OR), объединяющую очищаемые буферы. Следующее краткое описание функции `glClear()` включает таблицу, в которой приведены буферы, которые можно очистить, их названия и главы, где эти буферы описываются.

void `glClearColor`(GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);
Устанавливает текущий цвет очистки, используемый для очистки буфера в режиме RGBA. (См. главу 4 для получения подробной информации о режиме RGBA.) Значения красного (*red*), зеленого (*green*), синего (*blue*) и альфа-компонентов (*alpha*) устанавливаются при необходимости в диапазоне [0, 1]. По умолчанию действует белый цвет (0, 0, 0, 0).

void `glClear`(Glbitfield *mask*);
Очищает заданный буфер текущим значением. Аргумент *mask* определяет с помощью логического битового ИЛИ очищаемые буферы, перечисленные в табл. 2.1.

Таблица 2.1. Буферы для очистки

Буфер	Имя	Глава
Буфер цвета	GL_COLOR_BUFFER_BIT	Глава 4
Буфер глубины	GL_DEPTH_BUFFER_BIT	Глава 10
Буфер накопления	GL_ACCUM_BUFFER_BIT	Глава 10
Буфер трафарета	GL_STENCIL_BUFFER_BIT	Глава 10

Перед использованием команды очистки нескольких буферов необходимо определить значения цвета закраски после очистки для каждого очищаемого буфера, если не подходят значения, определенные по умолчанию для цвета RGBA, буферов глубины, накопления и трафарета. Кроме команд `glClearColor()` и `glClearDepth()`, устанавливающих значения для очистки для буферов цвета и глубины, используются команды `glClearIndex()`, `glClearAccum()` и `glClearStencil()`, определяющие значения очистки — индексы для буферов цвета, накопления и трафарета. (См. главы 4 и 10 для получения информации об этих буферах и их использовании.)

OpenGL позволяет определить несколько буферов, так как обычно очистка — довольно медленная операция, поскольку обрабатывается каждый пиксел окна (их могут быть миллионы), а некоторые графические контроллеры позволяют очищать несколько буферов одновременно. Оборудование, не поддерживающее одновременную очистку буферов, выполняет ее последовательно. Различие между

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

и

```
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
```

в том, что при одинаковом результате первый пример будет работать быстрее на большинстве компьютеров. Естественно, первый пример никогда не будет работать медленнее второго.

Установка цвета

В OpenGL описание очертания рисуемого объекта независимо от описания его цвета. Всегда часть геометрического объекта рисуется с использованием текущей цветовой схемы. Цветовая схема может быть простой, вида «рисуем все ярко-красным цветом», или сложной, например «нарисовать объект вида синего пластика, освещение задается желтыми точечными источниками света, плюс общее неяркое красновато-коричневое освещение где-то еще». Обычно программисты, использующие OpenGL, сначала устанавливают цвет или цветовую схему, а затем рисуют объекты. До тех пор пока цвет или цветовая схема не будут изменены, все объекты отображаются этим цветом или в этой цветовой схеме. Данный метод позволяет OpenGL добиваться более высокопроизводительного рисования, чем при явном задании цвета каждый раз.

Например, псевдокод

```
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
draw_object(C);
```

рисует объекты A и B красным цветом, а объект C — синим. Команда в четвертой строке, устанавливающая текущий цвет зеленым, — лишняя.

Окрашивание, освещение и затенение — отдельные темы для целой главы или большого раздела. Для рисования видимых графических примитивов достаточно только сведений об установке текущего цвета. Эта информация содержится в нескольких следующих абзацах. (См. главы 4 и 5 для получения дополнительной информации.)

Для установки цвета используется команда `glColor3f()`. У нее три аргумента, все вещественного типа с плавающей запятой, принимающие значения от 0,0 до 1,0. Эти параметры по порядку задают интенсивность красного, зеленого и синего компонентов цвета. Их можно считать спецификацией смешивания цветов: 0,0 означает полное отсутствие данного компонента, а 1,0 — максимальную интенсивность данного цвета. Таким образом, код

```
glColor3f(1.0, 0.0, 0.0);
```

задает ярчайший красный цвет, который может отобразить система, без примесей зеленого или синего. Все нули определяют черный цвет, а все единицы — белый. Установка всех трех составляющих в 0,5 даст серый тон (средний между белым и черным). Ниже приведены восемь команд с описанием цвета, который они задают:

```
glColor3f(0.0, 0.0, 0.0);       черный
glColor3f(1.0, 0.0, 0.0);       красный
glColor3f(0.0, 1.0, 0.0);       зеленый
glColor3f(1.0, 1.0, 0.0);       желтый
glColor3f(0.0, 0.0, 1.0);       синий
glColor3f(1.0, 0.0, 1.0);       маджента
glColor3f(0.0, 1.0, 1.0);       циан
glColor3f(1.0, 1.0, 1.0);       белый
```

Видно, что первые три аргумента команды для задания цвета очистки `glClearColor()` совпадают с аргументами функции `glColor3f()`. Четвертый — это значение альфа-составляющей, описываемой в разделе «Смешивание» главы 6. Будем полагать, что значение четвертого аргумента функции `glClearColor()` равно 0,0, оно является значением по умолчанию.

Ускорение завершения рисования

Как было показано в разделе «Конвейер рисования OpenGL» главы 1, современные графические системы можно представить в виде сборочного конвейера. Центральный процессор (CPU) обрабатывает команды рисования. Возможно, другое оборудование выполняет геометрические преобразования. Выполняется отсечение, затем затенение и/или текстурирование. Наконец, значения записываются в битовые плоскости для отображения. В высокопроизводительных архитектурах за каждую из этих операций отвечает отдельный блок, специально оптимизированный для ее максимально быстрого выполнения. В такой архитектуре CPU не ждет окончания выполнения команды рисования для того, чтобы выполнить следующую. Пока CPU отправляет вершины дальше по конвейеру, оборудование для преобразований выполняет преобразования предыдущего набора, выполняется отсечение набора, введенного перед предыдущим, и т. д. В системах, где CPU ждет результатов обработки команды, задержки перед обработкой следующей команды значительно снижают общую скорость конвейера.

Кроме того, приложения могут выполняться больше чем на одной машине. Например, предположим, что основная программа выполняется где-то в другом месте (на компьютере, называемом клиентом), а результат рисования отображается на вашей рабочей станции или терминале (сервере), который соединен с клиентом через сеть. В этом случае было бы чрезвычайно неэффективно посылать каждую команду по сети по очереди, так как это вызвало бы значительные задержки. Обычно клиент перед посылкой команд группирует их в единый пакет. К сожалению, код сетевого клиента, как правило, не позволяет узнать, завершила ли графическая программа рисование кадра или сцены. В худшем случае он всегда будет ждать дополнительной команды рисования для заполнения пакета, и вы никогда не увидите результат рисования.

Поэтому в OpenGL существует функция `glFlush()`, которая заставляет клиент отправить пакет, даже если тот не полон. Если сеть отсутствует и все команды выполняются на сервере, функция `glFlush()` может не оказывать никакого эффекта. Тем не менее, если вы пишете программу, которая должна работать и в сети и без сети, включите вызов `glFlush()` в конец каждого кадра или сцены. Запомните, что `glFlush()` не ждет окончания рисования — она влечет немедленную отрисовку и, таким образом, гарантирует *вызов* всех ранее отложенных команд за ограниченное время, даже если больше команд рисования не поступит.

Есть ряд других ситуаций, когда используется `glFlush()`.

- В случае программной визуализации, когда изображения строятся в системной памяти и вы не хотите постоянно обновлять экран.

- Реализации, которые собирают набор команд отрисовки для компенсации начальных затрат. Приведенный выше пример сетевой передачи — один из вариантов такого подхода.

void `glFlush`(void);

Инициирует выполнение предыдущего набора команд OpenGL, гарантируя их завершение за конечный промежуток времени.

Некоторые команды, например команды обмена буферами в режиме двойной буферизации, автоматически запускают ожидающие команды по сети перед выполнением.

Если `glFlush()` вам недостаточно, используйте `glFinish()`. Действие этой команды аналогично действию `glFlush()`, но, кроме того, она ожидает извещения от графического оборудования или сигнала по сети о том, что рисование в видеобуфере закончено. Функция `glFinish()` используется, если нужно синхронизировать задачи, например быть уверенным, что трехмерное изображение уже на экране, перед использованием вывода на экран текстовой информации поверх рисунка (Display PostScript). В качестве другого примера можно привести организацию пользовательского ввода после вывода изображения на экран. После выполнения команды `glFinish()` графический процессор блокируется до получения сообщения от графического оборудования о завершении рисования. Запомните, что злоупотребление `glFinish()` может привести к снижению производительности приложения, особенно при работе по сети, так как требует дополнительных подтверждений. Если для вас достаточно функции `glFlush()`, используйте ее, а не `glFinish()`.

void `glFinish`(void);

Инициирует выполнение всех предыдущих команд OpenGL. Команда не возвращает управление до тех пор, пока все запрограммированное в предыдущих командах не будет выполнено.

Управление системой координат

Всякий раз при начальном открытии окна или последующем его перемещении или изменении размера система управления окнами посылает соответствующие уведомления. При использовании GLUT процесс уведомлений автоматизирован: вызывается функция, зарегистрированная с помощью функции `glutReshapeFunc()`. Зарегистрированная функция обратной связи должна:

- восстанавливать прямоугольную область, служащую холстом для рисования;
- определять координатную систему для рисования объектов.

В главе 3 будет показано, как определить трехмерную систему координат, а сейчас мы определим простую, двухмерную систему координат, в которой можно рисовать несколько объектов. Вызовем `glutReshapeFunc(reshape)`, где `reshape()` — функция, приведенная ниже в листинге 2.1.

Листинг 2.1. Функция обработки изменения размеров окна

```
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrto2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
```

GLUT передает этой функции два аргумента: ширину и высоту в пикселах нового, перемещенного и с измененным размером окна. Функция `glViewport()` настраивает пиксельный прямоугольник для рисования в новом окне. Следующие три функции определяют систему координат для рисования с левым нижним углом в точке (0, 0) и правым верхним углом в точке (w, h) (рис. 2.1).

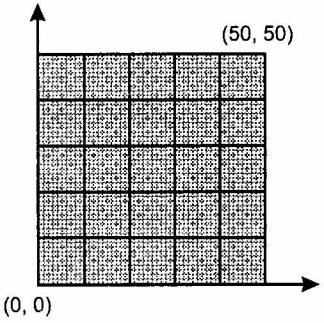


Рис. 2.1. Система координат с w = 50 и h = 50

Другими словами, представьте лист миллиметровой бумаги. Параметры w и h в процедуры `reshape()` определяют число квадратов по строкам и столбцам на этой бумаге. Затем поместим на этот лист оси координат. Функция `gluOrtho2D()` помещает начало координат (0, 0) в нижний, самый левый квадрат и назначает каждому квадрату единичный размер. Теперь при построении точек, линий и многоугольников они появятся на этой бумаге в легко предсказуемых для нас местах. (Считаем все объекты двумерными.)

Описание точек, линий и многоугольников

В этом разделе описывается создание графических примитивов в OpenGL. Все графические примитивы в конечном счете задаются с помощью вершин (*vertrces*) — координат, определяющих точки, концы сегментов линий и углы многоугольников. Следующий раздел рассматривает отображение этих примитивов и методы управления ими.

Что такое точки, линии и многоугольники?

Возможно, вы знаете математическое определение терминов *точка*, *линия* и *многоугольник*. Данные понятия в OpenGL означают похожее, но не совсем то же.

Первое отличие вытекает из ограничений компьютерных вычислений. В любой реализации OpenGL все вычисления с плавающей запятой происходят с конеч-

ной точностью и имеют погрешности округления. Следовательно, координаты точек, линий и многоугольников в OpenGL заведомо неточны.

Более существенное различие происходит из-за ограничений дисплея растрового типа. В дисплеях минимальным отображаемым элементом является пиксел, и несмотря на то, что он может быть размером менее 1/100 дюйма, он все равно больше математического определения точки и толще математического определения линии. Когда OpenGL выполняет вычисления, предполагается, что точка представлена вектором чисел с плавающей запятой (вещественными числами). Однако обычно (но не всегда) точка рисуется в виде одиночного пиксела, и множество различных точек с немного различающимися координатами может быть нарисовано OpenGL в одном и том же пикселе.

Точки

Точка представляется набором вещественных чисел, называемых *вершиной*. Все внутренние вычисления выполняются для трехмерных вершин. Определяемые пользователем двухмерные вершины (когда заданы только координаты x и y) также являются трехмерными, но с координатой z, равной 0.

ПРИМЕЧАНИЕ

OpenGL работает в однородной системе координат трехмерной проекции, поэтому для внутренних вычислений все вершины представляются четырьмя координатами вещественного типа (x, y, z, w). Если w не равно 0, эти координаты приводятся к Евклидовым, трехмерным (x/w, y/w, z/w). Можно задавать координату w в командах OpenGL, но это бывает нужно очень редко. Если координата w не задана, она принимается равной 1.0 (См. приложение E для получения более подробной информации об однородной системе координат.)

Линии

В OpenGL термин *линия* связан с отрезком, а не с математическим термином, определяющим бесконечную линию. Существует простой способ определения серии соединенных отрезков или замкнутой последовательности отрезков (рис. 2.2). Во всех случаях линии состоят из последовательности соединенных отрезков, определяемых вершинами их концов.

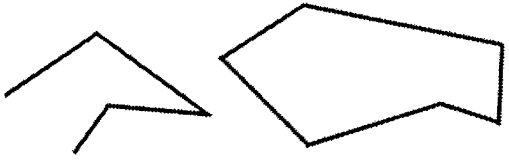


Рис. 2.2. Два вида связанных линий

Многоугольники

Многоугольник (полигон) — площадь, ограниченная одиночным замкнутым контуром, состоящим из отрезков, определяемых вершинами их концов. Обычно многоугольники рисуются с закрашенными внутри пикселами, но можно также их рисовать контуром или набором точек (см. раздел «Подробно о многоугольниках»).

Обычно многоугольники бывают сложными, поэтому OpenGL накладывает жесткие ограничения на задание простых многоугольников. Во-первых, границы многоугольников в OpenGL не могут пересекаться (в математике такие многоугольники называются *простыми*). Во-вторых, многоугольники в OpenGL должны быть *выпуклыми*, то есть они не должны иметь «углублений». Если говорить строже, область называется выпуклой, если для любых двух точек, находящихся внутри ее, отрезок, соединяющий их, также находится внутри. На рис. 2.3 представлены примеры «правильных» и «неправильных» многоугольников. Однако OpenGL не ограничивает число отрезков контура. Запомните, что нельзя определить многоугольники с отверстиями. Они невыпуклые, и их граница не может быть нарисована замкнутой последовательностью отрезков. Если попытаться задать невыпуклый залитый многоугольник, ожидаемого результата не получится. Это типичная ситуация — в большинстве систем, например, ничто, кроме *выпуклой оболочки*, не может быть закрашено.

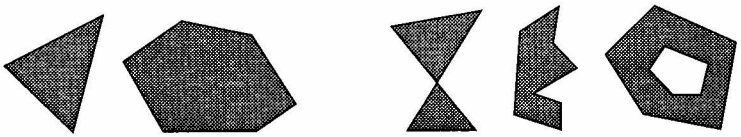


Рис. 2.3. «Правильные» и «неправильные» многоугольники

Причина, по которой OpenGL вводит ограничения на правильность многоугольников, — простота создания оборудования для быстрой заливки многоугольников данного вида. Простые многоугольники могут быть закрашены со скоростью, соответствующей их простоте. А вот что трудно и не быстро — это определить (для увеличения производительности), является ли многоугольник простым.

Многие поверхности в реальном мире содержат непростые многоугольники, невыпуклые многоугольники или многоугольники с отверстиями. Так как все такие многоугольники могут быть созданы путем объединения простых выпуклых многоугольников, библиотека GLU предоставляет возможности для построения более сложных объектов. Эти команды получают комплексное описание и разбивают объекты на группы простых многоугольников, которые могут быть нарисованы OpenGL. (См. раздел «Мозаичное представление многоугольников» в главе 11 для получения более подробной информации о командах разбивки.)

Так как вершины в OpenGL всегда трехмерны, точки, задающие границы одиночных многоугольников, не нужно непременно располагать в одной пространственной плоскости. (Конечно, это происходит во многих случаях — например, если все координаты *z* нулевые или если многоугольник — *треугольник*.) Если вершины многоугольника не лежат в одной плоскости, после различных пространственных вращений, изменений точек обзора и проецирования на экран точки могут перестать образовывать простой многоугольник. Например, изобразим *четырехугольник*, вершины которого немного выходят за пределы плоскости и выглядят так, как будто они принадлежат одной плоскости. Из него можно получить непростой многоугольник, похожий на бабочку (рис. 2.4.), который не

гарантирует корректное отображение. Эта ситуация отражает не все проблемы, которые могут возникнуть, если вы приближаете кривую поверхность к прямоугольнику, точки которого лежат в одной плоскости. Можно решить данную проблему, если использовать треугольники, так как любые три точки всегда лежат в одной плоскости.

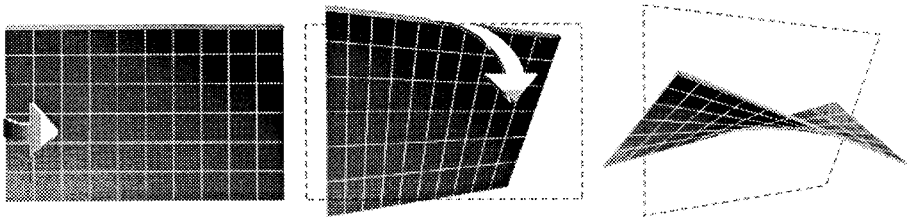


Рис. 2.4. Преобразование пространственного многоугольника в поверхность (сложный многоугольник)

Прямоугольники

Так как прямоугольники часто используются в графических приложениях, OpenGL предоставляет примитив для рисования закрашенного прямоугольника, `glRect*()`. Можно рисовать прямоугольник как многоугольник, как это описано в разделе «Рисование графических примитивов в OpenGL», но функция `glRect*()` в вашей реализации OpenGL, скорее всего, оптимизирована для рисования прямоугольников.

void `glRect`{sifd}(*TYPE x1, TYPE y1, TYPE x2, TYPE y2*);
void `glRect`{sifd}`v`(*TYPE *v1, TYPE *v2*);
Рисует прямоугольник, задаваемый координатами углов (*x1*, *y1*) и (*x2*, *y2*). Прямоугольник располагается в плоскости $z = 0$ и имеет стороны, параллельные осям x и y. В векторной форме функции углов задаются двумя указателями на массивы, каждый из которых содержит пару (x, y).

Запомните, что прямоугольник располагается в определенном месте трехмерного пространства (в плоскости xy и параллельно осям), но это можно изменить путем вращения и других преобразований. (См. главу 3 для получения более подробной информации.)

Кривые линии и кривые поверхности

Любую гладкую кривую линию или поверхность можно представить — с любой степенью точности — в виде набора коротких отрезков или маленьких многоугольников. Поэтому разбиение кривых и поверхностей на достаточное число частей с последующим представлением их отрезками или плоскими многоугольниками позволяет создавать кривые (рис. 2.5). Если у вас есть сомнения по поводу этого метода, представьте разбиение, выполняемое до тех пор, пока каждый отрезок или многоугольник не станет меньше пиксела.

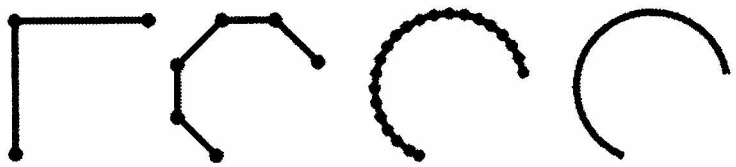


Рис. 2.5. Аппроксимирующие кривые

Так как кривые не являются графическими примитивами, OpenGL напрямую поддерживает их разбиение и рисование. (См. главу 12 для получения более подробной информации о рисовании кривых и поверхностей.)

Задание вершин

В OpenGL любой геометрический объект в конечном счете описывается набором вершин. Для их определения предназначена функция `glVertex*()`.

void `glVertex`{234}{sifd}[v](*TYPE coords*);
Определяет вершины, используемые для описания геометрических объектов. Соответствующие версии функции поддерживают от двух (x, y) до четырех координат (x, y, z, w) на одну вершину. Если применяется версия, не использующая координату z или w, подразумевается, что z равна 0, а w равна 1. Вызов функции `glVertex*()` имеет смысл только между операторными скобками `glBegin()` и `glEnd()`.

Пример использования `glVertex*()` приведен в листинге 2.2.

Листинг 2.2. Правильное использование glVertex*()

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);
GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

Первый пример определяет вершину с трехмерными координатами (2, 3, 0). (Помните, что если координата z не задана, то она равна 0.) Координаты во втором примере — (0.0, 0.0, 3.1415926535898) (вещественное число с двойной точностью). Третий пример задает вершину тремя координатами (1.15, 0.5, –1.1) в однородной системе координат. (Помните, что значения координат x, y и z делятся на значение координаты w.) И наконец, в последнем примере, *dvect* — указатель на массив трех вещественных чисел с двойной точностью.

На некоторых компьютерах векторная форма `glVertex*()` более эффективна, так как графической подсистеме нужно обработать только один параметр. Специальное аппаратное обеспечение позволяет отправлять весь набор координат за один раз. Если ваш компьютер способен на такое, в ваших интересах скомпоновать данные, то есть упорядочить координаты вершин в памяти. В этом случае возможен выигрыш в производительности. (См. раздел «Массивы вершин».)

Рисование графических примитивов в OpenGL

Теперь, когда вы умеете определять вершины, вам нужно знать, как приказать OpenGL интерпретировать этот набор в виде точек, линий или многоугольника. Для этого каждый набор заключается в операторные скобки — команды `glBegin()` и `glEnd()`. Аргумент функции `glBegin()` определяет тип описываемого вершинами графического примитива. Например, в листинге 2.3 определяется набор вершин, описывающий многоугольник, показанный на рис. 2.6.

Листинг 2.3. Залитый многоугольник

```
glBegin(GL_POLYGON);
    glVertex2f (0.0, 0.0);
    glVertex2f (0.0, 3.0);
    glVertex2f (4.0, 3.0);
    glVertex2f (6.0, 1.5);
    glVertex2f (4.0, 0.0);
glEnd();
```

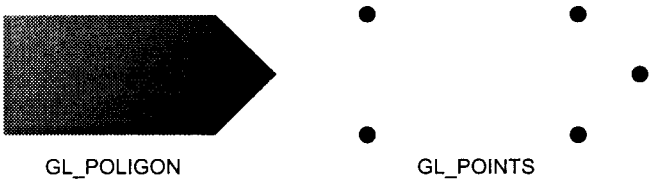


Рис. 2.6. Рисование многоугольника или набора точек

Если вместо `GL_POLYGON` использовать `GL_POINT`, будут определены пять точек, показанных на рис. 2.6. В табл. 2.2. приведены десять возможных значений аргумента функции `glBegin()` и соответствующие им типы примитивов.

void `glBegin`(Glenum *mode*);

Определяет начало списка вершин, описывающих графический примитив. Тип примитива указывается значением параметра *mode*, одним из перечисленных в табл. 2.2.

Таблица 2.2. Значения параметра mode и их описания

Значение	Описание
GL_POINTS	Отдельные точки
GL_LINES	Каждая пара вершин задает отрезок
GL_LINE_STRIP	Серия соединенных отрезков
GL_LINE_LOOP	Аналогично предыдущему, причем последняя точка соединяется с первой
GL_TRIANGLES	Тройка вершин задает треугольник
GL_TRIANGLE_STRIP	Рисуются треугольники с общей стороной
GL_TRIANGLE_FAN	Рисуются треугольники с общей стороной. Другой способ соединения сторон
GL_QUADS	Четыре вершины задают четырехугольник
GL_QUAD_STRIP	Четырехугольники с общей стороной
GL_POLYGON	Рисуется простой правильный многоугольник

void `glEnd`(void);
Определяет конец списка вершин.

На рис. 2.7 показаны примеры всех графических примитивов, перечисленных в табл. 2.2, с описанием пикселов, рисуемых для каждого объекта. Заметьте, что определены дополнительные примитивы для точек, типов линий и многоугольников. Таким образом, существует множество путей рисования одинаковых примитивов. Выбранный метод зависит от набора вершин.

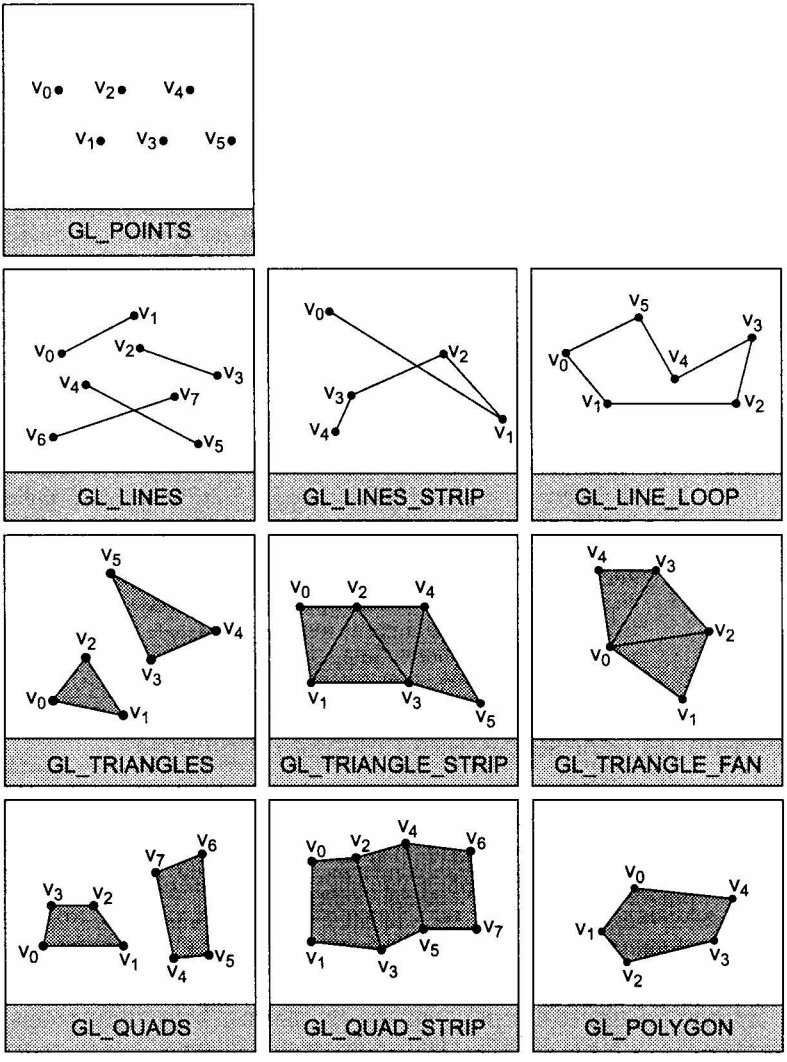


Рис. 2.7. Типы графических примитивов

Пусть между операторными скобками `glBegin()` и `glEnd()` описано n вершин (v_0, v_1, v_2, ..., v_{n-1}). Тогда:

GL_POINTS	Рисует точки в каждой n-й вершине
GL_LINES	Рисуется серия несвязанных отрезков. Отрезки соединяют вершины v_0 и v_1, v_2 и v_3 и т. д. Если n — нечетное, последний отрезок рисуется между вершинами v_{n-3} и v_{n-2}, а вершина v_{n-1} игнорируется
GL_LINE_STRIP	Рисует отрезки между вершинами v_0 и v_1, затем между v_1 и v_2, и т. д., последний отрезок — между вершинами v_{n-2} и v_{n-1}. Таким образом, рисуется $n-1$ отрезков. Нет никаких ограничений на описание вершин. Линии могут произвольно пересекаться
GL_LINE_LOOP	Аналогично GL_LINE_STRIP, за исключением того, что последний отрезок рисуется между вершинами v_{n-1} и v_0 и завершает цикл
GL_TRIANGLES	Рисует серию треугольников, используя вершины v_0, v_1, v_2, затем v_3, v_4, v_5, и т. д. Если n не кратно 3, последние одна или две вершины игнорируются
GL_TRIANGLE_STRIP	Рисует серию треугольников, используя вершины v_0, v_1, v_2, затем v_2, v_1, v_3, (запомните порядок), затем v_2, v_3, v_4 и т. д. Порядок гарантирует, что все треугольники рисуются в одном направлении, поэтому полученная лента может корректно формировать часть поверхности. Сохранение направления важно для таких операций, как отбраковка (см. раздел «Обращение и отбраковка поверхностей многоугольников»). Значение n должно быть больше или равно 3 для того, чтобы что-то нарисовалось
GL_TRIANGLE_FAN	Аналогично GL_TRIANGLE_STRIP, но порядок вершин v_0, v_1, v_2, затем v_0, v_2, v_3, затем v_0, v_3, v_4, и т. д. (см. рис. 2.7)
GL_QUADS	Рисует серию четырехугольников, используя вершины v_0, v_1, v_2, v_3, затем v_4, v_5, v_6, v_7, и т. д. Если n не кратно 4, последние одна, две или три вершины игнорируются
GL_QUAD_STRIP	Рисует серию четырехугольников, начиная с v_0, v_1, v_3, v_2, затем v_2, v_3, v_5, v_4, затем v_4, v_5, v_7, v_6, и т. д. (см. рис. 2.7.) Значение n должно быть больше или равно 4, чтобы что-то нарисовалось. Если n — нечетное, последняя вершина отбрасывается
GL_POLYGON	Рисует многоугольник, используя точки v_0,..., v_{n-1} как вершины. Значение n должно быть больше или равно 3, иначе ничего не нарисуется. Кроме того, грани многоугольника не должны пересекаться и многоугольник должен быть выпуклым. Если вершины не удовлетворяют этим условиям, результат непредсказуем

Ограничения на использование glBegin() и glEnd()

Наиболее важная информация о вершинах — их координаты, указываемые командой `glVertex*()`. Но кроме этой команды существует ряд других, позволяющих определять дополнительную информацию для каждой вершины — цвет, вектор нормали, координаты текстуры или любую их комбинацию — с помощью специальных команд. Кроме того, есть несколько других команд, которые разрешено помещать между операторными скобками `glBegin()` и `glEnd()`. В табл. 2.3 приведен полный список таких команд.

Никакие другие команды OpenGL не могут находиться между операторными скобками `glBegin()` и `glEnd()`, и вызов большинства из них приведет к возбуждению сообщения об ошибке. Использование некоторых команд массивов вершин,

таких, как `glEnableClientState()` и `glVertexPointer()`, между `glBegin()` и `glEnd()` приводит к непредсказуемым результатам, но ошибок не генерирует. (Также непредсказуемыми результатами опасны команды `glX*()`.) Подобных ситуаций следует избегать, так как локализация ошибок довольно трудна.

Таблица 2.3. Команды, допускаемые между операторными скобками glBegin() и glEnd()

Команда	Действие	Глава
glVertex*()	Устанавливает координаты вершин	Глава 2
glColor*()	Устанавливает RGBA-цвет	Глава 4
glIndex*()	Устанавливает индекс цвета	Глава 4
glSecondaryColor*()	Устанавливает вторичный цвет для приложений с посттекстурированием	Глава 9
glNormal*()	Задает координаты вектора нормали	Глава 2
glMaterial*()	Задает свойства материала	Глава 5
glFogCoord*()	Устанавливает координаты тумана	Глава 6
glTexCoord*()	Устанавливает координаты текстур	Глава 9
glMultiTexCoord*()	Устанавливает координаты текстур для мультитекстурирования	Глава 9
glEdgeFlag*()	Управляет рисованием границ	Глава 2
glArrayElement()	Извлекает массив вершин	Глава 2
glEvalCooord*(), glEvalPoint*()	Генерирует координаты	Глава 12
glCallList(), glCallLists()	Выполняет список отображения	Глава 7

Однако это касается только команд OpenGL. Между `glBegin()` и `glEnd()`, конечно, не запрещается вставлять любые конструкции языка программирования (за исключением вышеупомянутых `glX*()`). Например, листинг 2.4 рисует окружность.

Листинг 2.4. Языковые конструкции между glBegin() и glEnd()

```
#define PI 3.1415926535898
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
for (i = 0;  i < circle_points;  i++)   {
    angle = 2*PI*i/circle_points;
    glVertex2f(cos(angle), sin(angle));
}
glEnd();
```

ПРИМЕЧАНИЕ

Метод рисования круга, приведенный в листинге 2.4, не самый эффективный, особенно если контур нужно рисовать часто. Конечно, графические команды довольно быстры, но в данном коде для каждой вершины вычисляются углы и обсчитываются sin() и cos(). Кроме того, добавляются затраты на выполнение цикла. Другим способом вычисления вершин окружности является использование команд GLU. (См. раздел «Поверхности второго порядка: сферы, цилиндры и диски» в главе 11). Если нужно нарисовать несколько окружностей, вычислите координаты вершин один раз и сохраните их в списке отображения (см. главу 7) или используйте для рисования массив вершин.

Все команды `glVertex*()` должны находиться между операторными скобками `glBegin()` и `glEnd()`, если только они не образуют список отображения. (Их использование в любом другом месте не даст никакого эффекта.) Если они образу-

ют список отображения, то исполняются лишь между `glBegin()` и `glEnd()`. (См. главу 7 для получения информации о списках отображения.)

Заметьте, вершины генерируются только при использовании команд `glVertex*()`. Во время вызова `glVertex*()` OpenGL связывает получившуюся вершину с информацией о текущем цвете, координатах текстуры, векторе нормали и т. д. Посмотрите на код, приведенный ниже. Первая точка рисуется красным, вторая и третья — синим, несмотря на другие команды, определяющие цвет:

```
glBegin(GL_POINTS);
    glColor3f(0.0, 1.0, 0.0);     /* зеленый */
    glColor3f(1.0, 0.0, 0.0);     /* красный*/
    glVertex(...);
    glColor3f(1.0, 1.0, 0.0);     /* желтый */
    glColor3f(0.0, 0.0, 1.0);     /* синий */
    glVertex(...);
    glVertex(...);
glEnd();
```

Вы вправе применить любую комбинацию всех 24 вариантов команды `glVertex*()` между `glBegin()` и `glEnd()`, хотя в реальных приложениях для всех вызовов достаточно одного вида. Если ваши данные о вершинах последовательные и повторяющиеся (например, `glColor*()`, `glVertex*()`, `glColor*()`, `glVertex*()`, ...), можно увеличить производительность программы за счет массивов вершин. (См. раздел «Массивы вершин».)

Управление основными состояниями

В предыдущем разделе вы видели пример параметра состояния — текущий RGBA-цвет и то, как он связан с примитивом. OpenGL поддерживает много состояний и параметров состояний. Объекты могут быть отображены с освещением, текстурированием, удалением скрытых поверхностей, туманом и другими состояниями, влияющими на их вид.

По умолчанию большинство этих состояний отключено. Их включение может быть довольно затратным. Например, включение карты текстур почти наверняка замедлит процесс построения примитива. Однако качество изображения повысится, и рисунок будет выглядеть более реалистичным вследствие расширения графических возможностей.

Для переключения большинства состояний используются две простые команды:

void `glEnable`(GLenum *cap*);

void `glDisable`(GLenum *cap*);

`glEnable()` — включает возможность, а `glDisable()` — выключает. Более 60 значений могут быть параметрами `glEnable()` и `glDisable()`. Например, `GL_BLEND` (управляющее смешиванием RGBA-цветов), `GL_DEPTH_TEST` (управляющее сравнением глубины и обновляющее буфер глубины), `GL_FOG` (управляющее туманом), `GL_LINE_STIPPLE` (прерывистые линии) и `GL_LIGHTING` (управляющее освещением).

Также можно проверять текущее значение параметра состояния командой

GLboolean `glIsEnabled`(GLenum *capability*);
Возвращает значение `GL_TRUE` или `GL_FALSE`, в зависимости от того, включена или выключена интересующая возможность.

Состояния, которые вы уже видели, имеют всего два значения: включено и выключено. Однако большинство команд OpenGL устанавливает для параметров состояния более сложные значения. Например, команда `glColor3f()` устанавливает три значения, являющиеся частями состояния `GL_CURRENT_COLOR`. Существует пять команд для опроса таких состояний:

void `glGetBooleanv`(GLenum *pname*, GLboolean **params*);
void `glGetIntegerv`(GLenum *pname*, GLint **params*);
void `glGetFloatv`(GLenum *pname*, GLfloat **params*);
void `glGetDoublev`(GLenum *pname*, GLdouble **params*);
void `glGetPointerv`(GLenum *pname*, GLvoid ***params*);
Получает тип параметра состояния: логический (boolean), целый (integer), с плавающей запятой (floating-point), вещественный с двойной точностью (double-precision) или указатель (pointer). Параметр *pname* — символьная константа, определяющая параметр состояния, а *params* — указатель на массив указанного типа, в котором сохраняются составные параметры состояния. В таблицах в приложении Б перечислены значения параметра *pname*. Например, для получения текущего RGBA-цвета таблица в приложении Б указывает, что нужно использовать `glGetIntegerv(GL_CURRENT_COLOR, params)` или `glGetFloatv(GL_CURRENT_COLOR, params)`. При необходимости выполняется преобразование типов для того, чтобы вернуть значение требуемого типа.

Эти команды управляют большинством, но не всеми запросами информации состояний. (См. раздел «Команды запроса» в приложении Б для получения информации о дополнительных 16 командах запроса.)

Отображение точек, линий и многоугольников

По умолчанию точка рисуется как одиночный пиксел на экране, линия рисуется сплошной, толщиной в один пиксел, а многоугольники рисуются со сплошной внутренней заливкой. В следующих разделах показывается, как можно изменить эти режимы по умолчанию.

Подробно о точках

Для изменения размера отображаемой точки используется команда `glPointSize()`, аргументом которой является нужный размер в пикселах.

void `glPointSize`(GLfloat *size*);
Устанавливает ширину рисуемой точки (в пикселах). Переменная *size* должна быть больше 0.0 и по умолчанию принимается равной 1.0.

Реальное число пикселов на экране, которые подсвечиваются для точек различной ширины, зависит от того, включен или отключен режим устранения ступенчатости. (Устранение ступенчатости — технология сглаживания точек и линий во время рисования. См. разделы «Сгаживание» и «Параметры точки» главы 6). Если устранение ступенчатости отключено (по умолчанию), дробное значение ширины округляется до целого и на экране рисуется квадратная область пикселов. Таким образом, если ширина равна 1.0, квадрат получается размером 1 × 1 пиксел, если ширина равна 2.0 — 2 × 2 пиксела и т. д.

При включении режима сглаживания или множественной выборки рисуется группа пикселов в виде круга, и пикселы на границе обычно рисуются бледнее остальных, чтобы придать границе плавный вид. В этом режиме дробные значения ширины не округляются.

Большинство реализаций OpenGL поддерживают очень большой размер точки. Выяснить минимальный и максимальный размеры точки можно с помощью команды `glGetFloatv()` со значением `GL_ALIASED_POINT_SIZE_RANGE`. Точно так же можно узнать диапазон размеров точки в режиме устранения ступенчатости с помощью команды `cglGetFloatv()` со значением `GL_SMOOTH_POINT_SIZE_RANGE`. Поддерживаемый размер подправленной точки лежит между минимальным и максимальным значениями. Вызов функции `glGetFloat()` со значением `GL_SMOOTH_POINT_SIZE_GRANULARITY` позволяет узнать поддерживаемую точность отображения исправленной точки. Например, если выполняется команда `glPointSize (2.37)`, а точность равна 0.1, размер точки округлится до 2.4.

Подробно о линиях

В OpenGL можно определять линии с различной толщиной и прерывистые линии различных видов — пунктирные, штриховые, штрихпунктирные различных типов и т. д.

Толщина линии

void `glLineWidth`(GLfloat *width*);

Устанавливает толщину рисуемой линии (в пикселах). Параметр *width* должен быть больше 0.0 и по умолчанию равен 1.0.

Линии, рисуемые при включенном режиме устранения ступенчатости или множественной выборки, выглядят более ровными. (См. разделы «Сглаживание точек или линий» главы 6 и «Сглаживание геометрических примитивов с помощью мультивыборки» главы 6.) Без сглаживания толщина 1, 2 и 3 соответствует линиям толщиной в 1, 2 и 3 пиксела. При включенном режиме устранения ступенчатости возможно задание дробной толщины линии, а пикселы по границам рисуются более светлыми, чем сама линия. Так же, как и в случае с размером точки, некоторые реализации OpenGL могут ограничивать толщину исправленной линии максимальной толщиной сглаженной линии, округленной до ближайшего целого значения. Диапазон возможных значений толщины исправленных линий можно узнать с помощью команды `glGetFloatv()` со значением `GL_ALIASED_LINE_WIDTH_RANGE`.

Для определения максимальной и минимальной толщины неисправленной линии и поддерживаемой степени точности используется функция `glGetFloatv()` со значениями `GL_SMOOTH_LINE_WIDTH_RANGE` и `GL_SMOOTH_LINE_WIDTH_GRANULARITY`.

ПРИМЕЧАНИЕ

Запомните, что по умолчанию толщина линии — 1 пиксел, но она может казаться больше на экранах низкого разрешения. Для экрана компьютера это не проблема, но, если OpenGL используется для рисования на плоттерах высокого разрешения, линия в один пиксел может быть почти невидима. Для устранения зависимости толщины линии от разрешения необходимо учитывать физические размеры пикселов.

ПРИМЕЧАНИЕ

Для несглаженной линии ее толщина не измеряется по перпендикуляру к линии. Если абсолютный угол наклона меньше 1.0, измерение производится по оси y, в противном случае — по оси x. Рисование сглаженной линии практически эквивалентно рисованию закрашенного прямоугольника заданной ширины, отцентрированного точно по линии.

Прерывистые линии

Для создания прерывистых линий (пунктирных или штрихованных) используется команда `glLineStipple()`, определяющая тип штриховки, и `glEnabled()`, включающая соответствующий режим:

```
glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STIPPLE);
```

void `glLineStipple`(Glint *factor*, GLushort *pattern*);

Устанавливает текущий тип образца штриховки для линии. Аргумент *pattern* — 16-битная последовательность 0 и 1, которая повторяется необходимое для протяженности линии число раз. Единица означает, что текущий пиксел подсвечивается, а 0 — нет. Образец может удлиняться с помощью аргумента *factor*, который умножает каждую последовательность 0 и 1. То есть, если в образце присутствует последовательность из трех единиц, она удлиняется до шести, когда *factor* равен 2. Параметр *factor* принимает значения от 1 до 256. Режим прерывистых линий должен быть уже включен с помощью значения `GL_LINE_STIPPLE` функции `glEnable()`. Для его отключения используется функция `glDisable()` с тем же значением аргумента.

В приведенном выше примере образец имеет вид 0x3F07 (0011111100000111 в двоичном виде), то есть сначала рисуются 3 пиксела, затем следует промежуток в 5 пикселов, затем рисуются 6 пикселов и пропускаются 2 пиксела. Отсчет идет с младшего бита. При значении *factor* = 2: подсвечиваются 6 пикселов, затем промежуток в 10 пикселов, опять подсвечиваются 12 пикселов и снова промежуток в 4 пиксела. Рисунок 2.8 демонстрирует рисование линий с различным шаблоном и значением повтора. Отключенный режим рисования прерывистых линий равносилен образцу 0xFFFF без повторов — то есть сплошной линии. Помните, что прерывистые линии также могут быть определены с различной шириной.

Таким образом, алгоритм вывода прерывистых линий следующий: начинаем рисовать линию, и каждый раз при рисовании каждого пиксела образец сдвигается на 1 бит (или на *factor* пикселов, если *factor* не равен 1). Когда рисуется серия

соединенных отрезков между одиночными операторными скобками `glBegin(` и `glEnd()`, при переходе от одного отрезка к другому возврата к младшему би ту не происходит, и образец продолжает сдвигаться. При выполнении функци `glEnd()` образец сбрасывается в начало. Если рисуются линии с использование `GL_LINES`, образец сбрасывается для каждой независимой линии.

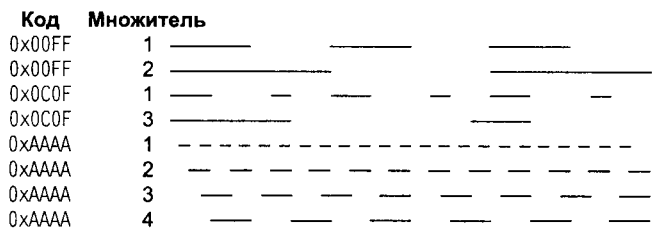


Рис. 2.8. Линии с различным фактором повтора

Программа из листинга 2.5 рисует линии с различными комбинациями образц и толщины линии. Также листинг демонстрирует, что произойдет, если лини рисуются как серия сегментов, а не единым блоком. Результат работы программ показан на рис. 2.9.

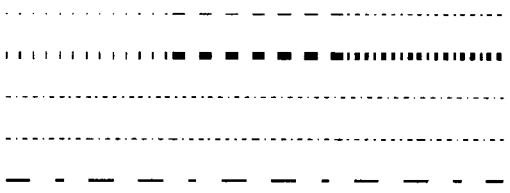


Рис. 2.9. Прерывистые линии

Листинг 2.5. Шаблоны прерывистых линий: lines.c

```
#define drawOneLine(x1,y1,x2,y2)    glBegin(GL_LINES);    \
    glVertex2f((x1),(y1)); glVertex2f((x2),(y2)); glEnd();

void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
}

void display(void)
{
    int i;
    glClear(GL_COLOR_BUFFER_BIT);
/* устанавливаем белый цвет для всех линий */
    glColor3f(1.0, 1.0, 1.0);
/* в первой строке 3 линии разного вида */
    glEnable(GL_LINE_STIPPLE);
    glLineStipple(1, 0x0101);  /* пунктирная */
    drawOneLine(50.0, 125.0, 150.0, 125.0);
    glLineStipple(1, 0x00FF);  /* штрихованная */
```

```
    drawOneLine(150.0, 125.0, 250.0, 125.0);
    glLineStipple(1, 0x1C47);  /* штрихпунктирная */
    drawOneLine(250.0, 125.0, 350.0, 125.0);

/* во второй строке 3 толстые линии разного вида */
    glLineWidth(5.0);
    glLineStipple(1, 0x0101);  /* пунктирная */
    drawOneLine(50.0, 100.0, 150.0, 100.0);
    glLineStipple(1, 0x00FF);  /* штрихованная */
    drawOneLine(150.0, 100.0, 250.0, 100.0);
    glLineStipple(1, 0x1C47);  /* штрихпунктирная */
    drawOneLine(250.0, 100.0, 350.0, 100.0);
    glLineWidth(1.0);

/* в третьей строке 6 штрихпунктирных линий */
/* как часть одиночной пунктирной линии     */
    glLineStipple(1, 0x1C47);  /* штрихпунктирная */
    glBegin(GL_LINE_STRIP);
    for (i = 0; i < 7; i++)
        glVertex2f(50.0 + ((GLfloat) i * 50.0), 75.0);
    glEnd();

/* в четвертой строке 6 одинаковых штрихпунктирных независимых линий */
    for (i = 0; i < 6; i++) {
        drawOneLine(50.0 + ((GLfloat) i * 50.0), 50.0,
                    50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
    }

/* В пятой строке одна штрихпунктирная линия, а число повторов равно 5 */
    glLineStipple(5, 0x1C47);  /* штрихпунктирная */
    drawOneLine(50.0, 25.0, 350.0, 25.0);
    glDisable(GL_LINE_STIPPLE);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(400, 150);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

Подробно о многоугольниках

Обычно многоугольники рисуются со сплошной закраской, но их можно рисовать в виде контура или в виде точек в вершинах. Закраска многоугольника может быть сплошной (заливка) или фактурной по некоторому заданному образцу. Если соседние многоугольники имеют общие границы или вершины, пикселы на границах или вершинах рисуются всего один раз — они принадлежат только одному многоугольнику — с той целью, чтобы у частично прозрачных многоугольников границы не рисовались дважды, что сделало бы их более темными (или светлыми, в зависимости от цвета рисования). Результатом этого может быть отсутствие заливки в одном или нескольких пиксельных столбцах или строках в узких многоугольниках.

Для сглаживания закрашенных многоугольников рекомендуется использовать режим мультивыборки. Для получения более детальной информации см. раздел «Сглаживание геометрических примитивов с помощью мультивыборки» главы 6.

Многоугольники в виде точек, границ или заливки

Многоугольники имеют две стороны — лицевую и оборотную, которые могут быть отрисованы различно, в зависимости от того, какая сторона сейчас видна. Это позволяет получать такой разрез сплошных объектов, где ясно различаются внутренние и внешние части. По умолчанию обе поверхности рисуются одинаково. Для изменения режима по умолчанию или рисования только контура или вершин используется функция `glPolygonMode()`.

void `glPolygonMode`(GLenum *face*, GLenum *mode*);

Управляет режимом рисования лицевой и оборотной сторон многоугольников. Параметр *face* может принимать значение `GL_FRONT_AND_BACK`, `GL_FRONT` или `GL_BACK`, а параметр *mode* — значение `GL_POINT`, `GL_LINE` или `GL_FILL`, определяющие, рисуются ли точки, контур или залитый многоугольник. По умолчанию обе стороны многоугольника рисуются залитыми.

Например, можно нарисовать многоугольник с залитой лицевой стороной и оборотной в виде контура с помощью следующих двух команд:

```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

Обращение и отбраковка поверхностей многоугольников

Условно многоугольники, вершины которых заданы против часовой стрелки, называются *лицевыми* (front-facing). Можно создавать поверхности с любой «разумной» заливкой — математики могут назвать такую поверхность ориентируемым многообразием (возможны сферы, кольца и чайники; невозможны бутылки Клейна и ленты Мебиуса) — из многоугольников с непротиворечивой ориентацией. Другими словами, можно использовать все многоугольники с направлением обхода по часовой стрелке или все многоугольники, заданные против часовой стрелки. (Это, по существу, математическое определение *ориентируемости*.)

Допустим, вы должным образом описали модель ориентированной поверхности, но вам нужна ориентация по часовой стрелке для невидимой стороны. Вы можете поменять оборотную сторону («вывернуть наизнанку») с помощью функции `glFrontFace()`, задающей желательную ориентацию лицевого многоугольника.

void `glFrontFace`(GLenum *mode*);

Управляет тем, как определяется лицевой многоугольник. По умолчанию аргумент *mode* равен `GL_CCW`, что означает ориентацию против часовой стрелки для вершин формируемого многоугольника в оконных координатах. Если *mode* равен `GL_CW`, рассматривается многоугольник с ориентацией поверхностей по часовой стрелке.

ПРИМЕЧАНИЕ

Ориентация (по часовой или против часовой стрелки) вершин также называется намоткой — winding.

Несколько слов для полного закрытия вопроса построения поверхностей из непрозрачных многоугольников с последовательной ориентацией: ни один заднеобращенный многоугольник (*back-facing*) не виден — они всегда скрываются лицевыми многоугольниками. Если точка наблюдения расположена снаружи поверхности, вы можете включить отбраковку (*culling*) для отброса многоугольников, которые OpenGL определяет как оборотные. И наоборот, если вы смотрите изнутри объекта, вам видны только заднеобращенные многоугольники. Чтобы OpenGL отбрасывал лицевые или невидимые многоугольники, используйте команду `glCullFace()` и включите отбраковку с помощью функции `glEnable()`.

void `glCullFace`(GLenum *mode*);

Определяет, какие многоугольники должны быть отброшены (отбракованы) перед переводом в оконные координаты. Переменная *mode* принимает значение `GL_FRONT`, `GL_BACK` или `GL_FRONT_AND_BACK` для указания лицевых, оборотных или всех многоугольников. Чтобы добиться результата, необходимо включить отбраковку с помощью функции `glEnable()` со значением `GL_CULL_FACE`. Для отключения отбраковки используется функция `glDisable()` с тем же значением аргумента.

ПРИМЕЧАНИЕ

В более технологичных терминах определение типа многоугольника зависит от области многоугольника, вычисленной в оконных координатах. Один из способов вычисления выполняется по формуле

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_i y_{i\oplus 1} - x_{i\oplus 1} y_i,$$

где x_i и y_i — оконные координаты x и y i-й вершины n-вершинного многоугольника, а

$$i \oplus 1 = (i+1) \bmod n.$$

Допустим, определен режим GL_CCW; если a > 0, вершины многоугольника образуют лицевой многоугольник, в противном случае — оборотный. Если определен режим GL_CW и a < 0, то многоугольник лицевой, иначе — оборотный.

Упражнение

Измените листинг 2.5, добавив в него несколько закрашенных многоугольников Поэкспериментируйте с различными цветами и режимами. Также включите от браковку для оценки вносимого ею эффекта.

Фактурные многоугольники

По умолчанию закраска многоугольников — сплошная. Но, аналогично прерыви стым линиям, она также может быть *фактурной* (stippling), определяемой маской 32 × 32 бита, задаваемой посредством функции `glPolygonStipple()`.

void `glPolygonStipple`(const GLubyte **mask*);

Определяет образец фактурирования для многоугольников. Аргумент *mask* — ука затель на битовую карту размером 32 × 32, которая интерпретируется как маск из 0 и 1. Если появляется 1 — пиксел подсвечивается, а если 0 — нет. Рисунок 2.1 показывает, как из символов аргумента *mask* получается трафарет. Режим фактур ной закраски включается и выключается командами `glEnable()` и `glDisable()` с значением `GL_POLYGON_STIPPLE`. Интерпретация данных *mask* происходит в зави симости от режима хранения, задаваемого параметром `GL_UNPACK*` для команды `glPixelStore*()`. (См. раздел «Режимы хранения пикселов» главы 8.)

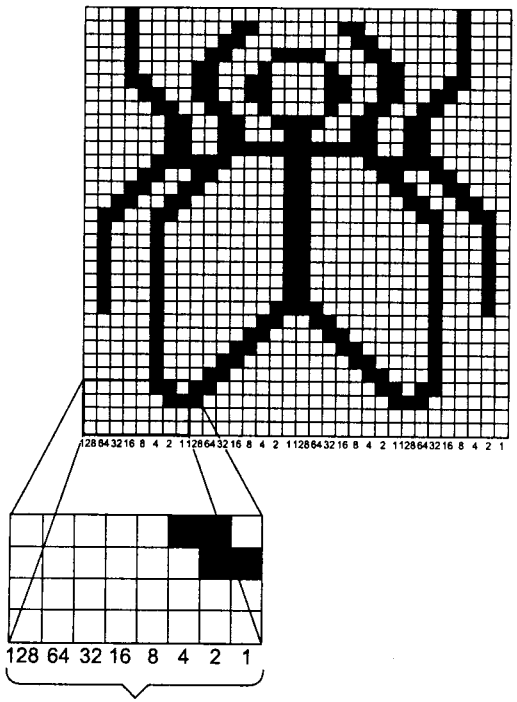


Рис. 2.10. Создание фактурного образца

Кроме задания собственно образца, необходимо включить фактурирование:

```
glEnable(GL_POLYGON_STIPPLE);
```

Для отключения режима фактурирования используется функция `glDisable()` с тем же значением аргумента.

Рисунок 2.11 показывает результат рисования многоугольника со сплошной закраской и двумя разными фактурными закрасками. Текст программы приведен в листинге 2.6. Инверсия белого в черное (из рис. 2.10 в рис. 2.11) возникает из-за того, что программа для случая рис. 2.10 выполняет рисование белым на черном фоне.

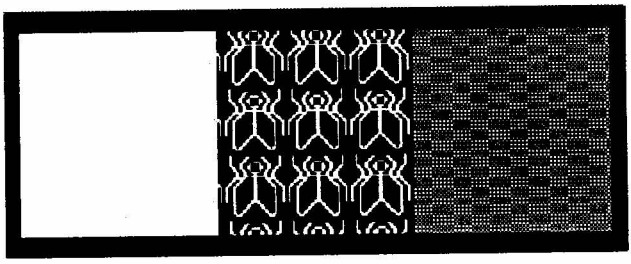


Рис. 2.11. Фактурные многоугольники

Листинг 2.6. Фактурные образцы для многоугольников: polys.c

```
void display(void)
{
    GLubyte fly[] = {
       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
       0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
       0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
       0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
       0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
       0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
       0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
       0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
       0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
       0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
       0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
       0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
       0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
       0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
       0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
       0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};

    GLubyte halftone[] = {
       0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
       0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
       0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
       0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
       0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
       0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
       0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
```

продолжение ⇩

Листинг 2.6 (*продолжение*)

```
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55};
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
/* рисуем один многоугольник с заливкой */
/* затем два фактурных многоугольника   */
    glRectf(25.0, 25.0, 125.0, 125.0);
    glEnable(GL_POLYGON_STIPPLE);
    glPolygonStipple(fly);
    glRectf(125.0, 25.0, 225.0, 125.0);
    glPolygonStipple(halftone);
    glRectf(225.0, 25.0, 325.0, 125.0);
    glDisable(GL_POLYGON_STIPPLE);
    glFlush();
}

void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h); }

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(350, 150);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

Для достижения максимальной эффективности от хранения фактурных образцов используйте списки отображения. (См. раздел «Философия проектирования списков отображения» главы 7).

Маркирование граничных ребер многоугольников

ПРИМЕЧАНИЕ

OpenGL умеет рисовать только выпуклые многоугольники, но на практике используется множество невыпуклых многоугольников. Для рисования их обычно разбивают на выпуклые — обычно треугольники, как показано на рис. 2.12, а затем рисуют треугольники. Конечно, тогда не будет работать функция glPolygonMode() для рисования контура, так как будут отображены все внутренние контуры треугольников. Для решения этой проблемы можно указать OpenGL, какие отдельные вершины определяют границы. На основании этой информации OpenGL назначает каждой вершине бит-идентификатор того, принадлежит она границе или нет. Затем при рисовании многоугольника в режиме GL_LINE неграничные отрезки не рисуются. На рис. 2.12 штриховые линии показывают добавочные нерисуемые границы.

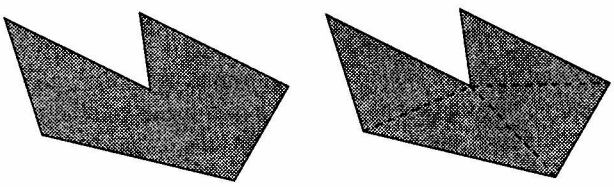


Рис. 2.12. Разделение невыпуклых многоугольников

По умолчанию все вершины помечаются как определяющие границу, но можно вручную контролировать установку *признака границы* с помощью команды `glEdgeFlag*()`. Эта команда используется между операторными скобками `glBegin()` и `glEnd()` и применяется ко всем вершинам, определяемым за ней, до следующего вызова `glEdgeFlag()`. Она применима только к вершинам многоугольников, треугольников и четырехугольников, но не к вершинам, определенным для разбивки треугольников или прямоугольников.

void `glEdgeFlag`(GLboolean *flag*);
void `glEdgeFlagv`(const GLboolean **flag*);
Определяет, относится ли вершина к границе многоугольника. Если *flag* равен `GL_TRUE`, флаг краев устанавливается в `TRUE` (по умолчанию) и любая вершина полагается относящейся к границе до следующего вызова этой функции с *flag*, равным `GL_FALSE`.

Например, программа из листинга 2.7 рисует контур, показанный на рис. 2.13.

Листинг 2.7. Пометка границ многоугольника

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V0);
    glEdgeFlag(GL_FALSE);
    glVertex3fv(V1);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V2);
glEnd();
```

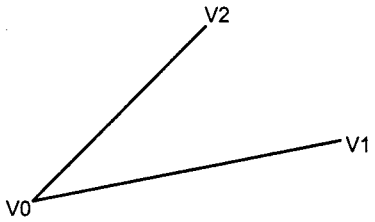


Рис. 2.13. Очерчивание многоугольника при использовании флага краев

Векторы нормали

Вектор нормали (или, коротко, *нормаль*) — вектор в направлении, перпендикулярном к поверхности. Для плоских поверхностей это направление одинаково для всех точек, но для кривых поверхностей оно может быть различным. OpenGL позволяет определять нормали для каждой вершины или каждого многоугольника. Вершины одного многоугольника могут иметь одинаковые (для плоских поверхностей) или различные (для кривых поверхностей) нормали.

Векторы нормалей для объектов определяют ориентацию их поверхностей в пространстве — в частности, относительно источников света. Эти векторы используются OpenGL для расчета, сколько света получает объект в вершинах. Освещение — отдельная большая тема — обсуждается в главе 5, но поскольку нормали определяются вместе с заданием геометрии объекта, рассматриваются они здесь.

Функция `glNormal*()` используется для установки текущей нормали, определяемой значением аргумента. Последующие вызовы `glVertex*()` связывают определяемые вершины с текущей нормалью. Часто каждая вершина имеет собственную нормаль, что влечет серию различных вызовов, как, например, в листинге 2.8.

Листинг 2.8. Нормали к поверхности в вершинах

```
glBegin (GL_POLYGON);
    glNormal3fv(n0);
    glVertex3fv(v0);
    glNormal3fv(n1);
    glVertex3fv(v1);
    glNormal3fv(n2);
    glVertex3fv(v2);
    glNormal3fv(n3);
    glVertex3fv(v3);
glEnd();
```

void `glNormal3`{bsidf}(*TYPE nx, TYPE ny, TYPE nz*);
void `glNormal3`{bsidf}v(const *TYPE *v*);

Устанавливает текущие векторы нормали согласно аргументам. Не векторные версии (без `v`) имеют три аргумента, определяющие вектор нормали (*nx, ny, nz*). Кроме того, есть векторная версия функции (с `v`), принимающая массив из трех элементов для определения нужной нормали. Скалярные версии `b`, `s` и `i` принимают значения в диапазоне [−1.0, 1.0].

Нет никакой магии в нахождении нормалей к объекту — это некоторые вычисления, которые включают в себя вычисление производных. Но схожий эффект дают приемы, раскрытые в приложении Д. Если это для вас не откровение, или если вы предпочитаете готовые результаты, или если вы не хотите использовать освещение, можете и не читать это приложение.

Конечно, в любой точке поверхности есть два перпендикулярных вектора, направленных в противоположных направлениях. Нормалью считается тот, что ориентирован вовне от моделируемой поверхности. Если нужно обратить внешнюю и внутреннюю части модели, измените каждый вектор нормали из (x, y, z) в $(-x, -y, -z)$.

Помните, что векторы нормали определяют только направление и их длина не имеет значения. Нормали могут быть любой длины, но в конечном счете перед вычислением освещенности они все преобразуются к единичной длине. (Такой вектор называется *нормализованным*.) В общем, вам нужны нормализованные векторы нормали. Для получения вектора нормали единичной длины разделите каждую его составляющую x, y, z на длину нормали:

$$\sqrt{x^2 + y^2 + z^2}.$$

Векторы нормали нормируются каждый раз при выполнении модельных преобразований — а именно при повороте и параллельном переносе (см. главу 3). Если вы выполняете нестандартные преобразования (такие, как масштабирование или умножение на матрицу сдвига) или определяете неединичные нормали, можно автоматически нормализовать ваши векторы нормали после преобразований — командой `glEnable(GL_NORMALIZE)`.

Если вы используете единичные нормали и выполняете только однородное масштабирование (одинаковое масштабирование значений x, y и z), включите режим `GL_RESCALE_NORMAL`. Тогда нормали будут масштабироваться к единичной длине постоянным множителем, полученным из матрицы модельно-видовых преобразований.

Помните, что автоматическая нормализация или масштабирование обычно требуют дополнительных вычислений, которые могут снизить производительность ваших приложений. Масштабирование нормалей с помощью `GL_RESCALE_NORMAL` обычно менее времяемкое, чем полная нормализация с помощью `GL_NORMALIZE`. По умолчанию автоматическая нормализация и масштабирование отключены.

Массивы вершин

Разумеется, вы уже заметили, что в OpenGL рисование графических примитивов складывается из множества вызовов функций. Так, рисование 20-стороннего многоугольника сопровождается 22 вызовами: один — `glBegin()`, по одному вызову для каждой вершины, и, наконец, вызов `glEnd()`. Дополнительная информация, такая как в двух предыдущих примерах кода (флаг краев или нормали к поверхности), может привести к удвоению или утроению числа вызовов функций,

необходимых для построения одного геометрического объекта, и, соответственно, к большим задержкам и падению производительности.

Дополнительной проблемой является избыточная обработка вершин, общих для смежных многоугольников. Например, куб на рис. 2.14 имеет 6 лицевых граней и 8 общих вершин. И, если действовать стандартно, каждая вершина будет описана три раза — для каждой лицевой грани, к которой она относится. Таким образом, будет обработано 24 вершины, в то время как достаточно восьми.

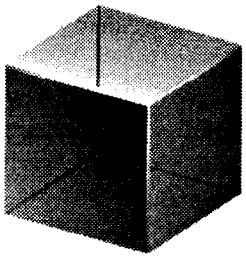


Рис. 2.14. Шесть граней. Восемь общих вершин

OpenGL предоставляет набор функций для работы с массивами вершин. Массивы позволяют определять большое количество информации, связанной с вершинами, и получать доступ к этим данным посредством небольшого количества вызовов функций. В нашем примере все 20 вершин в 20-стороннем многоугольнике могут быть помещены в один массив и обработаны одной функцией. Если для каждой вершины также определены векторы нормалей, все 20 нормалей к поверхности могут быть помещены в другой массив и также обработаны одним вызовом.

Объединение данных в массивы увеличивает производительность приложений за счет безызбыточной обработки общих вершин и минимизации числа вызовов. (Общие вершины не поддерживаются всеми реализациями OpenGL.)

ПРИМЕЧАНИЕ

Массивы вершин стали стандартом в OpenGL версии 1.1. В версии 1.4 добавлена поддержка хранения в них координат тумана и вторичных цветов.

При обработке массивов вершин геометрических объектов выделяются три шага:

1. Объявляем восемь массивов, каждый из которых хранит данные различных типов: координаты вершин, нормали к поверхности, RGBA-цвета, вторичные цвета, индексные цвета, координаты тумана, координаты текстур или флаги краев многоугольников.
2. Помещаем данные в массив или массивы. Доступ к массивам осуществляется по адресу в памяти. В клиент-серверной модели эти данные сохраняются в адресном пространстве памяти клиента.
3. Рисуем геометрию на основе данных. OpenGL получает данные из всех активированных массивов с помощью различных указателей. В клиент-серверной модели данные перемещаются в адресное пространство сервера. Есть три пути для этого, выбор зависит от типа решаемой задачи:

- индивидуальный доступ к элементам массива по индексу (неупорядоченные перемещения);
- создание списка отдельных элементов массива (упорядоченные перемещения);
- обработка последовательных элементов массива.

В версии 1.4 добавлена поддержка множественного доступа к массиву с помощью одной команды.

Смешанные массивы вершин — другой общий метод организации данных. Вместо нескольких различных массивов, каждый из которых хранит данные своего типа (цвет, нормали, координаты и т. д.), можно помещать данные различных типов в один массив. (См. раздел «Смешанные массивы».)

Шаг 1: активация массивов

Сначала необходимо вызвать функцию `glEnableClientState()` со значением, указывающим, какие массивы активируются. Теоретически, потребуется до восьми вызовов этой функции для активации всех восьми массивов. Практически обычно активируется до шести массивов. Например, маловероятно, что потребуется одновременно активировать `GL_COLOR_ARRAY` и `GL_INDEX_ARRAY`, так как программа может работать только в RGBA-режиме или режиме цветовых индексов, но никак не в обоих одновременно.

void `glEnableClientState`(Glenum *array*);
Определяет массив для активации. Аргументом может служить константа из списка со следующими символическими именами: `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_INDEX_ARRAY`, `GL_NORMAL_ARRAY`, `GL_FOG_COORDINATE_ARRAY`, `GL_TEXTURE_COORD_ARRAY` и `GL_EDGE_FLAG_ARRAY`.

При использовании освещения может потребоваться определение векторов нормали для каждой вершины (см. раздел «Векторы нормали».) В этом случае нужно активировать оба массива — массив нормалей к поверхности и массив координат вершин:

```
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

Допустим, необходимо нарисовать геометрический объект только одним цветом, для чего следует выключить освещение. Выключение делается функцией `glDisable()` (см. главу 5). Теперь, когда освещения нет, также надо приостановить изменение значений векторов нормалей, которые не влияют на рисование. Для этого используется функция:

```
glDisableClientState(GL_NORMAL_ARRAY);
```

void `glDisableClientState`(Glenum *array*);
Определяет массив для отключения из того же списка констант, что и для функции `glEnableClientState()`.

Вы можете спросить: почему разработчики OpenGL ввели новые (и длинные) имена функций, вроде `gl*ClientState()`. Почему не использовать функции `glEnable()` и `glDisable()`? Одной из причин явилось то, что функции `glEnable()` и `glDisable()` могут сохраняться в списках отображения, а спецификация массивов вершин этого не допускает, так как данные остаются на стороне клиента.

Если активно мультитекстурирование, включение и отключение клиентских массивов затрагивает только текущую текстуру. (См. раздел «Мультитекстурирование» главы 9 для получения более подробной информации.)

Шаг 2: помещение данных в массивы

Здесь в наших руках две возможности: прямой путь, когда одна команда заполняет одиночный массив на стороне клиента, таких команд восемь по числу массивов; команда, заполняющая несколько клиентских массивов за один раз из единственного смешанного массива.

void `glVertexPointer`(Glint *size*, Glenum *type*, Glsizei *stride*, const GLvoid **pointer*);
Определяет пространственные координаты. Указатель *pointer* — адрес памяти первой координаты первой вершины в массиве. Параметр *type* определяет тип данных (`GL_SHORT`, `GL_INT`, `GL_FLOAT` или `GL_DOUBLE`) для каждой координаты массива. Аргумент *size* — число координат вершины, может принимать значения 2, 3 или 4. Аргумент *stride* — смещение в байтах между соседними вершинами. Если это значение равно 0, вершины считаются плотно упакованными.

Функции для доступа к остальным семи массивам аналогичны:

void `glColorPointer`(Glint *size*, Glenum *type*, Glsizei *stride*, const GLvoid **pointer*);
void `glSecondaryColorPointer`(Glint *size*, Glenum *type*, Glsizei *stride*, const GLvoid **pointer*);
void `glIndexPointer`(Glenum *type*, Glsizei *stride*, const GLvoid **pointer*);
void `glNormalPointer`(Glenum *type*, Glsizei *stride*, const GLvoid **pointer*);
void `glFogCoordPointer`(Glenum *type*, Glsizei *stride*, const GLvoid **pointer*);
void `glTexCoordPointer`(Glint *size*, Glenum *type*, Glsizei *stride*, const GLvoid **pointer*);
void `glEdgeFlagPointer`(Glsizei *stride*, const GLvoid **pointer*);

Основное различие между функциями — должны ли определяться размер и тип. Например, нормали к поверхности всегда имеют три компонента, то есть задание типа является лишним. Флаг краев — всегда типа boolean, и нет необходимости в задании типа данных. Таблица 2.4 содержит список допустимых значений для размера и типов данных.

Для реализаций OpenGL, поддерживающих мультитекстурирование, массив координат текстур определяется с помощью функции `glTexCoordPointer()` толь-

ко для активной в данный момент текстуры. (См. раздел «Мультитекстурирование» главы 9 для получения более подробной информации.)

Таблица 2.4. Массивы вершин. Допустимые значения размеров и типов данных

Команда	Размер	Значения аргумента type (тип данных)
glVertexPointer	2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glColorPointer	3, 4	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glSecondaryColorPointer	3	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glIndexPointer	1	GL_UNSIGNED _BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glNormalPointer	3	GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glFogCoordPointer	1	GL_FLOAT, GL_DOUBLE
glTexCoordPointer	1, 2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glEdgeFlagPointer	1	Тип аргумента не задается (тип данных должен быть GLboolean)

В листинге 2.9 используются массивы вершин для RGBA-цветов и координат вершин. Вещественные значения RGB и соответствующие им целочисленные координаты (x, y) загружаются в массивы GL_COLOR_ARRAY и GL_VERTEX_ARRAY.

Листинг 2.9. Активация и загрузка массивов вершин: varray.c

```
static GLint vertices[] = {25, 25,
                           100, 325,
                           175, 25,
                           175,  325,
                           250, 25,
                           325, 325};
static GLfloat colors[] = {1.0, 0.2, 0.2,
                           0.2, 0.2, 1.0,
                           0.8, 1.0, 0.2,
                           0.75, 0.75, 0.75,
                           0.35, 0.35, 0.35,
                           0.5, 0.5, 0.5};

glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState (GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(2, GL_INT, 0, vertices);
```

Шаг по индексу

Параметр *stride* функции `gl*Pointer()` указывает OpenGL метод доступа к данным, передаваемым указателем на массив. Его значением должно быть количество байтов между началом двух элементов или 0, являющийся частным случаем. Например, данные о вершинах (цвет RGB и координаты (x, y, z)) хранятся в едином массиве вида

```
static GLfloat intertwined[] =
       {1.0, 0.2, 1.0, 100.0, 100.0,0.0,
        1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
        1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
        0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
        0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
        0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

Значения цвета в этом массиве извлекаются от начала массива (которое можно представить в виде *&interwined[0]*), с шагом, равным 6 × `sizeof(GLfloat)` байтов, чего хватает на описание цвета и значений координат вершин. Этот шаг нужен для перемещения к данным следующей вершины:

```
glColorPointer(3, GL_FLOAT, 6*sizeof(GLfloat), &intertwined[0]);
```

Для получения координат вершин нужно начать не с начала массива, а с четвертого элемента (в C отсчет начинается с нуля):

```
glVertexPointer(3, GL_FLOAT, 6*sizeof(GLfloat), &intertwined[3]);
```

Если ваши данные хранятся подобно массиву *intertwined*, обратитесь к разделу «Смешанные массивы», где представлена подробная информация о хранении подобного рода данных.

Если шаг равен нулю, данные каждого типа (RGB-цвет, индексный цвет, координаты и т. д.) должны быть плотно упакованы. Данные в массиве обязаны быть однородными, то есть должны присутствовать все данные о RGB-цвете, все данные о координатах вершин или какие-то другие данные подобного рода.

Шаг 3: разыменование и рисование

Пока содержимое массива вершин разыменовано, массив остается на стороне клиента и его легко изменить. В шаге 3 содержимое массива отправляется серверу, а затем проходит вниз по конвейеру отрисовки.

Как уже говорилось, вы можете получить данные из одиночных элементов массива (по индексу), из упорядоченного списка элементов массива (который может быть ограничен подмножеством данных массива вершин) или из последовательности элементов массива.

Разыменование элемента массива

void `glArrayElement`(Glint *ith*);

Извлекает данные только одной вершины (*ith*) для всех задействованных массивов вершин. Для массива координат вершин соответствующей командой будет `glVertex`[*size*][*type*]`v()`, где *size* принимает значения [2, 3, 4], а *type* — [s, i, f, d] для типов `Glshort`, `Glint`, `GLfloat` и `GLdouble` соответственно. Размер и тип определяются функцией `glVertexPointer()`. Для остальных активных массивов `glArrayElement()` вызывает `glEdgeFlag()`, `glTexCoord`[*size*][*type*]`v()`, `glColor`[*size*][*type*]`v()`, `glSecondaryColor3`[*type*]`v()`, `glIndex`[*type*]`v()`, `glNormal3`[*type*]`v()` и `glFogCoord`[*type*]`v()`. Если включен массив координат вершин, функция `glVertex*v()` исполняется последней после обработки до семи соответствующих значений массивов (если они активны).

Обычно функция `glArrayElement()` помещается между операторными скобками `glBegin()` и `glEnd()`. (Если функция вызывается вне их, она устанавливает текущее состояние для всех активных массивов, за исключением массива вершин, который не имеет текущего состояния.) В программе из листинга 2.10 рисуется треугольник с использованием третьей, четвертой и шестой вершин из доступного массива вершин (еще раз напоминаем, что в языке C отсчет начинается с нуля).

Листинг 2.10. Использование функции glArrayElement() для задания цвета и вершин

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(2, GL_INT, 0, vertices);
glBegin(GL_TRIANGLES);
glArrayElement(2);
glArrayElement(3);
glArrayElement(5);
glEnd();
```

При выполнении последние пять строк равносильны следующему коду:

```
glBegin(GL_TRIANGLES);
glColor3fv(colors + (2 * 3));
glVertex2iv(vertices + (2 * 2));
glColor3 fv(colors + (3 * 3));
glVertex2iv(vertices + (3 * 2));
glColor3 fv(colors + (5 * 3));
glVertex2iv(vertices + (5 * 2));
glEnd();
```

Так как функция `glArrayElement()` вызывается только для одной вершины, она может увеличить число вызовов функций и снизить производительность.

Нелишне предупредить, что если содержимое массивов изменяется между `glBegin()` и `glEnd()`, то нет гарантий, что вы получаете оригинальные данные или изменяете данные интересующего элемента. Для большей уверенности не изменяйте содержимое любых элементов массива, которые могут быть доступны до завершения формирования примитива.

Разыменование элементов массива по списку

Функция `glArrayElement()` удобна для неупорядоченного доступа к данным массивов. Сходные функции `glDrawElements()`, `glMultiDrawElements()` и `glDrawRangeElements()` хороши для доступа к данным в массиве, когда нужен более упорядоченный образ действий.

void `glDrawElements`(GLenum *mode*, GLsizei *count*, GLenum *type*, void **indices*);

Определяет последовательность графических примитивов. Значение *count* — это число элементов, индексы которых хранятся в массиве *indices*. Параметр *type* должен быть равен `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT` или `GL_UNSIGNED_INT`, это тип данных массива *indices*. Аргумент *mode* указывает тип рисуемых примитивов и принимает одно из значений, передаваемых в функцию `glBegin()`; например, `GL_POLYGON`, `GL_LINE_LOOP`, `GL_LINES`, `GL_POINTS` и т. д.

Эффект от функции `glDrawElements()` схож с результатом применения следующей последовательностью команд:

```
glBegin(mode);
for (i = 0;  i < count; i++)
    glArrayElement(indices[i]);
glEnd();
```

Кроме того, функция `glDrawElements()` проверяет корректность значений *mode*, *count* и *type*. Также, в отличие от предыдущей последовательности, функция `glDrawElements()` оставляет некоторые состояния неопределенными. После ее выполнения текущий RGB-цвет, вторичный цвет, цветовой индекс и флаг краев не определены, если соответствующие массивы активированы.

С помощью функции `glDrawElements()` вершины на каждой грани куба могут быть помещены в массив индексов. Листинг 2.11 представляет два способа использования функции `glDrawElements()` для построения куба. Рисунок 2.15 показывает номера вершин, используемых в листинге 2.11.

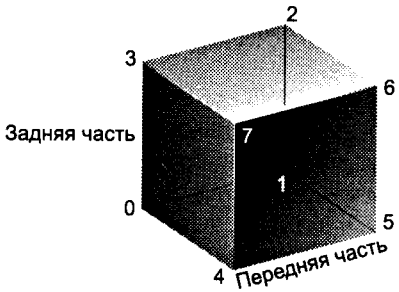


Рис. 2.15. Куб с пронумерованными вершинами

Листинг 2.11. Использование функции glDrawElements() для разыменования нескольких элементов массива

```
static GLubyte frontIndices[] = {4, 5, 6, 7};
static GLubyte rightIndices[]  = {1, 2,  6, 5};
static GLubyte bottomIndices[] = {0, 1, 5, 4};
static GLubyte backIndices[]  = {0, 3, 2, 1};
static GLubyte leftIndices[] = {0, 4, 7, 3};
static GLubyte topIndices[]  = {2, 3, 7, 6};

glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, backIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

ПРИМЕЧАНИЕ

Инкапсуляция glDrawElements() между операторными скобками glBegin()/glEnd() является ошибкой.

С несколькими типами примитивов (такими как `GL_QUADS`, `GL_TRIANGLES` и `GL_LINES`) возможна упаковка нескольких списков индексов в один массив. Так как

примитив GL_QUADS представляет каждую группу из четырех вершин в виде многоугольника, можно упаковать эти индексы, используемые в листинге 2.11, в одиночный массив, как показано в листинге 2.12.

Листинг 2.12. Объединение нескольких вызовов функций glDrawElements() в один

```
static GLubyte allIndices[] = {4, 5, 6, 7, 1, 2, 6, 5,
                               0, 1, 5, 4, 0, 3, 2, 1,
                               0, 4, 7, 3, 2, 3, 7, 6};
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, allIndices);
```

Для других типов примитивов упаковка индексов из различных массивов в одиночный массив даст различные результаты. В листинге 2.13 используются два вызова функции glDrawElements() с примитивом GL_LINE_STRIP для рисования двух прерывистых линий. Вы не можете просто объединить эти два массива и ограничить вызовы функции одним без связывания линий в одну, соединенную в вершинах 6 и 7. (Заметьте, что вершина 1 используется в обеих линиях только для того, чтобы показать, что это допустимо.)

Листинг 2.13. Два вызова функции glDrawElements(), рисующие две прерывистые линии

```
static GLubyte oneIndices[] = {0, 1, 2, 3, 4, 5,  6};
static GLubyte twoIndices[] = {7, 1,  8, 9, 10, 11};

glDrawElements(GL_LINE_STRIP, 7, GL_UNSIGNED_BYTE, oneIndices);
glDrawElements(GL_LINE_STRIP, 6, GL_UNSIGNED_BYTE, twoIndices);
```

В OpenGL версии 1.4 была введена функция glMultiDrawElements() с целью объединения нескольких вызовов функций glDrawElements() в один.

> void glMultiDrawElements(GLenum *mode*, GLsizei **count*, GLenum *type*, void ***indices*, GLsizei *primcount*);
>
> Вызывает последовательность выполнения из *primcount* (количество) функций glDrawElements(). Параметр *indices* — это массив указателей на списки элементов массива. Параметр *count* — массив количеств вершин, найденных в каждом соответствующем элементе списка. Параметры *mode* (тип примитива) и *type* (тип данных) такие же, как в функции glDrawElements().

Вызов функции glMultiDrawElements() равносилен циклу

```
for (i = 0; i < primcount; i++) {
    if (count[i] > 0)
        glDrawElements(mode, count[i], type, indices[i]);
}
```

Вызовы функции glDrawElements() в листинге 2.13 могут быть объединены в одиночный вызов функции glMultiDrawElements(), как показано в листинге 2.14.

Как и функции glDrawElements() и glMultiDrawElements(), функция glDrawRangeElements() предназначена для доступа к данным и рисования их содержимого. Функция glDrawRangeElements() также вносит дополнительные ограничения на значения индексов в расчете на рост производительности программ.

Ради оптимальной производительности некоторые реализации OpenGL могут осуществлять упреждающую выборку ограниченного объема данных о вершинах. Функция `glDrawRangeElements()` позволяет задавать диапазон вершин для упреждающей выборки.

Листинг 2.14. Использование функции glMultiDrawElements(): mvarray.c

```
static GLubyte oneIndices[] = {0, 1, 2, 3, 4, 5,  6};
static GLubyte twoIndices[] = {7, 1,  8, 9, 10, 11};
static GLsizei count[]  = {7,  6};
static GLvoid * indices[2] = {oneIndices, twoIndices};

glMultiDrawElements (GL_LINE_STRIP, count, GL_UNSIGNED_BYTE, indices, 2);
```

void `glDrawRangeElements`(GLenum *mode*, GLuint *start*, GLuint *end*, GLsizei *count*, GLenum *type*, void **indeces*);

Создает последовательность графических примитивов, аналогичных, но более ограниченных, чем определяемые с помощью функции `glDrawElements()`. Аргументы функции `glDrawRangeElements()` подобны соответствующим аргументам функции `glDrawElements()`, включая *mode* (вид примитива), *count* (число элементов), *type* (тип данных) и *indices* (данные о вершинах). Дополнительные аргументы *start* и *end* определяют диапазон значений *indices* включительно.

Ссылка вне диапазона [*start*, *end*] для вершин в массиве *indices* является ошибкой. Однако реализация OpenGL не ищет эту ошибку и не сообщает о ней. Таким образом, некорректное значение индекса может вызвать, а может и не вызвать, ошибку OpenGL, и результат полностью зависит от версии.

Чтобы определить рекомендуемый максимум числа вершин для предварительной обработки и максимального числа индексов (показывает число рисуемых вершин), используйте функцию `glGetIntegerv()` с режимами `GL_MAX_ELEMENTS_VERTICES` и `GL_MAX_ELEMENTS_INDICES`. Если значение *end* – *start* + 1 больше рекомендованного максимума предобрабатываемых индексов, функция `glDrawRangeElements()` нарисует корректное изображение, но производительность может снизиться.

Не каждый индекс в диапазоне [*start*, *end*] обязан задействоваться. Однако в некоторых реализациях, если вы определяете редко используемый диапазон, это влечет лишнюю обработку множества неиспользуемых вершин.

Возможно, что ваша реализация OpenGL, обращаясь к `glArrayElements()`, `glDrawElements()`, `glMultiDrawElements()` и `glDrawRangeElements()`, кэширует недавно востребованные (то есть измененные) вершины, позволяя приложению оперировать теми же данными без повторных преобразований. Возьмем, например, вышеупомянутый куб, у которого шесть граней (многоугольников) и только восемь вершин. Без `gl*Elements()` рисование всех шести граней требует обработки 24 вершин, из них 16 операций являются избыточными. При кэшировании OpenGL минимизирует избыточность и фактически обработает всего восемь вершин. (Повторное использование может быть ограничено внутри одиночного вызова `glDrawElements()` или `glDrawRangeElements()`, одного индекс-

ного массива для функций glMultiDrawElements() или для glArrayElements() внутри пары glBegin()/glEnd().)

Разыменование последовательности элементов массива

В то время как glArrayElements(), glDrawElements() и glDrawRangeElements() «прыгают» по вашим массивам данных, функция glDrawArrays() «перепахивает» их.

void glDrawArrays(GLenum *mode*, Glint *first*, GLsizei *count*);

Строит последовательность графических примитивов, обращаясь к элементам каждого активированного массива начиная с *first* и заканчивая *first* + *count* – 1. Аргумент *mode* определяет тип примитивов и получает такие же значения, как в glBegin(), например: GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS и т. д.

Результат выполнения glDrawArrays() аналогичен следующей последовательности команд:

```
glBegin (mode);
for (i = 0; i < count; i++)
    glArrayElement(first + i);
glEnd();
```

Так же, как и функция glDrawElements(), функция glDrawArrays() выполняет контроль значений своих аргументов и оставляет текущие значения RGB-цвета, вторичного цвета, индексного цвета, координат нормалей, координат тумана, координат текстуры и флага краев неопределенными, если соответствующий массив активирован.

В качестве упражнения можете попробовать изменить способ рисования икосаэдра в листинге 2.16 таким образом, чтобы использовались массивы вершин.

Функция glMultiDrawArrays(), как и функция glMutiDrawElements(), появилась в OpenGL версии 1.4 и предназначена для объединения нескольких вызовов функции glDrawArrays() в один вызов.

void glMultiDrawArrays(GLenum *mode*, Glint **first*, GLsizei **count*, GLsizei *primcount*);

Вызывает последовательность выполнения из *primcount* (количество) функций glDrawArray(). Аргумент *mode* определяет тип примитива, и его значения аналогичны значению аргумента функции glBegin(). Аргументы *first* и *count* определяют, где выполняется обработка каждого списка элементов массива. Для *i*-го элемента массива графический примитив строится начиная с *first[i]* и заканчивая *first[i]* + *count[i]* – 1.

Результат выполнения функции glMultiDrawArrays() аналогичен выполнению

```
for (i = 0; i < primcount; i++) {
    if (count[i] > 0)
        glDrawArrays(mode, first[i], count[i]);
}
```

Смешанные массивы

Ранее в этой главе (см. раздел «Фактурные многоугольники») уже говорилось о массивах с чередованием данных — смешанных. Пусть таким будет массив `intertwined`, хранящий смешанные данные о RGB-цвете и трехмерных координатах вершин. Эти данные можно инициировать вызовами `glColorPointer()` и `glVertexPointer()`:

```
static GLfloat intertwined[] =
       {1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
        1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
        1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
        0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
        0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
        0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

Массивы такого вида можно определить, указывая смещение данных конкретного типа, с помощью шага по индексу. Конечно, это нужно делать осторожно.

Но также существует команда-«монстр» `glInterleavedArrays()`, которая позволяет за один «присест» сформировать несколько массивов. Функция `glInterleavedArrays()` также включает и выключает соответствующие массивы (они перечислены в разделах «Шаг 1: активация массивов» и «Шаг 2: помещение данных в массивы»). Смешанный массив заполняется согласно одной из 14 предопределенных конфигураций, поддерживаемых функцией `glInterleavedArrays()`. Так, чтобы получить активный массив `intertwined` с чередованием RGB-цвета и координат вершин, можно дать следующую команду:

```
glInterleavedArrays(GL_C3F_V3F, 0, intertwined);
```

Она активирует массивы `GL_COLOR_ARRAY` и `GL_VERTEX_ARRAY`, а также отключает массивы `GL_SECONDARY_COLOR_ARRAY`, `GL_INDEX_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_FOG_COORDINATE_ARRAY`, `GL_TEXTURE_COORD_ARRAY` и `GL_EDGE_FLAG_ARRAY`.

Этот вызов дает результат, идентичный результату работы `glColorPointer()` и `glVertexPointer()`, примененных для определения значений шести вершин в каждом массиве. Для рисования примитивов, что соответствует шагу 3, все так же вызываем `glArrayElement()`, `glDrawElements()`, `glDrawRangeElements()` или `glDrawArrays()`.

> void `glInterleavedArrays`(GLenum *format*, GLsizei *stride*, void **pointer*)
>
> Инициализирует все восемь массивов, отключая массивы, не определенные аргументом конфигурации *format*, и включая те, что указаны. Аргумент *format* получает значение одной из 14 символьных констант, перечисленных в табл. 2.5. Значение аргумента *stride* определяет смещение в байтах между смежными вершинами. Если оно равно нулю, подразумевается, что вершины плотно упакованы. Указатель *pointer* ссылается на адрес первой координаты первой вершины в массиве.

Обратите внимание, что функция `glInterleavedArrays()` не поддерживает флаг краев.

Механизм работы `glInterleavedArrays()` сложен и требует иллюстраций в виде табл. 2.5 и листинга 2.15. В этих листинге и таблице есть переменные e_t, e_c и e_n типа boolean, включающие или отключающие координаты текстуры, цвет и векторы нормали; переменные s_t, s_c и s_v, определяющие размер (число компонентов) массивов координат текстуры, цвета и вершин. Переменная t_c — это тип данных для RGBA-цвета, который представляет собой массив, содержащий смешанные значения целого типа. Переменные p_c, p_n и p_v — вычисленные значения смещения для доступа к индивидуальным значениям цвета, нормалей и значения вершин, а s — шаг по индексу, то есть смещение (если оно не определено пользователем) для перемещения между элементами массива.

Эффект выполнения функции `glInterleavedArrays()` подобен результату работы последовательности команд, приведенных в листинге 2.15 со значениями из табл. 2.5. Все операции с указателями выполняются в единицах `sizeof(GLubyte)`.

Листинг 2.15. Результат выполнения команды glInterleavedArrays(format, stride, pointer)

```
int str;
/* устанавливаем et, ec, en, st, sc, sv, tc, pc, pn, pv, и s,
   как показано в табл. 2.5, и значение формата
*/

str = stride;
if (str == 0)
    str = s;
glDisableClientState(GL_EDGE_FLAG_ARRAY);
glDisableClientState (GL_INDEX_ARRAY);
glDisableClientState(GL_SECONDARY_COLOR_ARRAY);
glDisableClientState(GL_FOG_COORDINATE_ARRAY);

if (et) {
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    glTexCoordPointer(st, GL_FLOAT, str, pointer);
}
else
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
if (ec) {
    glEnableClientState(GL_COLOR_ARRAY);
    glColorPointer(sc, tc, str, pointer+pc);
}
else
    glDisableClientState(GL_COLOR_ARRAY);
if (en) {
    glEnableClientState(GL_NORMAL_ARRAY);
    glNormalPointer(GL_FLOAT, str, pointer+pn);
}
else
    glDisableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(sv, GL_FLOAT, str, pointer+pv);
```

В табл. 2.5 буквы T и F обозначают TRUE и FALSE, показатели f — sizeof (GLfloat) и c — четырехкратное значение sizeof(GLubyte), округленное до ближайшего числа, кратного f.

Наиболее простые форматы: GL_V2f, GL_V3F и GL_C3F_V3F. Для любых форматов с _C4UB_ применимы структурные типы данных или обычное приведение типов, а по указателю лежат четыре беззнаковых байта, то есть одно 32-битное слово.

В некоторых реализациях OpenGL использование смешанных массивов может ускорить работу приложения, когда известно точное расположение данных, то есть это случай, когда данные плотно упакованы и могут быть получены за одно обращение. Без смешанных массивов сначала определяется, как плотно упакованы данные, исходя из информации о смещении и размере.

Таблица 2.5. Значения параметров функции glInterleavedArrays()

Format	et	ec	en	st	sc	sv	tc	pc	pn	pv	s
GL_V2F	F	F	F			2				0	2f
GL_V3F	F	F	F			3				0	3f
GL_C4UB_V2F	F	T	F		4	2	GL_UNSIGNED_BYTE	0		c	c + 2f
GL_C4UB_V3F	F	T	F		4	3	GL_UNSIGNED_BYTE	0		c	c + 3f
GL_C3F_V3F	F	T	F		3	3	GL_FLOAT	0		3f	6f
GL_N3F_V3F	F	F	T			3			0	3f	6f
GL_C4F_N3F_V3F	F	T	T		4	3	GL_FLOAT	0	4f	7f	10f
GL_T2F_V3F	T	F	F	2		3				2f	5f
GL_T4F_V4F	T	F	F	4		4				4f	8f
GL_T2F_C4UB_V3F	T	T	F	2	4	3	GL_UNSIGNED_BYTE	2f		c + 2f	c + 5f
GL_T2F_C3F_V3F	T	T	F	2	3	3	GL_FLOAT	2f		5f	8f
GL_T2F_N3F_V3F	T	F	T	2		3			2f	5f	8f
GL_T2F_C4F_N3F_V3F	T	T	T	2	4	3	GL_FLOAT	2f	6f	9f	12f
GL_T4F_C4F_N3F_V4F	T	T	T	4	4	4	GL_FLOAT	4f	8f	11f	15f

ПРИМЕЧАНИЕ

Функция glInterleavedArrays() только активирует или деактивирует массивы вершин и заполняет массив вершин значениями. Она ничего не рисует. Вы должны завершить выполнение, как описано в разделе «Шаг 3: разыменование и рисование», и вызвать функцию glArrayElement(), glDrawElements(), glDrawRangeElements() или glDrawArrays() для разыменования элементов массивов и рисования графики.

Группы атрибутов

Из раздела «Управление основными состояниями» вы узнали, как установить или запросить отдельные состояния или параметры состояния. Это неплохо, но вы можете сохранять и восстанавливать значения набора связанных параметров состояния с помощью всего одной команды.

OpenGL объединяет связанные параметры состояния в *группы*. Например, параметр GL_LINE_BIT сочетает в себе пять параметров состояний: толщины ли-

нии, состояния GL_LINE_STIPPLE, шаблона прерывистой линии, фактора повтора и состояния GL_LINE_SMOOTH (см. раздел «Сглаживание» главы 6). Команды glPushAttrib() и glPopAttrib() позволяют сохранять и восстанавливать одновременно все пять параметров состояния.

Некоторые параметры состояния включены в несколько групп. Например, GL_CULL_FACE одновременно входит в наборы атрибутов многоугольников и активаций.

Начиная с версии 1.1, OpenGL поддерживает два различных стека атрибутов. Дополнительно к основному стеку (который хранит значения серверных параметров состояния), существует клиентский стек атрибутов, доступ к которому осуществляется командами glPushClientAttrib() и glPopClientAttrib().

В целом, использовать эти команды быстрее, чем получать, сохранять и восстанавливать значения самостоятельно. Некоторые значения могут поддерживаться на аппаратном уровне, и получить их не всегда просто. Также, если работа идет на удаленном клиенте, все данные перемещаются по сети для их получения, записи или восстановления. Тем не менее ваша реализация OpenGL может разместить стек атрибутов на сервере, устранив тем самым ненужную сетевую задержку.

Существует около 20 различных наборов атрибутов, которые могут быть сохранены и восстановлены посредством функций glPushAttrib() и glPopAttrib(). Также есть два клиентских набора атрибутов, которые могут быть сохранены и восстановлены функциями glPushClientAttrib() и glPopClientAttrib(). Как для сервера, так и для клиента атрибуты сохраняются в стеке с минимальной глубиной 16 групп. (Чтобы узнать реальную глубину стека для вашей реализации, выполните команду glGetIntegerv() со значениями аргументов GL_MAX_ATTRIB_STACK_DEPTH и GL_MAX_CLIENT_ATTRIB_STACK_DEPTH.) Помещение информации в переполненный стек или ее получение из пустого вызывает ошибку.

(См. приложение Б для получения информации о том, какие атрибуты записываются для определенных значений маски, то есть какие атрибуты принадлежат определенным группам).

void glPushAttrib(GLbitfield *mask*);

void glPopAttrib(void);

Функция glPushAttrib() сохраняет все атрибуты, определяемые битами значения аргумента *mask*, в стек атрибутов. Функция glPophAttrib() восстанавливает значения параметров состояния, сохраненных последним вызовом функции glPushAttrib(). В табл. 2.6 приведен список возможных битов маски; они могут быть объединены логическим ИЛИ (OR), что позволяет сохранять практически любую комбинацию атрибутов. Каждый бит соответствует набору отдельных параметров состояния. Например, GL_LIGHTING_BIT объединяет все параметры состояния, связанные с освещением, что включает в себя цвет текущего материала, фоновый, рассеянный и отраженный свет, самосвечение, список активных источников света и направление распространения света. При вызове функции glPopAttrib() все эти параметры восстанавливаются.

Специальная маска GL_ALL_ATTRIB_BITS предназначена для сохранения и восстановления всех параметров состояния во всех наборах атрибутов.

Таблица 2.6. Группы атрибутов

Маска	Группа
GL_ACCUM_BUFFER_BIT	Буфер накопления
GL_ALL_ATTRIB_BITS	–
GL_COLOR_BUFFER_BIT	Буфер цвета
GL_CURRENT_BIT	Текущий
GL_DEPTH_BUFFER_BIT	Буфер глубины
GL_ENABLE_BIT	Активация
GL_EVAL_BIT	Оценка
GL_FOG_BIT	Туман
GL_HINT_BIT	Подсказки
GL_LIGHTING_BIT	Освещение
GL_LINE_BIT	Линия
GL_LIST_BIT	Список
GL_MULTISAMPLE_BIT	Множественная выборка
GL_PIXEL_MODE_BIT	Пиксел
GL_POINT_BIT	Точка
GL_POLYGON_BIT	Многоугольник
GL_POLYGON_STIPPLE_BIT	Фактурный многоугольник
GL_SCISSOR_BIT	Отсечение
GL_STENCIL_BUFFER_BIT	Буфер трафарета
GL_TEXTURE_BIT	Текстура
GL_TRANSFORM_BIT	Деформация
GL_VIEWPORT_BIT	Точка наблюдения

void `glPushClientAttrib`(GLbitfield *mask*);

void `glPopClientAttrib`(void);

Функция `glPushClientAttrib()` сохраняет все атрибуты, определяемые параметром *mask*, в клиентском стеке атрибутов. Функция `glPopClientAttrib()` восстанавливает значения параметров состояний, ранее сохраненных последним вызовом `glPushClientAttrib()`. В табл. 2.7 приведен список символических имен для битов маски; они могут быть объединены логическим ИЛИ, что позволяет сохранять любую комбинацию клиентских атрибутов.

Две клиентские группы атрибутов, обратной связи и выбора, не могут быть сохранены или восстановлены с помощью стека.

Таблица 2.7. Клиентские группы атрибутов

Маска	Группы
GL_CLIENT_PIXEL_STORE_BIT	Хранение пикселов
GL_CLIENT_VERTEX_ARRAY_BIT	Массив вершин
GL_ALL_CLIENT_ATTRIB_BITS	–
Не могут быть сохранены или восстановлены	Обратная связь
Не могут быть сохранены или восстановлены	Выбор

Несколько советов по построению полигональных моделей поверхностей

Ниже рассмотрены несколько методов, применяемых при полигональной аппроксимации поверхностей. Возможно, когда вы более подробно познакомитесь с освещением и списками отображения (главы 5 и 7), вам захочется еще раз вернуться к этой главе, поскольку это связанные темы. Условия освещения определяют внешний вид рисуемой модели, а некоторые из приведенных ниже приемов более эффективны в связке со списками отображения. Помните, что при вычислении освещенности для получения хорошего результата должны быть определены векторы нормалей.

Полигональная аппроксимация поверхностей — это целое искусство, и ничто не заменит опыта. Но мы постарались наполнить этот раздел несколькими «указателями», которые будут способствовать вам в легком старте.

- Соблюдайте непротиворечивую ориентацию многоугольников. Делайте так, чтобы при просмотре снаружи все многоугольники поверхности были ориентированы в одном направлении (все или по часовой стрелке, или против нее). Поддержка целостной ориентации многоугольников очень важна для отбраковки и при двухстороннем освещении. Попытайтесь следовать этому правилу с самого начала, это поможет избавиться от проблем в будущем. (Если используется функция `glScale*()` с целью отражения геометрических объектов симметрично относительно каких-то осей, следует сменить ориентацию с помощью функции `glFrontFace()` для поддержания непротиворечивости.)
- При разбиении поверхностей следите за всеми нетреугольными многоугольниками. Три вершины треугольника гарантированно лежат в одной плоскости, а уже для вершин четырехугольника такой гарантии нет. Неплоские многоугольники с некоторых точек могут выглядеть пересекающими сами себя, и OpenGL не сумеет правильно их отобразить.
- Всегда существует обратная зависимость между скоростью рисования и качеством изображения. Если разбить поверхность на небольшое число многоугольников, то отрисовка будет быстрая, но может появляться зубчатость; если же сгенерировать миллионы многоугольников, надо думать, выглядеть это будет хорошо, но потребуется много времени на отрисовку. Идеально — вводить специальный параметр в процедуры разбиения, определяющий, насколько хорошо требуется выполнять разбиение, и, если объект рисуется на глазах, можно будет огрубить разбиение. Также при разбиении используйте большие многоугольники там, где поверхность практически ровная, и маленькие — в областях с большой кривизной.
- Для высококачественных изображений нужно более детальное разбиение на границах, чем внутри. Если поверхность поворачивается относительно глаза, границы тоже меняются. Границы появляются, когда векторы нормали перпендикулярны к вектору, направленному от поверхности к точке обзора. Это означает, что их вектор — нулевой. Алгоритм разбиения может выполнить дополнительное разбиение, если вектор в этой точке стремится к нулю.

- Старайтесь избегать Т-образных пересечений в моделях (рис. 2.16). Как видно из рисунка, нет гарантии, что отрезки AB и BC рисуются теми же пикселами, что и отрезок AC. Иногда это происходит, иногда нет; в зависимости от преобразований и положения это может привести к периодическому появлению разрывов на поверхности.

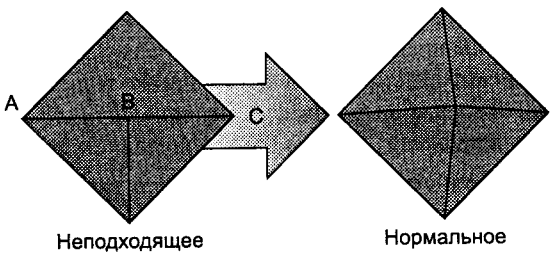


Рис. 2.16. Нежелательные Т-образные пересечения

- При построении замкнутой поверхности убедитесь, что используете одинаковые числа для координат начала и конца контура, иначе на поверхности возникнут разрывы из-за ошибок округления. Ниже приведен пример неправильного кода для рисования двухмерной окружности:

```
/* Не используйте этот код */
#define PI 3.14159265
#define EDGES 3 0

/* рисуем окружность */
glBegin(GL_LINE_STRIP);
for (i = 0; i <= EDGES; i++)
    glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES));
glEnd();
```

Ребра сомкнутся, только если компьютер в силах вычислить одинаковое значение для синуса и косинуса нуля и значения (`2*PI*EDGES/EDGES`). При всем доверии к вычислениям с плавающей запятой, авторы желают вам только лучшего... Для исправления этого кода нужно при i = EDGES подставить для синуса и косинуса значение 0, а не `2*PI*EDGES/EDGES`. (Или просто можно использовать `GL_LINE_LOOP`, а не `GL_LINE_STRIP`, и прерывать цикл по условию $i <$ EDGES.)

Пример: построение икосаэдра

Чтобы проиллюстрировать некоторые из изложенных выше советов по аппроксимации поверхностей, рассмотрим код рисования икосаэдра. Икосаэдр, будучи Платоновым телом из 20 треугольных граней и 12 вершин, является хорошей приблизительной аппроксимацией сферы.

Программа из листинга 2.16 определяет вершины и треугольники, составляющие икосаэдр, а затем его рисует.

Листинг 2.16. Рисование икосаэдра

```
#define X .525731112119133606
#define Z .850650808352039932

static GLfloat vdata[12][3]  =                                              {
    {-X, 0.0, Z},  {X,  0.0, Z},   {-X, 0.0, -Z},    {X,  0.0, -Z},
    {0.0, Z, X},   {0.0, Z, -X},    {0.0, -Z, X},     {0.0, -Z, -X},
    {Z, X, 0.0},   {-Z, X, 0.0},    {Z, -X,  0.0},    {-Z, -X, 0.0}
};
static GLuint tindices[20][3] = {
    {1,4,0}, {4,9,0}, {4,5,9}, {8,5,4}, {1,8,4},
    {1,10,8}, {10,3,8}, {8,3,5}, {3,2,5}, {3,7,2},
    {3,10,7}, {10,6,7}, {6,11,7}, {6,0,11}, {6,1,0},
    {10,1,6}, {11,0,9}, {2,11,9}, {5,2,9}, {11,2,7}
};

int i;

glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
/* информация о цвете должна быть здесь */
    glVertex3fv(&vdata[tindices[i][0]][0]);
    glVertex3fv(&vdata[tindices[i] [1]] [0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();
```

Странные числа X и Z были выбраны таким образом, чтобы расстояние от начала координат до любой вершины составляло 1,0. Координаты всех 12 вершин определены в массиве *vdata*[][], причем нулевая вершина — $\{-X, 0.0, Z\}$, первая — $\{X, 0.0, Z\}$ и т. д. Массив *tindices*[][] определяет, как вершины связываются в треугольники. Например, первый треугольник состоит из нулевой, четвертой и первой вершин. При указании вершин в приведенном порядке все треугольники приобретут одинаковую ориентацию.

Строку с информацией о цвете следует заменить командой, определяющей цвет i-й грани. Если на это место не вставить никакого кода, все грани будут нарисованы одним цветом и нельзя будет разглядеть трехмерный объект. Альтернативой определения цвета для каждой грани является определение векторов нормалей и использование освещения, как описано в следующем разделе.

ПРИМЕЧАНИЕ

Во всех примерах, приведенных в этом разделе, нужно сохранять вычисленные значения вершин и нормалей до тех пор, пока поверхность не нарисована, так как эти вычисления не нужно выполнять при каждом рисовании поверхности. Для сохранения подойдут как собственные структуры данных, так и списки отображения (см. главу 7).

Вычисление векторов нормали к поверхности

Если поверхность освещена, необходимо определить вектор нормали к поверхности. Вычисление нормализованного векторного произведения двух векторов

к поверхности дает вектор нормали. На плоской поверхности икосаэдра все три вершины, определяющие поверхность, имеют одинаковые векторы нормалей. В этом случае достаточно вычислить одну нормаль для каждого набора из трех вершин. Код в листинге 2.17 может заменить строку «информация о цвете...» в листинге 2.16.

Листинг 2.17. Определение векторов нормали к поверхности

```
GLfloat d1[3], d2[3], norm[3];
for (j = 0;  j < 3;  j++)   {
    d1[j]  = vdata[tindices[i][0]][j]  - vdata[tindices[i][1]][j];
    d2[j]  = vdata [ tindices [i] [1] ] [j]  - vdata[tindices[i][2]][j];
}
normcrossprod(d1, d2, norm);
glNormal3fv(norm);
```

Процедура `normcrossprod()` вычисляет нормализованное векторное произведение двух векторов, как показано в листинге 2.18.

Листинг 2.18. Вычисление нормализованного векторного произведения двух векторов

```
void normalize(float v[3])
{
    GLfloat d = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
    if  (d == 0.0) {
        error("вектор нулевой длины");
        return;
    }
    v[0] /= d;
    v[1] /= d;
    v[2] /= d;
}
void normcrossprod( float v1[3], float v2[3], float out [3])
{
    out[0] =v1[1]*v2[2] - v1[2]*v2[1];
    out[1] =v1[2]*v2[0] - v1[0]*v2[2];
    out[2] =v1[0]*v2[1] - v1[1]*v2[0];
    normalize(out);
}
```

Если икосаэдр используется для аппроксимации закрашенной сферы, вы можете захотеть определить векторы нормалей, перпендикулярные к истинной поверхности сферы, а не к граням. Для сферы получение векторов нормалей делается просто: по любой точке в направлении от центра к соответствующей вершине. Так как информация о вершинах определена для икосаэдра с единичным радиусом, векторы нормалей такие же. В следующем коде сглаженная затененная сфера аппроксимируется икосаэдром (полагаем, что освещение включено, как описано в главе 5):

```
glBegin(GL_TRIANGLES);
for (i = 0;  i < 20;  i++)   {
    glNormal3fv(&vdata[tindices[i][0]][0]);
    glVertex3fv(&vdata[tindices[i][0]][0]);
```

```
    glNormal3fv(&vdata[tindices[i][1]][0]);
    glVertex3fv(&vdata[tindices[i][1]][0]);
    glNormal3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();
```

Повышение качества модели

Двадцатигранная аппроксимация сферы не выглядит хорошо, поэтому обратимся к простому способу увеличения точности аппроксимации. Представьте икосаэдр вписанным в сферу и разделенным на треугольники, как показано на рис. 2.17. Поместим на поверхность новые вершины путем нормализации (разделим их на коэффициент для получения единичной длины). Этот процесс разбиения повторяется до достижения необходимой точности. Три объекта на рис. 2.17 состоят из 20, 80 и 320 аппроксимирующих треугольников.

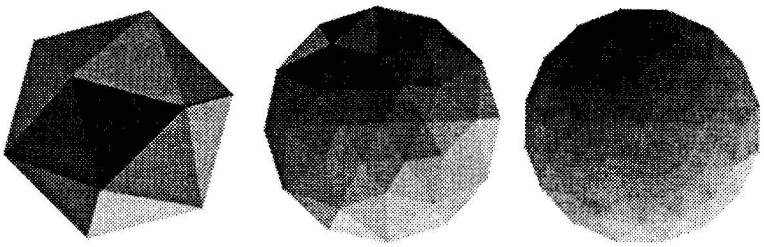


Рис. 2.17. Разукрупнение с целью улучшения полигональной аппроксимации поверхности

Листинг 2.19 демонстрирует получение из 20-гранного икосаэдра 80-гранной поверхности.

Листинг 2.19. Разбиение граней на треугольники

```
void drawtriangle(float *v1, float *v2, float *v3)
{
    glBegin(GL_TRIANGLES);
        glNormal3fv(v1);
        glVertex3fv(v1);
        glNormal3fv(v2);
        glVertex3fv(v2);
        glNormal3 fv(v3);
        glVertex3fv(v3);
    glEnd();
}

void subdivide(float *v1, float *v2, float *v3)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;

    for (i =0; i < 3; i++) {
        v12[i] = (v1[i]+v2[i])/2.0;
        v23[i]  =  (v2[i]+v3[i])/2.0;
```

продолжение

Листинг 2.19 (*продолжение*)

```
        v31[i]  =  (v3[i]+v1[i])/2.0;
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    drawtriangle(v1, v12, v31);
    drawtriangle(v2, v23, v12);
    drawtriangle(v3, v31, v23);
    drawtriangle(v12, v23, v31);
}

for (i = 0; i < 20; i++) {
    subdivide(&vdata[tindices[i][0]][0],
              &vdata[tindices[i][1]][0],
              &vdata[tindices[i][2]][0]);
}
```

Теперь выполним рекурсивное разбиение треугольников до требуемой точности. Для чего в листинг 2.19 внесем небольшие изменения, и это будет уже листинг 2.20. Если значение «глубины» (вложенности) равно 0, разбиение не выполняется и треугольники рисуются «как есть». Если глубина равна 1, разбиение выполняется один раз и т. д.

Листинг 2.20. Рекурсивное разбиение

```
void subdivide(float *v1,  float *v2,  float *v3,  long depth)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;

    if (depth == 0) {
        drawtriangle(v1, v2, v3);
        return;
    }
    for  (i = 0; i < 3; i++) {
        v12[i] = (v1[i]+v2[i])/2.0;
        v23[i] = (v2[i]+v3[i])/2.0;
        v31[i] = (v3[i]+v1[i])/2.0;
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    subdivide(v1, v12, v31, depth-1);
    subdivide(v2, v23, v12, depth-1);
    subdivide(v3, v31, v23, depth-1);
    subdivide(v12, v23, v31, depth-1);
}
```

Обобщенное разбиение

Рассмотренная технология рекурсивного разбиения пригодится и для других видов поверхностей. Обычно рекурсия прекращается, если достигается заданная

глубина или выполняется некоторое условие на кривизну (сильно искривленные части поверхности выглядят лучше при большом разбиении).

Для нахождения более общего решения проблемы разбиения рассмотрим произвольную поверхность, определяемую двумя параметрами, $u[0]$ и $u[1]$. Допустим, имеется две функции:

```
void surf(GLfloat u[2], GLfloat vertex[3], GLfloat normal[3]);
float curv(GLfloat u[2]);
```

При передаче $u[]$ в качестве аргумента функции `surf()` возвращается соответствующая трехмерная вершина и вектор нормали (длиной 1). Если $u[]$ передается функции `curv()`, вычисляется и возвращается кривизна поверхности в данной точке. (См. учебник по дифференциальной геометрии для получения информации о вычислении кривизны.)

Листинг 2.21 показывает рекурсивную процедуру разбиения треугольника до тех пор, пока не достигается максимальная вложенность или максимальная кривизна в трех вершинах не станет меньше некоего предела.

Листинг 2.21. Обобщенное разбиение

```
void subdivide(float u1[2], float u2[2], float u3[2],
               float cutoff, long depth)
{
    GLfloat v1[3], v2[3], v3[3], n1[3], n2[3], n3[3];
    GLfloat u12[2], u23[2], u32[2];
    GLint i;

    if (depth == maxdepth || (curv(u1) < cutoff &&
                 curv(u2) < cutoff && curv(u3) < cutoff)) {
        surf(u1, v1, n1);
        surf(u2, v2, n2);
        surf(u3, v3, n3);
        glBegin(GL_POLYGON);
            glNormal3fv(n1); glVertex3fv(v1);
            glNormal3fv(n2); glVertex3fv(v2);
            glNormal3fv(n3); glVertex3fv(v3);
        glEnd();
        return;
    }
    for  (i = 0; i < 2; i++) {
        u12[i] = (u1[i] + u2[i])/2.0;
        u23[i] = (u2[i] +u3[i])/2.0;
        u31[i] = (u3[i] +u1[i])/2.0;
    }
    subdivide(u1, u12, u31, cutoff, depth+1);
    subdivide(u2, u23, u12, cutoff, depth+1);
    subdivide(u3, u31, u23, cutoff, depth+1);
    subdivide(u12, u23, u31, cutoff, depth+1)
}
```

Визуализация

Прочитав эту главу, вы сможете:

- Просматривать геометрическую модель в любом положении, осуществляя преобразования в трехмерном пространстве
- Управлять местоположением наблюдателя в трехмерном пространстве
- Удалять ненужные части модели из сцены
- Работать с набором матриц, которые управляют изменениями модели, и проецировать модель на экран
- Осуществлять групповые трансформации для имитации сложных систем в движении, таких как солнечная система или рука робота
- Обращать результаты конвейера геометрической обработки

В главе 2 было рассказано, как с помощью OpenGL строить геометрические модели для вывода на сцену. Теперь нужно определиться, как расположить модели в сцене, и определить наиболее удачную точку ее обзора. Конечно, можно использовать и значения по умолчанию, но, скорее всего, вы захотите их изменить.

Посмотрите на цветной рисунок 1. Программа, создавшая это изображение, содержала единственное геометрическое описание «строительного» блока. Каждый блок был тщательно позиционирован в сцене: некоторые блоки были разбросаны на полу, другие уложены друг на друга на столе, а третьи — собраны вместе для создания глобуса. Также нами была выбрана нужная точка обзора (viewpoint). Очевидно, мы хотели видеть угол комнаты с глобусом. Но как далеко от стены и где точно должна располагаться точка обзора? Нам нужно, чтобы в конечной сцене присутствовал хороший вид из окна, была видна часть пола, и все объекты в сцене были не просто видны, но и представлены в интересном ракурсе. Эта глава объясняет, как использовать OpenGL для решения следующих задач: позиционирование и ориентация модели в трехмерном пространстве и определение положения (также и в трехмерном пространстве) наблюдателя. Все это помогает точно определить, какое изображение получится на экране.

Вам следует помнить, что суть компьютерной графики состоит в создании двухмерных образов трехмерных объектов (они должны быть двухмерными, поскольку рисуются на плоском экране), но, принимая решения, определяющие, что именно рисуется на экране, вам следует представлять объект в трех измерениях. Общее заблуждение состоит в том, что при создании трехмерной графики многие с самого начала полагают, что результирующее изображение появится на плоском двумерном экране. Перестаньте думать о том, какие пикселы нужно рисовать, а вместо этого пытайтесь передать трехмерный мир. Представьте модель в некоторой 3D-вселенной, находящейся внутри вашего компьютера, и доверьте последнему работу по прорисовке нужных пикселов.

Совокупность трех операций позволяет компьютеру преобразовать координаты трехмерных объектов в положение пикселов на экране:

- Трансформации, реализуемые умножением матриц, в том числе моделирование, визуализация и проецирование. К таким операциям относятся вращение, перемещение, масштабирование, отражение, ортогональная и перспективная проекции. Сцена строится с помощью комбинации нескольких операций.
- Так как сцена рисуется в прямоугольном окне, объекты (или части объектов), оказывающиеся вне этого окна, должны быть отсечены. В трехмерной компьютерной графике отсечение происходит путем отбрасывания объектов, находящихся за плоскостью отсечения.
- В конечном итоге должно быть установлено соответствие между преобразованными координатами и пикселами на экране. Этот процесс называют преобразованием *окна просмотра* (viewport).

Перечисленные операции и методы управления ими описываются в следующих основных разделах данной главы:

- **Обзор: аналогия с камерой.** Дает представление о процессе преобразования через аналогию с процессом фотографирования. В этом разделе представлен простой пример программы, которая преобразует объект, и кратко описываются основные команды преобразований в OpenGL.
- **Команды преобразования общего назначения.**
- **Модельно-видовые преобразования.** Подробно объясняет, как определить и представить результат модельно-видовых преобразований. Эти преобразования устанавливают взаимное расположение модели и камеры для получения желаемого результата.
- **Преобразования проецирования.** Описывает, как задать форму и расположение *видимого объема* (viewing volume). Видимый объем определяется согласно тому, как сцена проецируется на экран (в перспективной или ортогональной проекции) и какие выступающие за этот объем части объектов отсекаются.
- **Преобразование окна просмотра.** Рассказывает, как контролировать преобразование координат трехмерной модели в координаты на экране.
- **Проблемы преобразований.** Дает несколько советов, как поступить в случае, если вы не можете получить желаемый эффект при моделировании, просмотре, проецировании объектов и вписывании их в окно просмотра.
- **Управление стеком матриц.** Обсуждает, как сохранить и восстановить какие-либо преобразования. Это особенно полезно, когда вы рисуете сложные объекты, состоящие из более простых элементов.
- **Дополнительные плоскости отсечения.** Описывает, как определить дополнительные плоскости отсечения, кроме определенных видимым объемом.
- **Примеры объединения нескольких преобразований.** Продемонстрирует вам несколько более сложное использование преобразований.
- Изучив раздел **«Обратные, или имитационные, преобразования»**, вы узнаете, как из оконных координат точки получить первоначальные координаты объекта. Само преобразование (без инверсии) также может быть эмулировано.

Дополнительно в OpenGL версии 1.3 были добавлены новые функции для поддержки развертывания матриц по строкам, а не по столбцам (транспонирования, в терминах OpenGL).

Обзор: аналогия с камерой

Процесс преобразования, подготавливающий изображение желаемой сцены, сходен с фотографированием. Как показано на рис. 3.1, шаги при работе с фотокамерой (или компьютером) могут быть следующими:

1. Установить штатив и направить камеру на сцену (видовое преобразование).
2. Подготовить фотографируемую сцену в нужной композиции (модельное преобразование).

3. Подобрать объектив или отрегулировать масштабирование (преобразование проекции).
4. Определиться с размером конечной фотографии, например, вы можете захотеть ее увеличить (вписывание в окно просмотра).

После того как все этапы выполнены, изображение может быть зафиксировано или сцена нарисована.

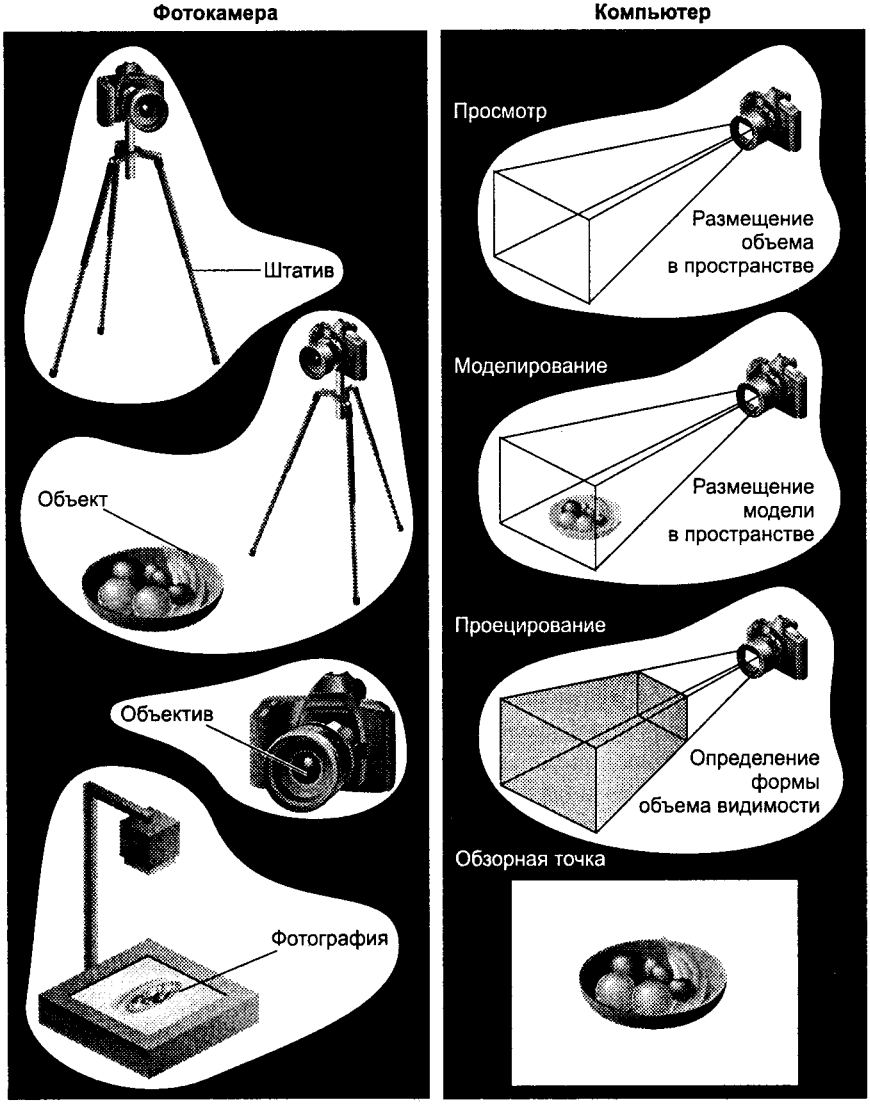


Рис. 3.1. Аналогия с камерой

Обратите внимание, что эти шаги соответствуют последовательности, в которой вы определяете необходимые преобразования в вашей программе, а не обязательно в последовательности, в которой над вершинами объектов выполняются важные математические операции. Видовые преобразования должны предшествовать модельным в коде, но можно определить проекцию и видовое преобразование в любой точке перед началом рисования. Рисунок 3.2 показывает порядок, в котором эти операции происходят на вашем компьютере.

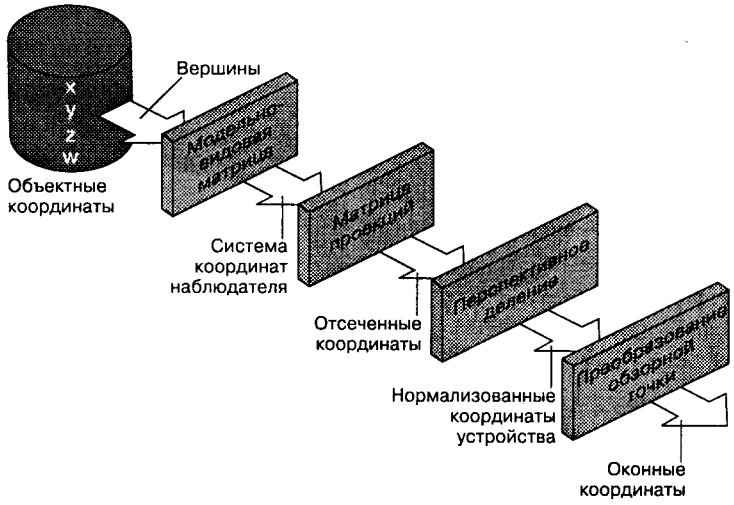


Рис. 3.2. Стадии преобразования вершин

Чтобы определить видовое, модельное и проекционное преобразования, мы создадим матрицу **M** размерности 4 × 4, которая затем будет умножаться на координаты каждой вершины **v** в сцене для осуществления преобразования:

$$v' = \mathbf{M}v.$$

(Помните, что вершины уже имеют четыре координаты (x, y, z, w), хотя в большинстве случаев $w = 1$, а для двухмерных данных $z = 0$.) Обратите внимание, что видовое и модельное преобразования автоматически применяются к векторам нормали к поверхности, дополнительно к координатам вершин. (Векторы нормали используются только в *системе координат наблюдателя*.) Это гарантирует правильное соотношение вектора нормали и данных вершин.

Определенные нами видовое и модельное преобразования формируют матрицу модели, которая применяется к входным координатам объекта для получения выходных координат в системе наблюдения. Затем срабатывают, если такие определены, дополнительные плоскости отсечения с целью удаления некоторых объектов из сцены или для показа объекта в разрезе.

После этого OpenGL применяет матрицу проекции для получения *усеченных координат*. Это преобразование определяет видимый объем; объекты вне его отсекаются и не попадают в финальную сцену. Затем выполняется *деление по перспективе* путем деления значений координат на w для получения *нормализованных*

координат устройства. (См. приложение E для получения дополнительной информации о значении w-координаты и ее влиянии на матричные преобразования.) Наконец, преобразованные координаты переводятся в *оконные координаты* в ходе преобразования окна просмотра. Изображение при этом сжимается или растягивается.

Можно предположить, что координат x и y достаточно для определения точек, отображаемых на экране. Однако все преобразования выполняются также и над координатой z. Это позволяет в конце процесса преобразования с помощью значений z правильно передать *глубину* (удаление от экрана). Одно из применений глубины — устранение ненужных прорисовок. Предположим, что существуют две вершины с одинаковыми координатами x и y, но с различными координатами z. Исходя из этой информации, OpenGL решает, какая из поверхностей закрывается другой, что помогает избежать прорисовок скрытых поверхностей. (См. главы 5 и 10 для получения дополнительной информации об этой технологии, называемой *удаление невидимых поверхностей*.)

Для более полного понимания текущей главы вам необходимо знание действий с матрицами. Если вы хотите пополнить свои знания в этой области, можете взять любой учебник по линейной алгебре.

Пример: рисование куба

Программа из листинга 3.1 рисует куб, масштабируемый модельным преобразованием (рис. 3.3). Видовое преобразование `gluLookAt()` позиционирует камеру и направляет ее туда, где рисуется куб. Также определяются преобразования проекции и окна просмотра. Оставшуюся часть этого раздела занимает рассмотрение листинга 3.1 и краткое объяснение использования команд преобразования. Последующие разделы содержат полное, детальное обсуждение всех команд преобразования OpenGL.

Листинг 3.1. Трансформированный куб: cube.c

```
void init(void)
 {
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}

void display(void)
 {
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glLoadIdentity();       /* очищаем матрицу */
             /* видовое преобразование  */
   gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glScalef(1.0, 2.0, 1.0);       /* модельное преобразование */
   glutWireCube(1.0);
   glFlush();
 }
```

продолжение ⇩

Листинг 3.1 (*продолжение*)

```
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
    glMatrixMode(GL_MODELVIEW);
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

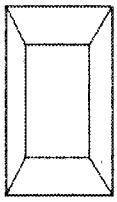


Рис. 3.3. Трансформированный куб

Видовое преобразование

Вспомним, что видовое преобразование аналогично размещению и нацеливанию фотоаппарата. В приведенном примере кода перед определением видового преобразования текущая матрица устанавливается в единичную с помощью функции `glLoadIdentity()`. Это необходимо, так как большинство команд преобразования умножают текущую матрицу на некоторую другую, а затем делают результирующую матрицу текущей. Без загрузки единичной матрицы вы будете накладывать полученный результат на результат предыдущей операции преобразования. В некоторых случаях такие наложения необходимы, но в основном матрица должна сбрасываться.

В листинге 3.1 после того, как матрица инициализирована, осуществляется видовое преобразование функцией `gluLookAt()`. Аргументы этой функции определяют положение камеры (или взгляда), ее направление и угол наклона. Используемые в примере значения помещают камеру в точку пространства с координатами (0, 0, 5), направляют оптическую систему на позицию (0, 0, 0) и задают вектор ориентации (0, 1, 0) — направление от низа к верху основания пирамиды видимого объема.

Если функция `gluLookAt()` не вызывается, действуют значения по умолчанию; то есть камера расположена в начале координат, указывает вниз вдоль оси *z* и имеет вектор ориентации (0, 1, 0). В листинге 3.1 с помощью функции `gluLookAt()` камера перемещается на пять единиц по оси *z*. (См. раздел «Модельно-видовые преобразования» для получения дополнительной информации.)

Модельное преобразование

Модельное преобразование применяется для позиционирования модели и задания ее ориентации. Например, вы можете повернуть, переместить или масштабировать модель — или реализовать некоторую комбинацию этих операций. В листинге 3.1 для таких целей используется функция `glScalef()`. Ее аргументы определяют, как осуществляется масштабирование по трем осям. Если все аргументы равны 1,0, она не имеет смысла. В результате работы программы куб растянут вдвое по оси *y*. Таким образом, если один из углов куба до преобразования находился в точке (3.0, 3.0, 3.0), то после него угол переместился в точку (3.0, 6.0, 3.0). В результате мы получим не куб, а прямоугольный параллелепипед.

В качестве упражнения замените в листинге 3.1 команду `gluLookAt()` на функцию `glTranslate()` с аргументами (0.0, 0.0, –5.0). Результат будет такой же, как и при использовании `gluLookAt()`. Вы уже догадались, почему?

Потому, что здесь вместо перемещения самой камеры (видовое преобразование), позволяющего захватить куб в объектив, мы подвинули куб относительно камеры (модельное преобразование). Эта двойственность объясняет, почему видовое и модельное преобразования объединяют в *модельно-видовую матрицу* перед тем, как их выполнить. (См. раздел «Модельно-видовые преобразования» для получения дополнительной информации.)

Обратите внимание, что видовые и модельные преобразования включены в процедуру `display()` вместе с прорисовкой куба `glutWireCube()`. Это удобно тем, что если, например, окно перемещается или масштабируется, `display()` при повторной прорисовке содержимого окна выполняет все нужные преобразования, что, гарантирует правильное отображение куба. Повторное использование процедуры `display()` подчеркивает необходимость загрузки единичной матрицы перед выполнением видовых и модельных преобразований, особенно если между вызовами процедуры `display()` могли быть проведены другие преобразования.

Проекционное преобразование

Определение преобразования проекции похоже на выбор объектива для фотоаппарата. Это преобразование можно считать определением поля зрения или видимого объема и, соответственно, указанием, какие объекты будут видимы и как они будут выглядеть. Это похоже на выбор между широкоугольным, обычным или телескопическим объективом. С широкоугольным объективом вы можете сделать более широкий кадр, чем с телескопическим, но последний позволяет приблизить удаленные объекты. При использовании компьютерной графики вам не нужно платить 10 000 долларов за 2000-миллиметровый телескопический объектив, достаточно задать меньшее значение для поля зрения.

Дополнительно проекционное преобразование определяет, каким образом объекты *проецируются* на экран. В OpenGL определены два основных типа проекций с наборами соответствующих команд. Первая — *перспективная* проекция; которая не то чтобы заглядывает в будущее, а позволяет видеть вещи так, как мы привыкли в обычной жизни. Перспектива заставляет удаленные объекты казаться меньшими по размеру, чем близкие; например, она «вынуждает» сходиться железнодорожные рельсы. Если вам нужно создать реалистичное изображение, вам стоит выбрать перспективную проекцию, которая определяется в листинге 3.1 с помощью команды `glFrustum()`.

Другим видом проекции является *ортогональная*, которая отображает объекты на экране без воздействия на их размеры. Ортогональная проекция используется в приложениях архитектурного и машинного дизайна, где нужно передать размеры объектов, а не их внешний вид. Архитекторы пользуются перспективой, чтобы показать, как детали зданий или внутреннее пространство выглядят с той или иной точки зрения. Потребность в ортогональной проекции появляется при подготовке точных чертежей, которые используются при конструировании зданий. (См. раздел «Преобразования проецирования».)

Перед вызовом `glFlustum()` следует сделать ряд подготовительных операций. Как показано в процедуре `reshape()` (листинг 3.1), первой выполняется команда `glMatrixMode()` со значением аргумента `GL_PROJECTION`. Это указывает, что текущая матрица определяет проекционное преобразование и что последующие преобразования работают с *матрицей проекций*. Как вы видите, несколькими строками ниже снова вызывается `glMaxtixMode()` со значением аргумента `GL_MODELVIEW`. Это означает, что все последующие преобразования затрагивают матрицу видовых преобразований, а не матрицу проекций. (См. раздел «Управление стеком матриц».)

Обратите внимание, что функция `glLoadIdentity()` используется для инициализации текущей матрицы проекции — для того, чтобы действовало только проекционное преобразование. Теперь можно вызвать `glFlustum()` собственно для преобразования. В данном примере (листинг 3.1) преобразования проекции и окна просмотра объединены в процедуре `reshape()`, которая вызывается при создании окна на экране и всякий раз при его перемещении или изменении размера. Это разумно, поскольку как проецирование, так и вписывание в окно просмотра, — операции, связанные непосредственно с визуализацией и зависят от размеров окна на экране и соотношения его сторон.

Упражнение

Замените `glFlustum()` в листинге 3.1 на подобную GLU-функцию `gluPerspective()` со значениями аргументов (60,0; 1,0; 1,5; 20,0). Затем поэкспериментируйте с различными величинами, особенно для параметров *fovy* и *aspect*.

Преобразование окна просмотра

В совокупности преобразования проекции и окна просмотра определяют, как сцена отображается на экран компьютера. Первое задает механизм отображения,

а окно просмотра отвечает за форму доступного экранного пространства, и, в аналогии с камерой, за размеры и местоположение фотографии — например, должна ли она быть увеличена или уменьшена. Аргументы функции `glViewport()` определяют начало координат доступного экранного пространства — в данном примере (0, 0) — и его ширину и высоту в пикселах. Это объясняет, почему данную команду вызывают в функции `reshape()`: если окно меняет размеры, область просмотра также должна быть изменена. Заметим, что часто ширина и высота приравниваются фактическим ширине и высоте вместо задания абсолютных размеров. (См. одноименный раздел «Преобразование окна просмотра».)

Рисование сцены

Как только все необходимые преобразования выполнены, вы можете рисовать сцену (это похоже на получение фотоснимка после нажатия кнопки). Прорисовывая сцену, OpenGL преобразует каждую вершину каждого объекта посредством модельно-видовых преобразований. Затем каждая вершина преобразуется, как определено проекцией, и отбрасывается, если она находится вне видимого объема, описанного преобразованием проекции. Наконец, прошедшие преобразования вершины разделяются по параметру w и отображаются в окно просмотра.

Команды преобразования общего назначения

В этом разделе обсуждаются некоторые команды OpenGL, которые будут вам полезны при реализации желаемых преобразований. Вы уже знакомы с двумя командами: `glMatrixMode()` и `glLoadIdentity()`. Четыре команды, описанные здесь, — `glLoadMatrix*()`, `glLoadTransposeMatrix*()`, `glMultMatrix*()`, и `glMultTransposeMatrix*()` — позволяют вам определять вручную любую матрицу преобразований или умножать текущую матрицу на нужную. Более специфичные команды преобразования — такие как `gluLookAt()` и `glScale*()` — описаны ниже.

Как говорилось ранее, перед выполнением преобразования нужно обновить матрицу модели или проекции. Выбор матрицы осуществляется командой `glMatrixMode()`. При использовании списков команд OpenGL, которые могут быть вызваны несколько раз, необходимо корректно сбрасывать матрицы. (Команда `glMatrixMode()` применима и для выбора *матрицы текстур*. Текстурирование обсуждается в разделе «Стек матриц текстур» главы 9.)

void `glMatrixMode`(GLenum *mode*);

Определяет, в какую из матриц вносятся изменения — модельно-видовую, проекционную или текстурную с помощью значения параметра *mode*, равного `GL_MODELVIEW`, `GL_PROJECTION` или `GL_TEXTURE` соответственно. Последующие команды преобразования будут работать с указанной матрицей. Обратите внимание, что одновременно может изменяться только одна матрица. По умолчанию работа ведется с модельно-видовой матрицей, и для всех трех режимов матрицы единичные.

Для подготовки текущей матрицы к преобразованиям необходимо выполнить ее перезагрузку командой `glLoadIdentity()`. Как правило, эту команду вызывают всегда перед проецированием или видовыми преобразованиями, но ее также можно вызывать и перед модельным преобразованием.

void `glLoadIdentity`(void);
Устанавливает текущую 4 × 4 матрицу равной единичной.

Если нужно использовать какую-то матрицу вместо текущей, используется функция `glLoadMatrix*()` или `glLoadTransposeMatrix*()`. Аналогично, функция `glMultMatrix*()` или `glMultTransposeMatrix*()` применяется для умножения текущей матрицы на матрицу, принимаемую в аргументе.

void `glLoadMatrix`{fd}(const *TYPE* *m);
Заполняет текущую матрицу 16 значениями, определяемыми аргументом m.

void `glMultMatrix`{fd}(const *TYPE* *m);
Умножает текущую матрицу на матрицу, определяемую аргументом m, и сохраняет результат в текущей матрице.

Все умножения матриц в OpenGL происходят следующим образом. Пусть есть две матрицы: текущая **C** и определенная с помощью команды `glMultMatrix*()` или любой другой команды преобразований матриц матрица **M**. После умножения конечная матрица равна **CM**. Так как операция матричного умножения не обладает свойством коммутативности, порядок множителей имеет значение.

Аргументом `glLoadMatrix*()` и `glMultMatrix*()` является вектор из 16 значений (m_1, m_2,..., m_{16}), который определяет матрицу **M** в порядке столбцов следующим образом:

$$\mathbf{M} = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}.$$

Если вы программируете на C и объявляете матрицу как $m[4][4]$, тогда в OpenGL, где матрицы развернуты в памяти по столбцам, элемент $m[i][j]$ находится на пересечении i-го столбца j-й строки. В стандарте C массивы хранятся по строкам, и здесь этот элемент берется из точки пересечения i-й строки и j-го столбца. Один из способов избежать разночтений между строками и столбцами состоит в объявлении матрицы как $m[16]$.

Другим способом является использование функций OpenGL `glLoadTranspose-Matrix*()` и `glMultTransposeMatrix*()`, которые оперируют с матрицами согласно стандарту C.

void `glLoadTransposeMatrix{fd}`(const *TYPE* *m);
Устанавливает 16 значений текущей матрицы, взятых из матрицы *m*, элементы которой хранятся по строкам. Такой же эффект оказывает вызов функции `glLoadMatrix*`(*mT*).

void `glMultTransposeMatrix{fd}`(const *TYPE* *m);
Умножает матрицу *m*, определяющую 16 значений, на текущую матрицу и сохраняет результат в последней. Эквивалентно вызову функции `glMultMatrix*`(*mT*).

Избежать лишнего пересчета и тем самым повысить эффективность преобразований можно за счет списков отображения, поместив в них часто используемые матрицы (и также им обратные). (См. раздел «Философия проектирования списков отображения» главы 7.)

OpenGL часто приходится выполнять транспонирование модельно-видовой матрицы для того, чтобы нормали и отсекающие плоскости были правильно преобразованы в координаты глаза.

Модельно-видовые преобразования

Модельные и видовые преобразования тесно связаны между собой и в OpenGL фактически объединены в одну модельно-видовую матрицу. (См. раздел «Пример: рисование куба».) Одна из нетривиальных дилемм, появившихся в компьютерной графике недавно, — это выбор соответствующей совокупности трехмерных преобразований. Как вы уже видели, задачи решаются неоднозначно: мы можем переместить камеру в прямом направлении или же объект в противоположном. Каждый из вариантов имеет преимущества и недостатки, но в ряде случаев какой-то один более естественно отражает суть преобразований. Если вы способны найти ненадуманный подход, применимый к вашему приложению, вам будет легче реализовать в программном коде нужные матричные операции, реализующие эти преобразования. Поэтому первая часть данного раздела посвящена вопросам продумывания преобразований, а вторая — командам. Все применяемые в дальнейших примерах команды для работы с матрицами вы уже встречали. Не забывайте, что перед выполнением модельно-видовых преобразований вы должны вызывать функцию `glMatrixMode()` со значением `GL_MODELVIEW` в качестве аргумента.

Размышления о преобразованиях

Начнем с двух простых действий: поворота на 45° против часовой стрелки относительно оси *z* и сдвига вниз по оси *x*. Предположим, что объект, который вы строите, мал по сравнению с перемещением (чтобы лучше видеть эффект от перемещения) и расположен в начале координат. Если сначала повернуть объект, а затем его сместить, объект будет располагаться на оси *x*. Если же сначала сместить его по оси *x*, а затем повернуть относительно начала координат, он попадет на линию $y = x$, как показано на рис. 3.4. То есть порядок действий является критич-

ным. Если вы производите преобразование А, а затем Б, почти всегда результат будет отличным от того, каким бы он был, если бы вы их выполняли в обратном порядке.

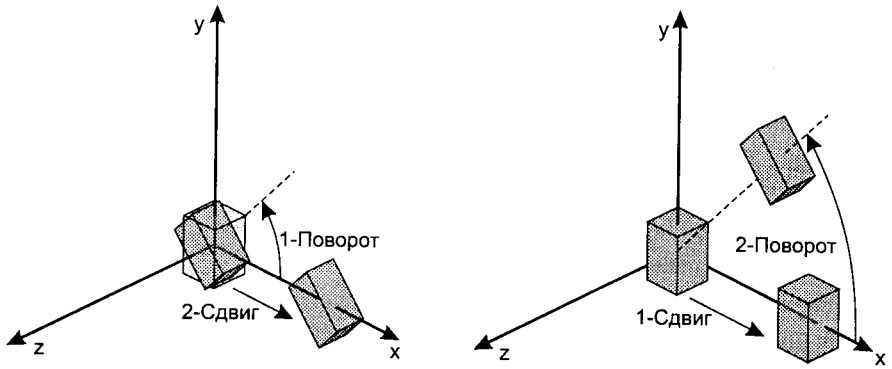


Рис. 3.4. Поворот или сдвиг, вот в чем вопрос

Теперь давайте поговорим о порядке, в котором производятся преобразования. Все модельно-видовые операции представляются матрицами размерностью 4 × 4. Каждое выполнение `glMultMatrix*()` или команды преобразования умножает новую матрицу **M** (4 × 4) на текущую модельно-видовую матрицу **C**; в результате образуется матрица **CM**. Затем вершины *v* умножаются на текущую модельно-видовую матрицу. Это означает, что последняя команда преобразования, вызванная в вашей программе, на самом деле является первой применяемой к вершинам: **CMv**. Таким образом, напрашивается мысль, что матрицы определяются в обратном порядке. Однако, как часто случается, по результатам размышлений что-то противоположное оказывается правильнее скороспелого вывода.

Посмотрите на следующий код, который рисует одну точку с помощью трех преобразований:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N); /* преобразование N */
glMultMatrixf(M); /* преобразование M */
glMultMatrixf(L); /* преобразование L */
glBegin(GL_POINTS);
glVertex3f(v); /* рисуем измененную вершину v */
glEnd();
```

Здесь матрица преобразований претерпевает изменения в порядке **I**, **N**, **NM** и **NML**, где **I** — единичная матрица. Измененная вершина — **NLMv**. Отсюда преобразование вершины равно **N(M(Lv))** — то есть вершина **v** сначала умножается на **L**, результат **Lv** умножается на **M**, и результат **MLv** умножается на **N**. Обратите внимание, что преобразования для **v** происходят фактически в обратном порядке по сравнению с тем, как они были указаны. (Реально выполняется только одно умножение вершины на модельно-видовую матрицу. В данном примере матрицы **N**, **M**, и **L** уже перемножены до того, как они применяются к **v**.)

Система фиксированных координат

Итак, если отталкиваться от системы фиксированных координат, в которой положением, ориентацией и масштабированием модели управляет умножение матриц, можно считать, что умножение осуществляется в обратном порядке относительно следования команд в написанном коде. Из простого примера, показанного в левой части рис. 3.4, становится понятно, что для того, чтобы после выполнения операций объект лежал на оси, необходимо сначала выполнить поворот, а затем сдвиг. А для этого нужно поменять последовательность операций на обратную, то есть код будет выглядеть следующим образом (где **R** — матрица вращения, а **T** — сдвига):

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(T); /* сдвиг */
glMultMatrixf(R); /* вращение */
draw_the_object();
```

Локальная система координат

Можно подойти к вопросу видовых преобразований с другой стороны и забыть о системе фиксированных координат в пользу локальной системы координат, которая привязана к рисуемому объекту. В ней все операции происходят применительно к локальным координатам, а умножение матриц в коде выполняется в естественном порядке написания команд. (Независимо от подхода, код остается тем же, просто воспринимается он иначе.) Возвращаясь к примеру перемещения-поворота, представим объект с прикрепленной к нему системой координат. Параллельный перенос сдвигает объект и его систему координат вниз по оси x. Затем объект поворачивается относительно начала координат (несмещенного), не меняя своего расположения на оси.

Этот вариант годится для приложений наподобие сочлененной руки робота: в плече, локте, запястьях и на каждом пальце. Чтобы оценить, как кончики пальцы переместятся относительно тела, предпочтительно начать вращение и смещение с плеча и опускаться вниз в порядке перечисления. Обратное представление является более запутанным.

Второй подход бывает более сложным, особенно при наличии масштабирования, и тем более когда оно неоднородно, то есть коэффициенты для различных осей разные. После неоднородного масштабирования сдвиг перемещает вершины с учетом множителя от предыдущего преобразования, так как система координат сжата или растянута. Такое масштабирование, смешанное с вращением, может привести к тому, что оси системы координат потеряют перпендикулярность.

Как упоминалось ранее, обычно в коде команды видового преобразования предшествуют командам модельных преобразований. В этом случае вершины модели сначала получают требуемую ориентацию, и затем срабатывают преобразования вида. Видовые команды должны идти первыми, поскольку умножение матриц задается в обратном порядке. Однако не обязательно определять какие-либо модельно-видовые преобразования, если достаточно действий по умолчанию. Когда

преобразования вида не определены, «камера» находится в начале координат, «смотрит» вдоль оси *z* в минус бесконечность. Если не заданы явно модельные преобразования, модель не перемещается и сохраняет свои положение, ориентацию и размер.

Так как команды модельных преобразований могут использоваться вместо видовых преобразований, они *принимаются в рассмотрение* первыми несмотря на то, что команды видовых преобразований в действительности *выдаются* раньше. Программисты при составлении кода должны отталкиваться от этого постулата. Часто сначала пишут весь код, необходимый для композиции сцены, включая позиционирование и ориентирование объектов друг относительно друга. Затем определяется местоположение точки наблюдения, связанной со всей сценой, и добавляется код для соответствующих преобразований просмотра.

Модельные преобразования

В OpenGL определены три команды модельных преобразований: `glTranslate*()`, `glRotate*()` и `glScale*()`. Эти три функции трансформируют объект (или систему координат, если думать об изменении системы координат, а не объекта) посредством сдвига, вращения и масштабирования. Результат эквивалентен мультиплицированию матриц преобразования, вращения или масштабирования с последующим вызовом команды `glMultMatrix*()`, принимающей полученную матрицу в качестве аргумента. Однако означенные команды дают результат быстрее, чем `glMultMatrix*()`. OpenGL автоматически вычисляет нужные матрицы за вас. (См. приложение E для получения более подробной информации.)

Каждое умножение матриц можно описать в терминах операций над вершинами геометрического объекта в фиксированной системе координат или в терминах операций над локальной системой координат, связанной с объектом.

Перемещение

void `glTranslate`{fd}(*TYPE x, TYPE y, TYPE z*);
Перемножает текущую матрицу и матрицу, которая перемещает объект с помощью значений *x*, *y* и *z* (или сдвигает локальную систему координат на те же значения).

Рисунок 3.5 показывает результат использования команды `glTranslate*()`.

Вращение

void `glRotate`{fd}(*TYPE angle, TYPE x, TYPE y, TYPE z*);
Перемножает текущую матрицу и матрицу, вращающую объект (или локальную систему координат) против часовой стрелки вокруг луча из начала координат в точку (*x*, *y*, *z*). Аргумент *angle* определяет угол поворота (в градусах).

Результат работы функции `glRotatef(45,0; 0,0; 0,0; 1,0)`, поворачивающей объект вокруг оси *z* на 45°, показан на рис. 3.6.

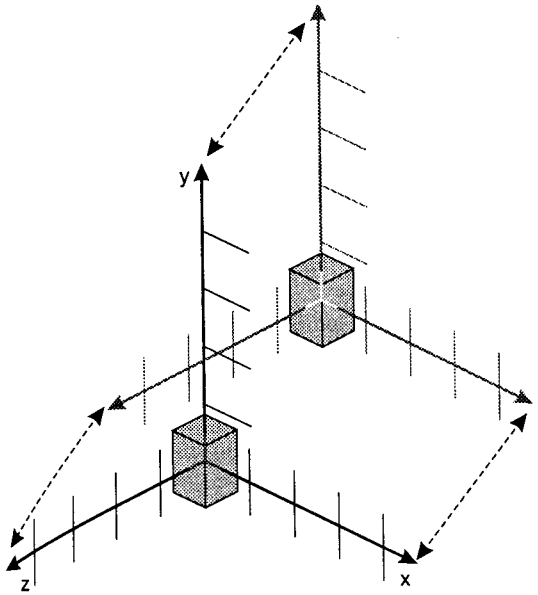


Рис. 3.5. Перемещение объекта

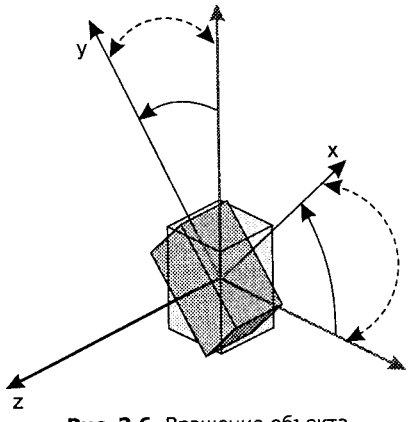


Рис. 3.6. Вращение объекта

Помните, что объект, располагающийся дальше от оси вращения, будет повернут больше (имеет большую орбиту вращения), чем объект, находящийся недалеко от оси вращения. Если значение аргумента *angle* равно нулю, команда `glRotate*()` ничего не делает.

Масштабирование

void `glScale`{fd}(*TYPE x, TYPE y, TYPE z*);
Умножает текущую матрицу на матрицу, определяющую растяжение, сжатие или зеркальное отражение объекта относительно осей. Каждая координата *x*, *y* и *z* каждой точки объекта умножается на соответствующее значение *x*, *y* или *z*. Если рассматривать операцию с точки зрения локальной системы координат, на те же значения растягиваются, сжимаются или отражаются оси локальной системы координат, и, соответственно, трансформируется связанный с ними объект.

Рисунок 3.7 демонстрирует результат выполнения команды

```
glScalef(2.0; -0.5; 1.0).
```

Функция `glScale*()` выполняет три преобразования, изменяющих размеры видимого объекта: масштабирование со значением больше 1.0 растягивает объект, а значение меньше 1.0 уменьшает его. Значение –1.0 приводит к зеркальному отражению объекта относительно оси. Прибегать к функции `glScale*()` рекомендуется только в случаях, когда это действительно необходимо, так как она увеличивает затраты на обсчет освещения, ренормализуя векторы нормалей после преобразования.

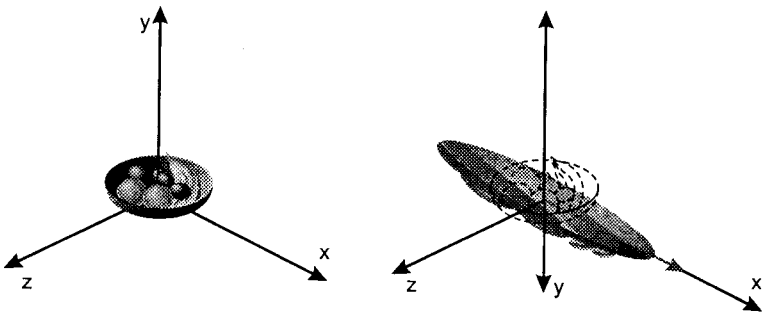


Рис. 3.7. Масштабирование и отражение объекта

ПРИМЕЧАНИЕ

Коэффициент масштабирования, равный нулю, влечет вырождение всех координат объекта в нуль. Это не очень хорошая идея, поскольку отменить действие операции невозможно. Говоря математически, матрицы не могут быть инвертированы, а инвертирование матриц требуется для операций освещения (см. главу 5). Иногда свертывание координат имеет смысл. Одним из таких случаев является вычисление тени на плоской поверхности (см. раздел «Тени» главы 14). Вообще говоря, если система координат коллапсирует в нуль, матрица проекций должна использоваться раньше, чем модельно-видовая матрица.

Пример кода модельного преобразования

В листинге 3.2 приведена часть программы, рисующей четыре раза треугольник, как показано на рис. 3.8. Осуществляются четыре преобразования треугольников:

- рисуется каркасный треугольник без преобразований;
- рисуется такой же треугольник, но линии каркаса штриховые и треугольник смещен (влево — в отрицательную сторону по оси x);
- треугольник рисуется штриховой линией с длинными штрихами, его высота (по оси y) убавлена на 50%, а ширина (по оси x) увеличена на 50%;
- рисуется повернутый треугольник с пунктирными границами.

Листинг 3.2. Использование модельного преобразования: model.c

```
glLoadIdentity();
glColor3f(1.0, 1.0, 1.0);
draw_triangle();                    /* сплошная линия */

glEnable(GL_LINE_STIPPLE);          /* штриховая линия */
glLineStipple(1, 0xF0F0);
glLoadIdentity();
glTranslatef(-20.0, 0.0, 0.0);
draw_triangle();

glLineStipple(1, 0xF00F);           /* длинные штрихи */
glLoadIdentity();
glScalef(1.5, 0.5, 1.0);
draw_triangle();

glLineStipple(1, 0x8888);           /* пунктирная линия */
```

```
glLoadIdentity();
glRotatef(90.0, 0.0, 0.0, 1.0);
draw_triangle();
glDisable(GL_LINE_STIPPLE);
```

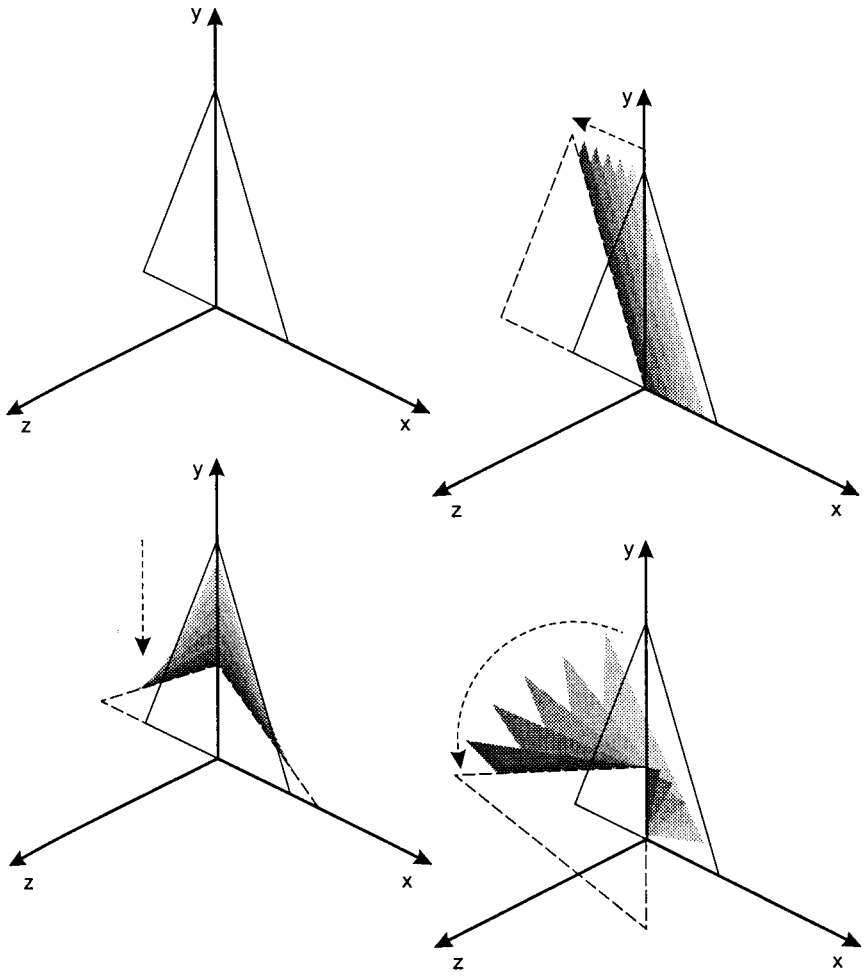


Рис. 3.8. Пример модельного преобразования

ПРИМЕЧАНИЕ

Иногда, когда программисты хотят заставить объект вращаться непрерывно, они пытаются это сделать путем повторных использований матрицы поворота с небольшим шагом. Проблема этого способа в том, что из-за округления результат тысяч небольших вращений постепенно удаляется от теоретического (вплоть до остановки вращения). Правильнее на каждом шаге увеличивать угол наклона и применять преобразование с этим значением.

Помните, что использование функции `glLoadIdentity()` предотвращает накопительный эффект от последующих трансформаций. Многократное ее примене-

ние может сказаться как положительно, так и отрицательно, например, если нужно накапливать результаты преобразований вида или модели. (См. раздел «Управление стеком матриц» для получения более подробной информации.)

Пособие по преобразованиям от Нэйта Робинса

Если вы уже вооружились набором учебных программ от Нэйта Робинса, пришло время запустить пособие по преобразованиям. (См. раздел «Обучающие программы по OpenGL от Нэйта Робинса».) С помощью этого самоучителя вы можете экспериментировать с преобразованиями поворота, сдвига и масштабирования.

Видовые преобразования

Видовое преобразование изменяет положение и ориентацию точки обзора. Если обратиться к аналогии с фотокамерой, преобразование просмотра определяет положение штатива камеры и направление оптической оси объектива на модель. Так как происходит перемещение камеры в некоторую точку и поворот ее до тех пор, пока она не будет указывать в нужном направлении, видовое преобразование просмотра обычно реализуется перемещением и вращением. Не забывайте, что для получения требуемого окончательного облика сцены в изображении или на фотографии можно передвигать либо камеру, либо все объекты сцены, но в противоположных направлениях. Таким образом, например, модельное преобразование, поворачивающее объекты против часовой стрелки, эквивалентно видовому преобразованию, вращающему камеру по часовой стрелке. Наконец помните, что команды видового преобразования должны вызываться до любых модельных преобразований, изменяющих объекты первыми.

Преобразования просмотра выполняются любым из способов, описанных ниже. Кроме того, можно использовать значения по умолчанию для положения и ориентации точки обзора, которые помещают ее в начало координат и направляют вдоль отрицательной оси *z*.

- Посредством одной или нескольких команд модельного преобразования (таких как `glTranslate*()` и `glRotate*()`). Результат можно представить как изменение положения камеры или всех объектов в сцене относительно неподвижной камеры.
- Для определения линии взгляда можно использовать команду библиотеки GLU `gluLookAt()`. Эта команда включает в себя набор команд вращения и перемещения.
- Создать свою собственную команду, включающую вращения и сдвиг. В некоторых приложениях может потребоваться команда, позволяющая удобно и быстро определять специфичное видовое преобразование. Например, можно определить углы крена, тангажа и курса самолета в полете или описать преобразование в терминах полярной системы координат для камеры, вращающейся вокруг объекта по определенной орбите.

Функции glTranslate и glRotate

Эмулируя преобразования просмотра командами модельного преобразования, мы пытаемся переместить позицию наблюдения в нужном направлении, оставляя объекты неподвижными. Так как точка обзора и объекты по умолчанию располагаются в начале координат (рис. 3.9), необходимо провести некоторую подготовку, чтобы объект был видим. Заметьте, что, как показано на рисунке, камера по умолчанию смотрит вниз по оси *z*. (Вы видите заднюю часть камеры.)

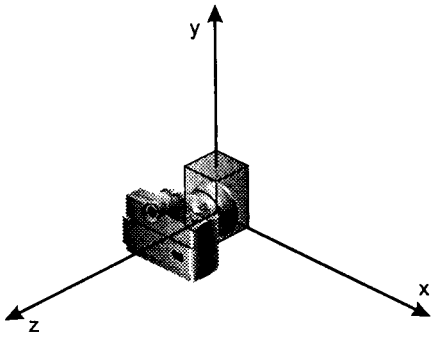


Рис. 3.9. Объект и точка просмотра в начале координат

В самом простом случае можно отодвинуть наблюдателя назад от объектов; это даст такой же эффект, что и перемещение объектов вперед от точки обзора. Помните, что по умолчанию «перед» находится в отрицательном направлении оси *z*; если вращается точка обзора, «перед» имеет другое значение. Таким образом, для отдаления точки обзора на пять единиц от объектов, как показано на рис. 3.10, используется следующая команда:

```
glTranslatef(0,0; 0,0; -5,0);
```

Эта команда перемещает объекты в сцене на −5 единиц вдоль оси *z*, что эквивалентно смещению камеры на +5 единиц вдоль оси *z*.

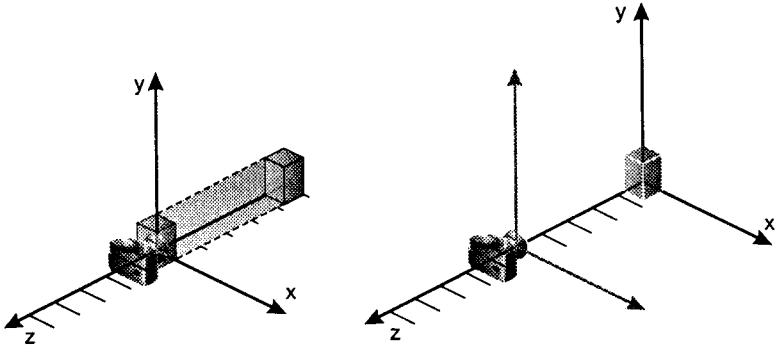


Рис. 3.10. Разделение точки просмотра и объекта

Предположим, что вы хотите взглянуть на объекты со стороны. Когда нужно использовать команду вращения, до или после команды переноса? Если думать

в терминах фиксированной системы координат, для начала надо представить объект и камеру в нуле системы координат. Сперва поворачиваем объект, а затем удаляем его от камеры так, чтобы была видна нужная сторона. В фиксированной системе координат команды должны вызываться в обратном порядке относительно оказываемого эффекта, то есть сначала записываем команду сдвига, а затем — команду вращения.

Теперь представим локальную систему координат. В этом случае при исполнении перемещение объекта и его локальной системы координат от начала координат предшествует вращению и команды записываются в том же порядке. Последовательность команд трансформации для получения требуемого результата следующая:

```
glTranslatef(0.0, 0.0, -5.0);
glRotatef(90.0, 0.0, 1.0, 0.0);
```

Если возникают проблемы с результатом от перемножения матриц, попробуйте обе системы координат — фиксированную и локальную — для определения, какой результат вам больше подходит. Напомним, что в фиксированной системе координат вращение всегда происходит относительно ее начала координат, а в локальной — вокруг нуля локальных координат. Как запасной вариант есть еще функция `gluLookAt()`, описанная ниже.

Функция glLookAt

Часто программисты строят сцену вокруг начала координат или какой-то другой подходящей точки, а затем смотрят на нее из точки, дающей требуемый вид сцены. Как следует из ее имени, функция `gluLookAt()` предназначена для определения точки наблюдения. Она получает три набора аргументов, определяющих положение точки обзора, направление камеры и вектор ориентации. Выберите требуемую точку обзора, дающую нужный вид сцены. Точка, определяющая направление взгляда, будет где-то в центре сцены. (Если сцена строилась в начале координат, этой точкой обычно является начало координат.) Определение вектора ориентации является немного более сложным. Если строится сцена реального мира в начале или вокруг начала координат, то положительное направление оси *y* даст требуемый вектор для функции `gluLookAt()`. Однако если разрабатывается симулятор полета, данный вектор перпендикулярен крыльям самолета и направлен от самолета в небо, когда самолет находится на земле.

> void `gluLookAt`(GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*, GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*, GLdouble *upx*, GLdouble *upy*, GLdouble *upz*);
>
> Определяет видовую матрицу и умножает ее на текущую матрицу. Нужная точка обзора описывается значениями *eyex*, *eyey* и *eyez*. Аргументы *centerx*, *centery* и *centerz* определяют любую точку на линии взгляда, но обычно — некоторую точку в центре сцены. Параметры *upx*, *upy* и *upz* задают вектор ориентации (up-vector) — направление от низа к верху основания пирамиды видимого объема.

Применение функции `gluLookAt()` особенно полезно, когда, например, необходимо получить панораму пейзажа. При объеме видимости, симметричном в на-

правлениях *x* и *y*, точка с координатами (*eyex*, *eyey*, *eyez*) всегда находится в центре сцены, поэтому серия команд с небольшими перемещениями этой точки даст, таким образом, панорамный вид.

По умолчанию камера располагается в начале координат, смотрит вниз по оси *z*, а положительное направление оси *y* указывает вектор ориентации. Это аналогично вызову функции

```
gluLookAt(0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0);
```

Значение *z* для линии взгляда равно −100,0, но может быть любым отрицательным значением *z*, так как линия взгляда останется такой же. Подобный вызов не имеет смысла, поскольку эти значения приняты по умолчанию (рис. 3.11). (Линии, исходящие из камеры, определяют объем видимости, то есть поле зрения.)

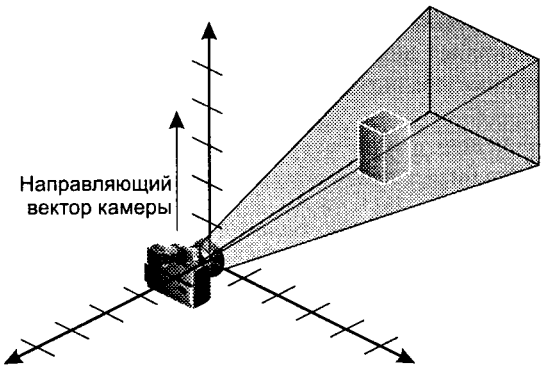


Рис. 3.11. Положение камеры по умолчанию

На рис. 3.12 изображен типичный результат использования функции `gluLookAt()`. Положение камеры определяется точкой (4, 2, 1). Камера нацелена на модель, то есть координаты точки на линии взгляда равны (2, 4, −3). Вектор ориентации (2, 2, −1) поворачивает точку наблюдения на 45°.

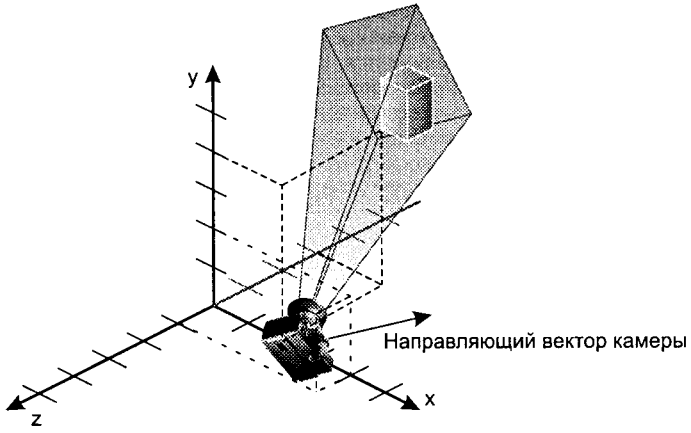


Рис. 3.12. Использование функции gluLookAt()

Вызов, дающий такой результат, выглядит следующим образом:

```
gluLookAt(4.0, 2.0, 1.0, 2.0, 4.0, -3.0, 2.0, 2.0, -1.0);
```

Функция `gluLookAt()` является частью библиотеки GLU, а не основной библиотеки OpenGL. Это сделано не потому, что функция — не нужная, а потому, что она комбинированная — включает в себя несколько базовых функций OpenGL, в частности `glTranslate*()` и `glRotate*()`. Пусть камера находится в произвольной точке обзора и ориентирована согласно линии взгляда, определенной с помощью функции `gluLookAt()`, а сцена расположена в начале координат. Для отмены действия `gluLookAt()` нужно переместить камеру в начало координат и позиционировать ее вдоль отрицательного направления оси *z* в положение по умолчанию. Просто перенесем камеру в начало координат. Легко представить серию поворотов вокруг каждой из трех осей неподвижной системы координат, которые позволят ориентировать камеру вдоль отрицательного направления оси *z*. Так как OpenGL позволяет делать вращение вокруг любой из осей, можно крутить камеру как угодно с помощью одного только вызова команды `glRotate*()`.

ПРИМЕЧАНИЕ

За один раз можно осуществить только одно видовое преобразование. Нельзя объединять результаты двух преобразований, как и камера не устанавливается на двух штативах сразу. Если требуется изменить положение камеры, вызовите предварительно функцию glLoadIdentity(), чтобы предупредить воздействие любых текущих преобразований просмотра.

Пособие по проецированию от Нэйта Робинса

Если у вас есть набор программ от Нэйта Робинса, запустите пособие по проецированию. В этом пособии показано применение `gluLookAt()` с различными значениями аргументов.

ПРИМЕЧАНИЕ

Для преобразования произвольного вектора таким образом, чтобы он совпадал с другим произвольным вектором (например, с отрицательной осью z), нужно провести несколько математических операций. Ось, вокруг которой производится вращение, можно получить с помощью векторного произведения двух нормализованных векторов. Для определения угла поворота оба образующих вектора сначала нормализуются. Косинус искомого угла между двумя векторами равен скалярному произведению нормализованных векторов. Угол вращения вокруг оси, определяемый векторным произведением, всегда принимает значения от 0° до 180°. (О нахождении векторного и скалярного произведений см. приложение Д.) Вычисление угла между двумя нормализованными векторами через арккосинус их скалярного произведения не очень точно, особенно для малых углов, но достаточно для начала работы.

Создание собственных функций

В некоторых специализированных приложениях может потребоваться определить собственные функции преобразования. Так как это делается достаточно редко и это достаточно специализировано, оставим данный вопрос большей частью на самостоятельное рассмотрение читателя. Следующие задания предлагают реализацию двух преобразований просмотра, которые могут пригодиться.

Упражнение

Допустим, пишется авиасимулятор и нужно отобразить мир глазами пилота из кабины самолета. Мир описывается координатной системой с началом на взлетно-посадочной полосе и координатами самолета (x, y, z). Как полагается, у самолета есть крен, наклон самолета к его поперечной оси и линия курса (это углы поворота самолета относительно центра тяжести).

Следующая подпрограмма осуществляет необходимые видовые преобразования:

```
void pilotView(GLdouble planex, GLdouble planey, GLdouble planez, GLdouble roll,
                        GLdouble pitch, GLdouble heading)
{
   glRotated(roll, 0.0, 0.0, 1.0);
   glRotated(pitch, 0.0, 1.0, 0.0);
   glRotated(heading, 1.0, 0.0, 0.0);
   glTranslated(-planex, -planey, -planez);
}
```

Предположим, приложение имитирует камеру, вращающуюся по орбите вокруг объекта, расположенного в начале координат. В этом случае видовое преобразование можно определить в полярной системе координат. Пусть переменная *distance* задает радиус орбиты или расстояние от камеры до начала координат. (Изначально камера смещена на *distance* единиц вдоль положительного направления оси z.) Параметр *azimuth* описывает угол поворота камеры вокруг объекта в плоскости xy, отмеренный относительно положительной полуоси y. Также существует параметр *elevation*, который указывает угол поворота камеры в плоскости yz, относительно положительного направления оси z. Наконец, параметр *twist* обозначает вращение объема видимости вокруг линии взгляда.

Следующая подпрограмма производит предложенные преобразования:

```
void polarView(GLdouble distance, GLdouble twist, GLdouble elevation,
GLdouble azimuth)
{
   glTranslated(0.0, 0.0, -distance);
   glRotated(-twist, 0.0, 0.0, 1.0);
   glRotated(-elevation, 1.0, 0.0, 0.0);
   glRotated(azimuth, 0.0, 0.0, 1.0);
}
```

Преобразования проецирования

В предыдущих разделах рассматривалось, как получить желаемую модельно-видовую матрицу для проведения правильных модельных и видовых преобразований. Данный раздел объясняет, как определить желаемую матрицу проекций, которая также изменяет вершины объектов сцены. Перед использованием любых преобразований, описанных здесь, не забудьте выполнить команды:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

Они применяют все преобразования к матрице проекций и позволяют не затрагивать модельно-видовую матрицу, соответственно, удается избежать смешивания преобразований, которое может появиться потому, что каждая команда преобразования проекций отвечает за отдельное преобразование.

Цель преобразования проецирования — определить *объем видимости*, используемый в двух случаях. Объем видимости устанавливает, как объект отображается на экран (посредством перспективного или ортогонального проецирования), и определяет, какие объекты или части объектов отсекаются в окончательном изображении. Вы можете представить, что точка обзора находится в одном конце видимого объема. Возможно, будет полезно перечитать раздел «Пример: рисование куба» на предмет преобразований, включая преобразование проекции.

Перспективное проецирование

Наиболее очевидная характеристика перспективной проекции — ракурс (foreshortening) в смысле сокращения размеров находящихся дальше от камеры объектов. Видимый объем в перспективной проекции представляет собой *усеченную пирамиду* (вершина которой срезана плоскостью, параллельной основанию). Объекты, попадающие в объем видимости, проецируются к вершине пирамиды, где находится камера или точка обзора, и те из них, что расположены ближе к точке обзора, выглядят больше, чем объекты, находящиеся ближе к основанию пирамиды. Этот метод используется в основном для анимации, визуального моделирования и в приложениях, где необходимо добиться определенной степени реализма, так как его работа схожа с работой глаза (или камеры).

Команда `glFrustum()`, определяющая усеченную пирамиду, вычисляет матрицу перспективной проекции и перемножает ее с текущей матрицей (обычно единичной). Напоминаем, что объем видимости предназначен для отсечения всех объектов, лежащих вне его. Четыре грани пирамиды, ее вершина и основание определяются шестью плоскостями отсечения объема видимости, как показано на рис. 3.13. Объекты или их части, находящиеся вне этих плоскостей, отсекаются в окончательном изображении. Функция `glFrustum()` не требует, чтобы видимый объем был симметричным.

void `glFrustum`(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

Создает матрицу для усеченной пирамиды и умножает на нее текущую матрицу. Объем видимости в виде усеченной пирамиды определяется аргументами в следующем порядке: (*left*, *bottom*, *-near*) и (*right*, *top -near*) задают координаты (x, y, z) левого нижнего и правого верхнего углов ближней плоскости отсечения соответственно; а *near* и *far* уточняют расстояние от точки обзора до ближней и дальней плоскостей отсечения. Они должны быть положительными.

Усеченная пирамида получает задаваемую по умолчанию ориентацию в трехмерном пространстве. Матрица проекций дает возможность выполнить некоторый поворот или перенос для изменения этой ориентации, но это сложно, и почти всегда можно без такого обойтись.

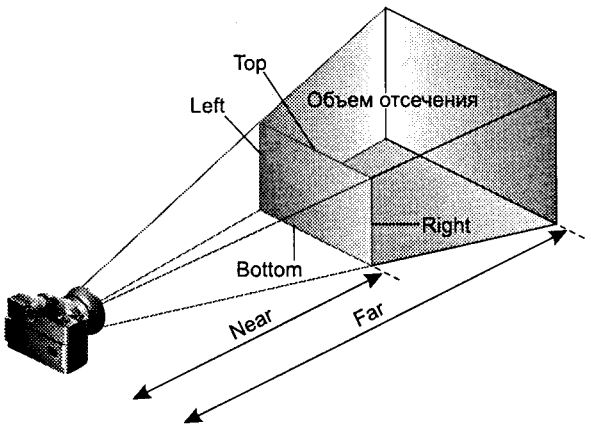


Рис. 3.13. Определение объема видимости с помощью функции glFrustum()

ПРИМЕЧАНИЕ

Усеченная пирамида не обязана быть симметричной, и ее ось не обязательно должна быть параллельна оси z. Например, glFrustum() сработает и для рисования изображения, получаемого при взгляде в прямоугольное окно, когда окно располагается выше, левее наблюдателя или под углом к нему. В фотосъемке такой ракурс применяется для получения необычной (двухточечной) перспективы. Также эта функция помогает получить изображения с разрешением выше нормального экранного, например для распечатки. Допустим, нужно нарисовать изображение с разрешением в два раза большим, чем разрешение экрана. Тогда разрежем основание видимого объема на четыре части, используя усеченную пирамиду, которая заполняет весь экран одной четвертой частью изображения, и выполним четыре рисования. Пиксельные данные каждого квадранта формируют соответствующую часть изображения высокого разрешения, собираемого в конечном итоге. (См. главу 8 для получения информации о чтении пиксельных данных.)

Хотя, в общем, функция `glFrustum()` умозрительно понимаема, она не является понятной на интуитивном уровне. Есть аналогичная по целям функция GLU `gluPerspective()`, которая создает тот же объем видимости, что и функция `glFrustum()`, но другим способом. Вместо определения углов ближней плоскости отсечения берутся угол поля зрения (Θ), как показано на рис. 3.14, в направлении y, и отношение ширины к высоте (x/y). (Ясно, для квадратной части экрана отношение сторон равно 1.0.) Этих двух параметров достаточно для задания неусеченной пирамиды, ориентированной вдоль линии взгляда, как показано на рис. 3.14.

void `gluPerspective`(GLdouble *fovy*, GLdouble *aspect*, GLdouble *near*, GLdouble *far*);

Создает матрицу для симметричной усеченной пирамиды и умножает на нее текущую матрицу. Значение аргумента *fovy* — угол поля зрения в плоскости *yz*. Его значение лежит в диапазоне [0.0, 180.0]. Аргумент *aspect* определяет соотношение сторон усеченной пирамиды — ширины к высоте. Аргументы *near* и *far* указывают расстояние между точкой наблюдения и плоскостями отсечения по оси *z*. Их значения всегда положительны.

Усекается пирамида заданием расстояния между точкой наблюдения и ближней и дальней плоскостями отсечения. Функция `gluPerspective()` умеет создавать

только усеченные пирамиды, симметричные относительно осей x и y, ориентированные по линии взгляда, но обычно этого достаточно.

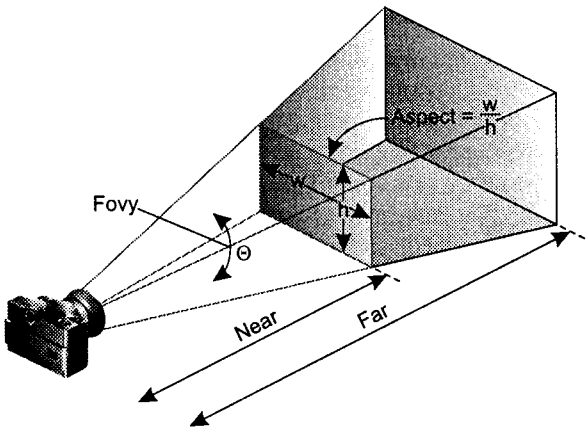


Рис. 3.14. Определение объема видимости с помощью функции gluPerspective()

Так же, как и в случае с `glFrustum()`, здесь допустимо изменить ориентацию объема видимости по умолчанию, с помощью вращения или переноса. По умолчанию наблюдатель считается помещенным в начало координат, а линия взгляда — направленной вдоль отрицательной оси z.

При использовании функции `gluPerspective()` нужно задать правильные значения для поля зрения, иначе изображение будет искажено. Чтобы выяснить, какие значения поля зрения правильны, ответьте на вопрос: как далеко обычно глаз располагается от экрана и насколько велико окно? Затем вычислите угол, соответствующий таким размерам и такому расстоянию. Наверняка он меньше, чем можно было предположить. Для сравнения: 94-градусное поле зрения с 35-миллиметровой камерой требуют 20-миллиметровых линз, которые являются широкоугольными. (См. раздел «Проблемы преобразования» этой главы.)

Конечно, OpenGL все равно, дюймы это или миллиметры. Проецирование и другие преобразования, по сути, безразмерны. Если вам нравится думать, что ближняя и дальняя плоскости отсечения располагаются на расстоянии 1.0 и 20.0 метров, дюймов, километров или парсеков — это ваше право. Главное — использовать одинаковые единицы измерения. Тогда конечное изображение будет с правильными пропорциями.

Ортогональное проецирование

При ортогональном (прямоугольном) проецировании объем видимости представляет собой прямоугольный параллелепипед, или коробку (рис. 3.15). В отличие от перспективного проецирования размер объема видимости не изменяется от одного конца к другому, и расстояние от камеры не влияет на размер объектов. Этот тип проецирования применяется в архитектурных чертежных приложениях и системах автоматизированного проектирования, где необходимо знать реальные размеры объектов и углы между ними.

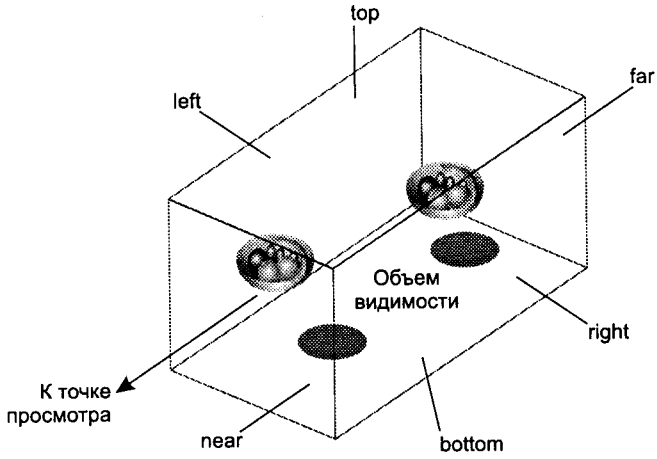


Рис. 3.15. Ортогональный объем видимости

Команда `glOrtho()` создает ортогональный параллельный объем видимости. Так же, как и в функции `glFrustum()`, задаются углы ближней плоскости отсечения и расстояние до дальней плоскости отсечения.

void `glOrtho`(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLDouble *top*, GLdouble *near*, GLdouble *far*);

Создает матрицу для ортогонального объема видимости и умножает на нее текущую матрицу. Аргументы (*left*, *bottom*, *-near*) и (*right*, *top*, *-near*) определяют точки на ближней плоскости отсечения, соответствующие левому нижнему и правому верхнему углам окна просмотра. Аргументы (*left*, *bottom*, *-far*) и (*right*, *top*, *-far*) определяют точки на дальней плоскости отсечения, которые соответствуют тем же проецируемым углам окна просмотра. Аргументы *near* и *far* могут быть положительными, отрицательными и равными нулю, но не одними и теми же.

Если не заданы другие преобразования, направление проецирования параллельно оси *z*, а наблюдатель смотрит в отрицательном направлении оси *z*.

Для особых случаев проецирования двухмерных изображений на двухмерный экран используется функция GLU `gluOrtho2D()`. Она идентична своей трехмерной версии, за тем исключением, что все координаты *z* объектов на экране считаются лежащими в диапазоне от −1.0 до 1.0. Если для построения двухмерных объектов использовались двухмерные функции определения вершин, все координаты *z* равны нулю. Таким образом, никакие объекты не будут отсечены.

void `gluOrtho2D`(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*);

Создает матрицу для проецирования двухмерных координат на экран и умножает на нее текущую матрицу. Отсекающей областью является прямоугольник с левым нижним углом в точке (*left*, *bottom*) и правым верхним углом в точке (*right*, *top*).

Пособие по проецированию от Нэйта Робинса

Если у вас есть набор программ Нэйта Робинса, запустите снова пособие по проецированию. На этот раз поэкспериментируйте с аргументами функций `gluPerspective()`, `glOrtho()` и `glFlustum()`.

Отсечение объемом видимости

После того как все вершины сцены изменены модельно-видовой матрицей и матрицей проекций, каждый примитив, находящийся вне объема видимости, отсекается. Геометрия видимого объема определяется шестью плоскостями отсечения. Вместе с тем можно определять дополнительные плоскости отсечения и располагать их где угодно. (См. раздел «Дополнительные плоскости отсечения».) Запомните, что OpenGL перестраивает углы многоугольников, попадающих под отсечение.

Преобразование окна просмотра

Вернувшись к аналогии с фотокамерой, напомним, что преобразование окна просмотра (viewport) соответствует этапу выбора размера фотографии. Вы хотите сделать стандартную альбомную фотографию или большой плакат? Так как мы имеем дело с компьютерной графикой, окно просмотра — это прямоугольная область окна, в которой рисуется изображение. На рис. 3.16 показано окно просмотра, занимающее собой весь экран. Окно просмотра указывается в координатах, которые отражают позицию пикселов на экране, относительно левого нижнего угла окна. Напомним, что к этому моменту все вершины уже прошли модельно-видовые преобразования и преобразования проекции, а те, что оказались за границами видимого объема, отсечены.

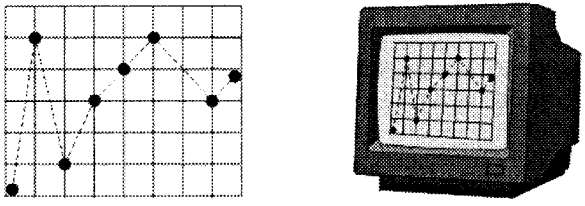


Рис. 3.16. Прямоугольное окно просмотра

Определение окна просмотра

Оконная система, а не OpenGL, отвечает за открытие окна на экране. Однако по умолчанию окно просмотра подразумевает соответствие размерам экранного окна. Функция `glViewport()` может быть использована для выбора меньшей области рисования. Например, так можно получить эффект «картинка в картинке» или отображения одновременно нескольких различных видов в одном окне.

void `glViewport`(Glint *x*, Glint *y*, GLsizei *width*, GLsizei *height*);
Определяет прямоугольник пикселов, в котором показывается изображение. Аргументы (*x*, *y*) задают левый нижний угол окна просмотра, а *width* и *height* устанавливают его размер. По умолчанию начальными значениями окна просмотра являются (*0*, *0*, *winWidht*, *winHeight*), где *winWidht* и *winHeight* — размеры экранного окна.

Аспектное отношение (ширины к высоте) для окна просмотра обычно совпадает с таковым для видимого объема. Иначе проецируемое в окно просмотра изображение будет деформировано (рис. 3.17). Помните, что последующие изменения размера экранного окна сами собой не затрагивают область просмотра. Приложение должно реагировать на эти изменения, соответственно подстраивая окно просмотра.

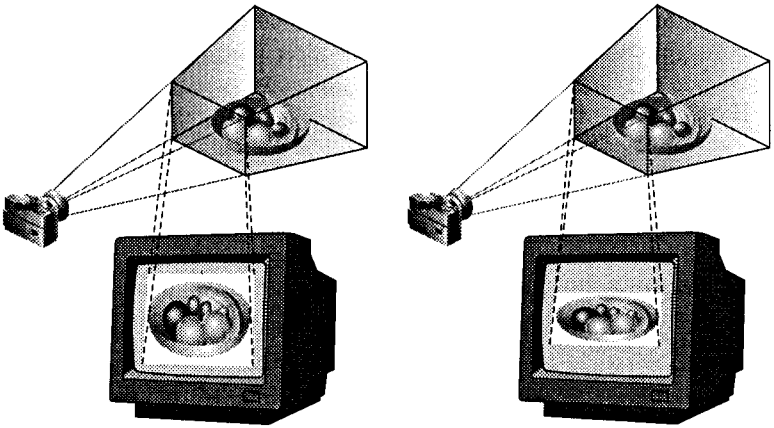


Рис. 3.17. Отображение видимого объема в окно просмотра

На рис. 3.17 для левой части проецирование квадратного изображения на квадратное окно просмотра было произведено следующим набором команд:

```
gluPerspective(fovy, 1.0, near, far);
glViewport0, 0, 400, 400);
```

А для правой части окно приведено к неравносторонней прямоугольной области просмотра, но проекция осталась неизменной. Изображение получается сжатым по оси *x*:

```
gluPerspective(fovy, 1.0, near, far);
glViewport(0, 0, 400, 200);
```

Для устранения несоответствия нужно привести аспектное отношение в соответствие с областью просмотра:

```
gluPerspective(fovy, 2.0, near, far);
glViewport(0, 0, 400, 200);
```

Упражнение

Измените приведенную программу таким образом, чтобы объект рисовался дважды, в разных окнах просмотра. Можете нарисовать объект с различными преобразованиями проецирования и/или вида для каждого окна просмотра. Чтобы создать два смежных окна просмотра, используйте команды, приведенные ниже, вместе с соответствующими командами модельно-видовых преобразований и преобразований проекции:

```
glViewport(0, 0, sizex/2, sizey);
                .
                .
                .
glViewport(sizex/2, 0, sizex/2, sizey);
```

Измененная координата глубины

Координата глубины (z) преобразуется при вписывании в окно просмотра (и после этого сохраняется в буфере глубины). Ее значение можно масштабировать для того, чтобы оно попало в требуемый диапазон, с помощью функции `glDepthRange()`. (Буфер глубины и использование координаты глубины обсуждаются в главе 10.) В отличие от оконных координат x и y, оконная координата z, которой оперирует OpenGL, всегда принимает значения в диапазоне от 0.0 до 1.0.

void `glDepthRange`(GLclampd *near*, GLclampd *far*);

Определяет преобразование для координаты z, происходящее во время отображения на окно просмотра. Аргументы *near* и *far* представляют минимальное и максимальное значения, которые могут быть сохранены в буфере глубины. По умолчанию они равны, соответственно, 0.0 и 1.0. Эти значения принудительно ограничиваются интервалом [0, 1].

В перспективной проекции измененная координата глубины (как и координаты x и y) подвергается перспективному делению по координате w. Чем дальше измененная координата глубины от ближней плоскости отсечения, тем менее точно ее местоположение (рис. 3.18).

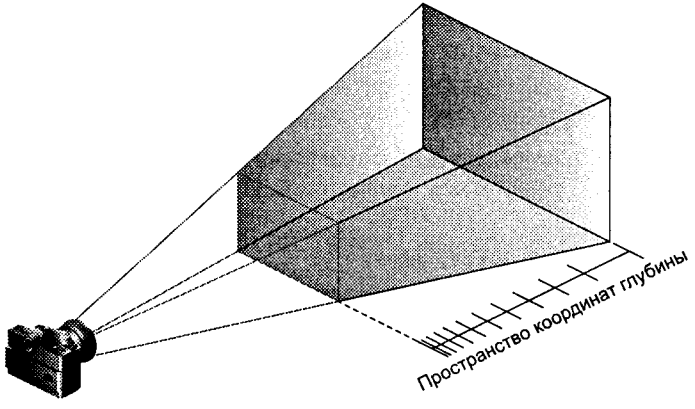


Рис. 3.18. Перспективная проекция и изменение координат глубины

Таким образом, перспективное деление влияет на точность операций, которые опираются на измененную координату глубины, и особенно на буферизацию по глубине, используемую для удаления невидимых поверхностей.

Проблемы преобразований

Фотографу несложно ориентировать камеру в правильном направлении, но в компьютерной графике положение и направление выражаются через углы и координаты. Не составляет труда, и мы можем это засвидетельствовать, получить известный эффект черного экрана. Хотя есть множество других причин для ошибки, чаще всего этот эффект, то есть отсутствие чего бы то ни было на экране, проистекает из-за неправильного позиционирования «камеры», когда модель выпадает из поля зрения. Похожая проблема имеет место, если не выбрано поле зрения, являющееся достаточно широким, чтобы увидеть объекты целиком, и достаточно узким, чтобы объекты были визуально различимы.

Если настойчивость черного экрана превосходит все ваши усилия, попробуйте решить проблему с помощью следующих действий:

1. Проверьте очевидные вещи. Удостоверьтесь, включена ли ваша система. На всякий случай взгляните, не рисуете ли вы цветом, совпадающим с цветом очистки экрана. Проверьте, правильно ли включены/выключены все нужные режимы (такие как освещение, текстурирование, альфа-смешивание, логические операции или устранение ступенчатости).
2. Помните, что в командах проецирования координаты ближней и дальней плоскостей означают расстояние от точки наблюдения и (по умолчанию) линия взгляда направлена вдоль оси *z*. Таким образом, если расстояние до ближней плоскости равно 1.0, а до дальней — 3.0, координаты *z* видимых объектов должны быть в интервале от −1.0 до −3.0. Для того чтобы убедиться, что вы не отсекаете все объекты в сцене, временно установите отступы для ближней и дальней плоскостей отсечения в какие-либо не имеющие смысла значения, например 0.001 и 1 000 000.0. Это изменит результат таких операций, как буферизация по глубине и туман, но позволит отобразить непреднамеренно отсеченные объекты.
3. Оцените, где находится точка обзора, в каком направлении вы смотрите и где располагаются объекты. Так проще получить реальное трехмерное пространство — используйте, например, руки, чтобы вообразить эти вещи.
4. Убедитесь, что вы знаете, вокруг чего вращаетесь. Вы можете считать, что вы поворачиваетесь вокруг произвольной точки, хотя вернулись назад к началу координат. Кружение вокруг любой точки — это нормально, но только если не ожидать такого поворота событий.
5. Проверьте ориентацию. Используйте функцию `gluLookAt()` для нацеливания видимого объема на ваши объекты или рисуйте объекты около начала координат и используйте функцию `glTranslate*()` для переноса камеры в направлении оси *z* на расстояние, достаточное, чтобы все они попали в объем види-

мости. Если у вас получилось захватить все объекты, попытайтесь постепенно изменять видимый объем до тех пор (как описано далее), пока не достигнете нужного результата.

6. Для перспективных преобразований убедитесь, что ближняя плоскость отсечения расположена не слишком близко к наблюдателю (камере) или точность буфера глубины достаточна.

Даже после того как вы нацелили камеру в правильном направлении и увидели объекты, они могут оказаться слишком маленькими или чересчур большими. В случае `gluPerspective()` бывает нужно изменить угол, определяющий поле зрения, подгонкой значения первого аргумента этой функции. Можно прибегнуть к тригонометрии для вычисления нужного угла поля зрения, учтя размеры объекта и его расстояние от точки обзора: тангенс половины требуемого угла равен половине размера объекта, деленной на расстояние до объекта (рис. 3.19). Для вычисления половинного угла от искомого в вашем распоряжении имеется библиотечная функция арктангенса. В листинге 3.3 приведен пример, где арктангенс вычислен с помощью функции `atan2()` из длин прилегающей и противоположной сторон (катетов) прямоугольного треугольника. Результат необходимо преобразовать из радианов в градусы.

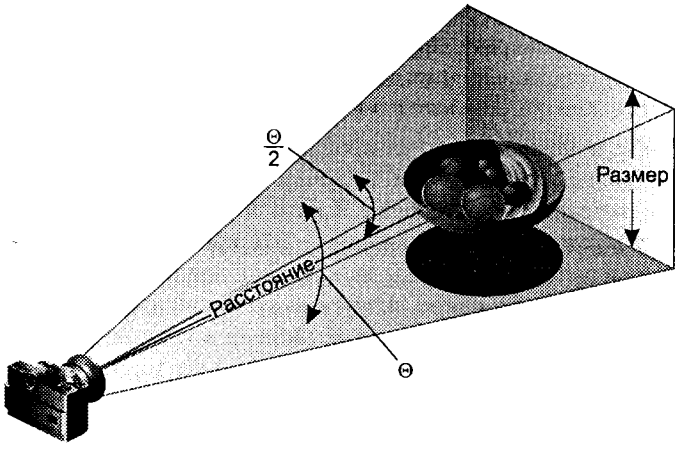


Рис. 3.19. Использование тригонометрии для вычисления поля зрения

Листинг 3.3. Вычисление поля зрения

```
#define PI 3.1415926535

double calculateAngle(double size, double distance)
{
    double radtheta, degtheta;

    radtheta = 2.0 * atan2 (size/2.0, distance);
    degtheta = (180.0 * radtheta) / PI;
    return degtheta;
}
```

Конечно, чаще размер объекта неизвестен, а известно только расстояние между точкой наблюдения и некоторой точкой в сцене. Для получения более точного значения выясните, где проходят границы прямоугольного параллелепипеда, ограничивающего сцену, найдя максимальные и минимальные координаты x, y и z по всем объектам сцены. Затем вычислите радиус сферы, описанной вокруг этого прямоугольного параллелепипеда, и используйте координаты центра сферы для получения расстояния, а радиус — для определения размера.

Пусть все координаты объекта удовлетворяют следующим условиям: $-1 \le x \le 3$, $5 \le y \le 7$ и $-5 \le z \le 5$. Координаты центра будут равны (1, 6, 0), а радиус описанной сферы равен расстоянию от центра до любого угла прямоугольного параллелепипеда, например с координатами (3, 7, 5):

$$\sqrt{(3-1)^2 + (7+6)^2 + (5-0)^2} = \sqrt{30} = 5.477.$$

Если координаты точки наблюдения — (8, 9, 10), расстояние от нее до центра равно

$$\sqrt{(8-1)^2 + (9-6)^2 + (10-0)^2} = \sqrt{158} = 12.570.$$

Тангенс половинного угла равен 5.477, деленным на 12.570, что составляет 0.4357, то есть половинный угол равен 23.54°.

Помните, что угол поля зрения влияет на оптимальное расположение точки обзора, если вы хотите получить реалистичное изображение. Например, если результатом вычисления явился угол в 179°, точка наблюдения должна быть удалена на долю дюйма от экрана для поддержания реализма. Если полученное значение угла просмотра слишком большое, нужно переместить позицию наблюдателя дальше от объекта.

Управление стеком матриц

Матрицы модельно-видовых преобразований и преобразований проекции, которые вы создаете, загружаете и перемножаете, это только вершина айсберга. Каждая из этих матриц — фактически самый верхний элемент стека (набора) матриц (рис. 3.20).

Рис. 3.20. Стек модельно-видовых матриц и матриц проекций

Стек матриц облегчает создание иерархических моделей, в которых сложные объекты составляются из простых. Допустим, вы рисуете автомобиль. У него есть четыре колеса, каждое из которых зафиксировано пятью болтами. В вашем распоряжении подпрограмма, рисующая обобщенное колесо, и подпрограмма, рисующая болт, так как все колеса и болты выглядят одинаково. Эти подпрограммы рисуют колесо или болт в некоторых заданных положениях и с конкретной ориентацией — например, центрированные в начале координат с осями, совпадающими с осью *z*. При рисовании колесно-болтового автомобиля необходимо вызвать функции рисования колес четыре раза с различными преобразованиями каждый раз, чтобы правильно расположить колеса. Для каждого колеса нужно нарисовать пять болтов, каждый раз располагая их в соответствующем месте соответствующего диска.

Предположим на минуту, что все необходимые действия — это нарисовать колеса и кузов машины. Применительно к русскому языку это звучит примерно так:

«Рисуем кузов машины. Запоминаем положение и переходим к правому переднему колесу. Рисуем колесо и отменяем последний переход, то есть текущее положение возвращается в центр кузова. Запоминаем позицию и переходим к левому переднему колесу...»

Точно так же для каждого колеса, если нужно нарисовать болт, запоминаем текущее положение и переходим к позиции каждого болта, отменяя преобразование после каждого рисования болта.

Так как преобразования хранятся в матрицах, стек (набор) матриц представляет собой идеальный механизм для данных операций запоминания, перехода и отката. Все матричные операции, описанные далее (`glLoadMatrix()`, `glLoadTransposeMatrix()`, `glMultMatrix()`, `glMultTransposeMatrix()`, `glLoadIdentity()` и команды создания специфичных матриц преобразований), связаны с текущей матрицей или с верхней матрицей стека. Управлять тем, какая матрица является верхней, можно с помощью команд управления стеком: `glPushMatrix()`, которая помещает текущую матрицу на вершину стека, и `glPopMatrix()`, которая отбрасывает верхнюю матрицу стека, как показано на рис. 3.21. (Как для любого стека, текущий элемент всегда находится на вершине стека.) По сути, команда `glPushMatrix()` означает «запомним текущее состояние», а команда `glPopMatrix()` — «вернемся к предыдущему состоянию».

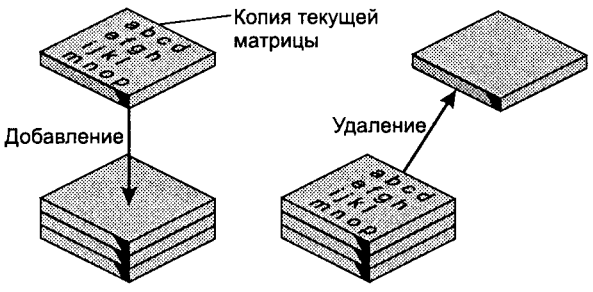


Рис. 3.21. Работа со стеком матриц

void `glPushMatrix`(void);

Перемещает все матрицы в текущем стеке матриц на один уровень вниз. Текущий стек матриц определяется с помощью команды `glMartrixMode()`. Верхняя матрица копируется, то есть ее содержимое дублируется в верхней и второй сверху матрице. Если матриц слишком много, генерируется ошибка.

void `glPopMatrix`(void);

Удаляет верхнюю матрицу из стека, уничтожая ее содержимое. Вторая сверху матрица становится верхней. Текущий стек определяется с помощью команды `glMatrixMode()`. Если в стеке содержится единственная матрица, вызов функции `glPopMatrix()` приводит к ошибке.

Программа из листинга 3.4 рисует автомобиль в последовательности кузов—колеса—болты.

Листинг 3.4. Работа со стеком матриц

```
draw_wheel_and_bolts()
{
  int i;

  draw_wheel();
  for(i=0;i<5;i++){
    glPushMatrix();
    glRotatef(72.0*i, 0.0, 0.0, 1.0);
    glTranslatef(3.0, 0.0, 0.0);
    draw_bolt();
    glPopMatrix();
  }
}

draw_body_and_wheel_and_bolts()
{
  draw_car_body();
  glPushMatrix();
    glTranslatef(40, 0, 30);     /* переходим к первому колесу */
    draw_wheel_and_bolts();
  glPopMatrix();
  glPushMatrix();
    glTranslatef(40, 0, -30);    /* переходим ко второму колесу */
    draw_wheel_and_bolts();
  glPopMatrix();
  ...                            /* так же рисуем последние два колеса */
}
```

В этом примере колеса и болты ориентированы вдоль оси *z*, болты расположены через 72 и на расстоянии трех единиц (это могут быть дюймы) от центра колеса. Передние колеса вынесены на 40 единиц вперед и на 30 единиц вправо и влево от начала координат автомобиля.

Стек более эффективен и удобен, чем одиночные матрицы, особенно когда он поддерживается оборудованием. При помещении матрицы в стек не нужно

копировать текущие данные для возврата в основной процесс, а оборудование может одновременно копировать более одного элемента матрицы. Иногда разумно хранить на вершине стека единичную матрицу, благодаря чему отпадает потребность постоянно вызывать функцию `glLoadIdentity()`.

Стек модельно-видовых матриц

Как вы уже видели ранее, в разделе «Модельно-видовые преобразования», модельно-видовая матрица содержит в себе данные от перемножения видовой и модельной матриц. Каждое видовое или модельное преобразование создает новую матрицу, которая перемножается с текущей модельно-видовой матрицей. Результат, который становится новой текущей матрицей, представляет собой композицию преобразований. Модельно-видовой стек содержит 32 матрицы размером 4 × 4. Изначально верхняя матрица — единичная. Некоторые реализации OpenGL позволяют хранить в стеке более 32 матриц. Чтобы узнать, сколько матриц может уместиться в стеке, используйте команду `glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, Glint *params)`.

Стек матриц проекций

По матрице проекций выполняется преобразование проецирования, описывающее видимый объем. Обычно матрицы проекций не объединяют, поэтому перед выполнением преобразований применяют функцию `glLoadIdentity()`. Также по этой причине глубина стека матриц проекций не превышает двух матриц. Некоторые реализации OpenGL предоставляют возможность хранения в стеке более двух матриц размером 4 × 4. Для определения глубины стека используется функция `glGetIntegerv(GL_MAX_PROJECTION_STACK_DEPTH, Glint *params)`.

Одним из применений второй матрицы в стеке являются приложения, в которых нужно отображать справочное окно с текстом дополнительно к обычному окну с трехмерной сценой. Так как легче всего отображать текст в ортогональной проекции, можно сохранить текущую проекцию, установить ортогональную проекцию, отобразить текст и вернуться к исходной точке:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();                    /* сохраняем текущую проекцию*/
    glLoadIdentity();
    glOrtho(...);                  /* устанавливаем для отображения справки */
    display_the_help();
glPopMatrix();
```

Заметьте, что мы также не меняем модельно-видовую матрицу.

ПРИМЕЧАНИЕ

Если вы немного знаете математику, вы можете создать собственную матрицу проекций, которая позволит выполнить нестандартное преобразование. Так, ни в OpenGL, ни в GLU не встроен механизм для создания угловой перспективы. Если вы собираетесь эмулировать рисование чертежей, такая матрица вам пригодится.

Дополнительные плоскости отсечения

Кроме шести плоскостей ограничения объема видимости (левой, правой, нижней, верхней, ближней и дальней) можно задать дополнительные плоскости отсечения (как показано на рис. 3.22). Это позволяет довольно просто убирать посторонние объекты из сцены — например, когда требуется отобразить разрез.

Каждая плоскость определяется коэффициентами ее уравнения: $Ax + By + Cz + D = 0$. Плоскости отсечения автоматически преобразуются соответственно модельно-видовым преобразованиям. Результирующий объем есть пересечение видимого объема и всех полупространств, определенных дополнительными плоскостями отсечения. Напомним, что у многоугольников, подлежащих отсечению, OpenGL автоматически пересчитывает вершины.

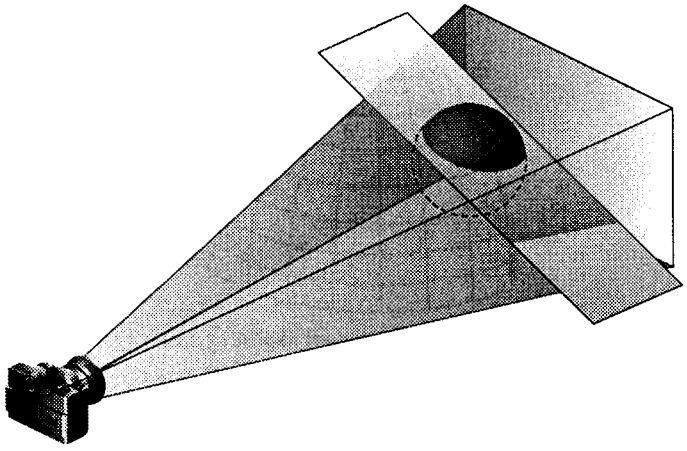


Рис. 3.22. Дополнительные плоскости отсечения и объем видимости

void `glClipPlane`(GLenum *plane*, const GLdouble **equation*);

Определяет плоскость отсечения. Указатель *equation* адресует точки для четырех коэффициентов уравнения плоскости, $Ax + By + Cz + D = 0$. Все точки, заданные в координатах глаза (x_e, y_e, z_e, w_e), для которых выполняется выражение $(A\ B\ C\ D)\mathbf{M}^{-1}(x_e, y_e, z_e, w_e)^T \geq 0$, лежат в полупространстве, определенном плоскостью отсечения, где $\mathbf{M}$ — текущая модельно-видовая матрица. Все точки, не принадлежащие полупространству, определенному плоскостью отсечения, отсекаются. Аргумент *plane* принимает значения `GL_CLIP_PLANE`*i*, где *i* — целое число, обозначающее плоскость отсечения, в диапазоне от 0 до максимального числа плоскостей отсечения минус один.

Каждую плоскость отсечения предварительно нужно определить:

```
glEnable(GL_CLIP_PLANEi);
```

Отключение плоскости отсечения выполняется командой

```
glDisable(GL_CLIP_PLANEi);
```

Все реализации OpenGL поддерживают как минимум шесть дополнительных плоскостей отсечения, однако в некоторых реализациях их может быть больше. Чтобы узнать действительное количество, используйте функцию `glGetIntegerv()` со значением аргумента `GL_MAX_CLIP_PLANES`.

ПРИМЕЧАНИЕ

Отсечение выполняется в координатах глаза, а не в координатах отсечения. Это различие заметно, если матрица проекций — вырожденная. (То есть матрица проекций по причине нулевого детерминанта переводит трехмерные координаты в двухмерные.)

Пример отсечения

Программа, приведенная в листинге 3.5, рисует каркас сферы с двумя плоскостями отсечения, которые отрезают три четверти сферы, как показано на рис. 3.23.

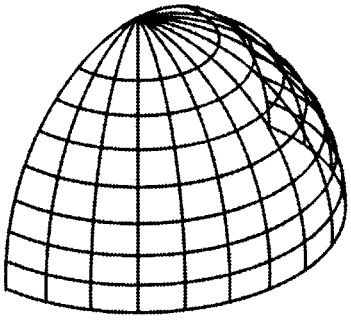


Рис. 3.23. Усеченный каркас сферы

Листинг 3.5. Каркас сферы с двумя плоскостями отсечения: clip.c

```
void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
}

void display(void)
{
    GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};
    GLdouble eqn2[4] = {1.0, 0.0, 0.0, 0.0};

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix();
    glTranslatef(0.0, 0.0, -5.0);

/*   отсекаем нижнюю половину — y < 0   */
    glClipPlane(GL_CLIP_PLANE0, eqn);
    glEnable(GL_CLIP_PLANE0);

/*   отсекаем левую половину — x < 0    */
    glClipPlane(GL_CLIP_PLANE1, eqn2);
    glEnable(GL_CLIP_PLANE1);
```

```
    glRotatef(90.0, 1.0, 0.0, 0.0);
    glutWireSphere(1.0, 20, 16);
    glPopMatrix();
    glFlush();
}
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode (GL_MODELVIEW);
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

Упражнение

- Попробуйте изменить коэффициенты плоскостей отсечения в листинге 3.5.
- Попробуйте осуществить модельно-видовое преобразование, например `glRotate*()`, для изучения функции `glClipPlane()`. Сделайте плоскости отсечения независимыми от объектов в сцене.

Примеры объединения нескольких преобразований

Этот раздел демонстрирует, как комбинировать несколько преобразований для получения требуемого результата. Он содержит два примера: солнечная система, объекты которой нужно вращать вокруг их осей и по их орбитам, и рука робота, сочлененная в нескольких местах, при движении которой преобразуются связанные с суставами системы координат.

Солнечная система

Программа, представленная в этом разделе, рисует простую Солнечную систему, состоящую из Солнца и планеты, при помощи команд рисования сфер. Вращение солнца и планеты вокруг собственных осей реализовано командой `glRotate*()`.

Для перемещения планеты по орбите применена команда `glTranslate*()`. Требуемые размеры сфер задаются аргументами функции `glutWireSphere()`.

Для рисования Солнечной системы сначала нужно определить модельно-видовые преобразования и преобразования проецирования. В данном примере используются команды `gluPerspective()` и `gluLookAt()`.

Рисование Солнца тривиально, поскольку Солнце неподвижно относительно начала фиксированной системы координат и перемещение не требуется. Для остального, то есть вращения Солнца вокруг оси, достаточно `glRotate*()`. Рисование планеты, вращающейся вокруг Солнца, как показано на рис. 3.24, требует нескольких модельных преобразований. Планета оборачивается вокруг своей оси за один день, а за год она совершает полный оборот вокруг Солнца.

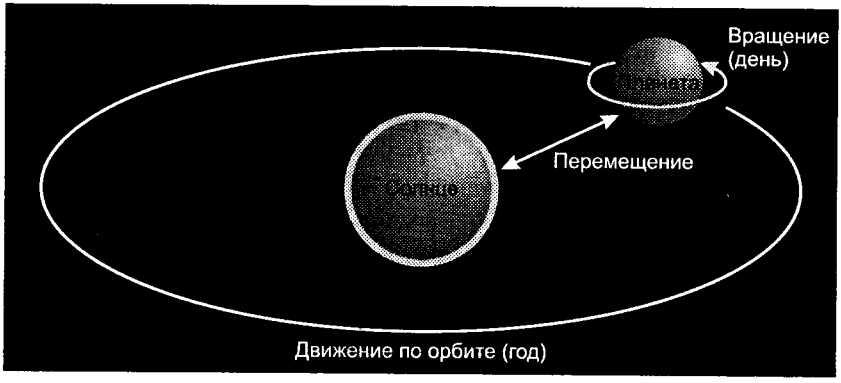


Рис. 3.24. Солнце и планета

Для определения порядка модельных преобразований представим, что происходит с локальной системой координат. Функция `glRotate*()` вращает локальную систему координат, которая изначально совпадала с фиксированной системой координат. Затем функция `glTranslate*()` перемещает локальную систему координат в точку на орбите планеты. Расстояние переноса равно радиусу орбиты. Таким образом, функция `glRotate*()` определяет положение планеты на орбите (текущее время года).

Затем вторая функция `glRotate*()` поворачивает локальную систему координат вокруг локальной оси, тем самым задавая время суток на планете. Когда все команды преобразований выполнены, рисуется планета.

Итак, имитация Солнечной системы реализуется следующей последовательностью команд (полностью программа приведена в листинге 3.6):

```
glPushMatrix();
glutWireSphere(1.0, 20, 16);          /* рисуем солнце */
glRotatef((GLfloat) year, 0.0, 1.0, 0.0);
glTranslatef(2.0, 0.0, 0.0);
glRotatef((GLfloat) day, 0.0, 1.0, 0.0);
glutWireSphere(0.2, 10, 8);           /* рисуем планету */
glPopMatrix();
```

Листинг 3.6. Планетарная система: planet.c

```
static  int year = 0, day = 0;

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);

   glPushMatrix();
   glutWireSphere(1.0, 20, 16);   /* рисуем солнце */
   glRotatef((GLfloat) year, 0.0, 1.0, 0.0);
   glTranslatef(2.0, 0.0, 0.0);
   glRotatef((GLfloat) day, 0.0, 1.0, 0.0);
   glutWireSphere(0.2, 10, 8);    /* рисуем планету */
   glPopMatrix();
   glutSwapBuffers();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW); glLoadIdentity();
   gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 'd':
         day = (day +10) % 3 60;
         glutPostRedisplay();
         break;
      case 'D':
         day = (day   – 10) % 360;
         glutPostRedisplay();
         break;
      case 'y':
         year = (year +5) % 3 60;
         glutPostRedisplay();
         break;
      case 'Y':
         year = (year   – 5) % 3 60;
         glutPostRedisplay();
         break;
```

продолжение ⇙

Листинг 3.6 (*продолжение*)

```
        default:
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Упражнение

- Попробуйте добавить луну к планете или несколько лун и планет. Подсказка: используйте функции `glPushMatrix()` и `glPopMatrix()` для хранения и восстановления положения и ориентации системы координат в требуемый момент времени. Если вы решили обзавестись несколькими спутниками, следует сохранять систему координат до позиционирования каждой луны и восстанавливать ее после рисования каждой луны.
- Попробуйте наклонить ось планеты.

Сочлененная рука робота

Этот раздел содержит программу, которая создает сочлененную руку робота из двух или более сегментов. Рука должна быть подвижна в плече, локте или других суставах. На рис. 3.25 приведена рука с одним подвижным суставом.

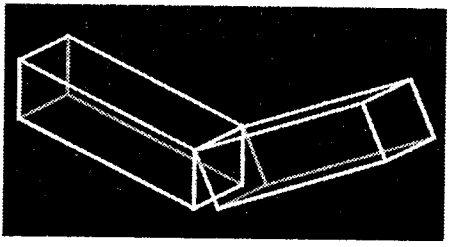


Рис. 3.25. Рука робота

Для рисования сегментов руки подходит масштабированный (вытянутый) куб, но сначала вам надо применить модельные преобразования для ориентирования каждого сегмента. Так как центр локальной системы координат изначально нахо-

дится в центре куба, нужно переместить локальную систему координат к одной из границ куба.

С другой стороны, хотя физически сустав двигается относительно шарнира, логически куб вращается вокруг своего центра. Поэтому после вызова функции `glTranslate*()` для указания точки вращения и функции `glRotate*()` для вращения куба мы возвращаемся к центру куба. Затем, перед рисованием, куб масштабируется (сплющивается или растягивается). Функции `glPushMatrix()` и `glPopMatrix()` ограничивают применение вносимого функцией `glScale*()` эффекта. Код рисования первого сегмента руки будет выглядеть примерно так (полная программа приведена в листинге 3.7):

```
glTranslatef(-1.0, 0.0, 0.0);
glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef(1.0, 0.0, 0.0);
glPushMatrix();
glScalef(2.0, 0.4, 1.0);
glutWireCube(1.0);
glPopMatrix();
```

Поскольку система координат раз была повернута, ось x уже ориентирована вдоль длины руки. Таким образом, для рисования следующего сустава необходим перенос вдоль оси x локальной системы координат в следующую точку вращения. В этой точке вращения опять можно использовать приведенный выше код. Данный процесс можно продолжить, переходя от сустава к суставу (плечо, локоть, запястье, пальцы):

```
glTranslatef(1.0, 0.0, 0.0);
glRotatef((GLfloat) elbow, 0.0, 0.0, 1.0);
glTranslatef(1.0, 0.0, 0.0);
glPushMatrix();
glScalef(2.0, 0.4, 1.0);
glutWireCube(1.0);
glPopMatrix();
```

Листинг 3.7. Рука робота: robot.c

```
static int shoulder = 0, elbow = 0;

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glTranslatef(-1.0, 0.0, 0.0);
   glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);
   glTranslatef(1.0, 0.0, 0.0);
   glPushMatrix();
   glScalef(2.0, 0.4, 1.0);
```

продолжение ⇩

Листинг 3.7 (*продолжение*)

```
    glutWireCube(1.0);
    glPopMatrix();


    glTranslatef(1.0, 0.0, 0.0);
    glRotatef((GLfloat) elbow, 0.0, 0.0, 1.0);
    glTranslatef(1.0, 0.0, 0.0);
    glPushMatrix();
    glScalef(2.0, 0.4, 1.0);
    glutWireCube(1.0);
    glPopMatrix();

    glPopMatrix();
    glutSwapBuffers();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(65.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0)
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -5.0);
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 's':   /*  клавиша s вращает плечо */
            shoulder = (shoulder + 5) % 360;
            glutPostRedisplay();
            break;
        case 'S':
            shoulder = (shoulder - 5) % 360;
            glutPostRedisplay();
            break;
        case 'e':  /* клавиша e вращает локоть */
            elbow = (elbow + 5) % 360;
            glutPostRedisplay();
            break;
        case 'E':
            elbow = (elbow - 5) % 360;
            glutPostRedisplay();
            break;
        default:
            break;
    }
}

int main(int argc, char** argv)
{
```

```
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Упражнение

- Измените листинг 3.7, добавив дополнительные сегменты к руке.
- Дополните руку сегментами в соответствующих местах, например пальцами на запястье, как показано на рис. 3.26. Подсказка: используйте функции `glPushMatrix()` и `glPopMatrix()` для сохранения и восстановления положения и ориентации системы координат в кисти. Текущая матрица сохраняется перед позиционированием каждого пальца и восстанавливается после его рисования.

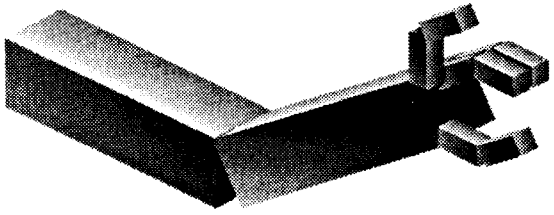


Рис. 3.26. Рука робота с пальцами

Пособия по преобразованиям от Нэйта Робинса

Если у вас есть набор программ от Нэйта Робинса, вернитесь к программе преобразований и запустите ее снова. Задействуйте раскрывающееся меню для изменения последовательности операций `glRotate*()` и `glTranslate()` и посмотрите на результат этих изменений.

Обратные, или имитационные, преобразования

Конвейер геометрической обработки очень удобен для работы с видовыми матрицами и матрицами проекций. В процессе преобразования мировые координаты вершин переводятся в оконные (экранные) координаты. Однако есть ситуации, в которых нужно повернуть этот процесс вспять. Например, когда приложение ориентируется в трехмерном пространстве с помощью мыши. Мышь возвращает положение указателя на плоскости; соответственно, приложению нужно как-то узнать трехмерные координаты на основе двухмерных.

Обратные преобразования выполняют функции GLU `gluUnProject()` и `gluUnProject4()`. На основе трехмерных оконных координат преобразованных вершин и всех преобразований, которые с ними происходили, функция `gluUnProject()` возвращает начальные мировые координаты вершин. Функция `gluUnProject4()` используется, если глубина не укладывается в диапазон по умолчанию [0; 1].

int `gluUnProject`(GLdouble *winx*, GLdouble *winy*, GLdouble *winz*, const GLdouble *modelMatrix*[16], const GLdouble *projMatrix*[16], const GLint *viewport*[4], GLdouble **objx*, GLdouble **objy*, GLdouble **objz*);

Переводит оконные координаты (*winx*, *winy*, *winz*) в координаты объекта, опираясь на преобразования, определенные модельно-видовой матрицей (аргумент *modelMatrix*), матрицей проекций (*projMatrix*) и окном просмотра (*viewport*). Координаты объекта возвращаются через указатели *objx*, *objy*, *objz*. Функция возвращает `GL_TRUE` в случае успешного завершения и `GL_FALSE` в случае ошибки (если матрица необратима). Эта операция не пытается отсекать координаты окна просмотра и равнодушна к координатам глубины, если они выходят за значения, заданные функцией `glDepthRange()`.

Существует ряд сложностей при попытке инвертирования процесса преобразования. Двухмерное положение на экране может отсчитываться в трехмерном пространстве от любого места на некоторой линии. Для устранения неоднозначности результата функция `gluUnProject()` требует, чтобы оконная координата глубины (*winz*) была представлена в терминах функции `glDepthRange()`. (Для получения более подробной информации о диапазоне глубины см. раздел «Измененная координата глубины».) По умолчанию для функции `glDepthRange()` значение *winz*, равное 0.0, требует мировые координаты преобразованной точки на ближней плоскости отсечения, в то время как *winz*, равное 1.0, соответствует точкам, расположенным на дальней плоскости отсечения.

Листинг 3.8 демонстрирует пример использования функции `gluUnProject()` с целью чтения координат мыши и определения трехмерной точки на ближней и дальней плоскостях отсечения. Вычисленные мировые координаты выводятся в стандартный поток вывода, но окно остается черным.

Листинг 3.8. Обратное преобразование: unproject.c

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
void mouse(int button, int state, int x, int y)
{
   GLint viewport[4];
   GLdouble mvmatrix[16], projmatrix[16];
   GLint realy;  /* OpenGL-координата y */
   GLdouble wx, wy, wz; /* возвращенные мировые x-, y-, z-координаты */

   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN) {
            glGetIntegerv(GL_VIEWPORT, viewport);
            glGetDoublev(GL_MODELVIEW_MATRIX, mvmatrix);
            glGetDoublev(GL_PROJECTION_MATRIX, projmatrix);
/* viewport[3] - высота окна в пикселах  */
            realy = viewport[3] - (GLint) y - 1;
            printf ("Координаты курсора равны (%4d, %4d)\n", x, realy);
            gluUnProject((GLdouble) x, (GLdouble) realy, 0.0,
               mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
            printf("Мировые координаты при z = 0,0 равны (%f, %f,
               %f)\n", wx, wy, wz);
            gluUnProject((GLdouble) x, (GLdouble) realy, 1.0,
               mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
            printf("Мировые координаты при z = 1,0 равны (%f, %f,
               %f)\n", wx, wy, wz);
         }
         break;
      case GLUT_RIGHT_BUTTON:
         if (state == GLUT_DOWN)
            exit(0);
         break;
      default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

В GLU 1.3 представлена измененная версия `gluUnProject()`. Команда `gluUnProject4()` может работать с нестандартным диапазоном `glDepthRange()` и со значением координаты w, отличным от 1.

int `gluUnProject4`(GLdouble *winx*, GLdouble *winy*, GLdouble *winz*, GLdouble *clipw*, const GLdouble *modelMatrix*[16], const GLdouble *projMatrix*[16], const GLint *viewport*[4], GLclampd *zNear*, GLclampd *zFar*, GLdouble **objx*, GLdouble **objy*, GLdouble **objz*, GLdouble **objw*);

Выполняет действия, аналогичные функции `gluUnProject()`. Переводит оконные координаты (*winx*, *winy*, *winz*) в координаты объекта, опираясь на преобразования, определенные модельно-видовой матрицей (аргумент *modelMatrix*), матрицей проекций (*projMatrix*), окном просмотра (*viewport*) и диапазоном глубины от *zNear* до *zFar*. Координаты объекта возвращаются через указатели *objx*, *objy*, *objz*.

Функция `gluProject()` из библиотеки GLU связана с функцией `gluUnProject()`. Функция `gluProject()` имитирует действия конвейера преобразований. Получая трехмерные мировые координаты и все преобразования над ними, функция `gluProject()` возвращает преобразованные оконные координаты.

int `gluProject`(GLdouble *objx*, GLdouble *objy*, GLdouble *objz*, const GLdouble *modelMatrix[16]*, const GLdouble *projMatrix[16]*, const GLint *viewport[4]*, GLdouble **winx*, GLdouble **winy*, GLdouble **winz*);

Переводит объектные координаты (*objx*, *objy*, *objz*) в оконные координаты, используя преобразования, определенные модельно-видовой матрицей (*modelMatrix*), матрицей проекций (*projMatrix*) и окном просмотра (*viewport*). Координаты объекта возвращаются через указатели *winx*, *winy*, *winz*. Функция возвращает значение `GL_TRUE` в случае успешного завершения и `GL_FALSE` — в случае ошибки.

ВНИМАНИЕ

Матрицы, обрабатываемые функциями gluUnProject(), gluUnProject4() и gluProject(), — стандартные развернутые по столбцам матрицы OpenGL. Вы можете использовать команды glGetDoublev() и glGetIntegerv() для получения текущих матриц GL_MODELVIEWPMATRIX, GL_PROJECTION_MATRIX и GL_VIEWPORT.

4

Цвет

После прочтения этой главы вы сможете:

- Выбирать и использовать в приложении RGBA-цвет или цветовой индекс
- Указывать требуемые цвета для рисования объектов
- Рисовать многоугольник, закрашенный более чем одним цветом, с плавными переходами цвета

Цель большинства приложений OpenGL — нарисовать цветное изображение в окне на экране. Окно — это прямоугольный массив пикселов, отображенных собственным цветом. Поэтому, в некотором смысле, цель всех вычислений, проводимых OpenGL, — определение конечного цвета каждого пиксела, рисуемого на экране. В следующих основных разделах этой главы описываются команды задания цвета и то, как OpenGL их интерпретирует:

- **Цветовосприятие.** Описывает, как глаз воспринимает цвета.
- **Вычисление цвета.** Рассматривает зависимости между пикселами на экране монитора и их цветом. Также в нем обозреваются модель RGBA и индексированные цвета.
- **RGBA в сравнении с индексированным цветом.** Рассказывает о том, как графическое оборудование поддерживает эти два режима и как выбрать нужный вам режим.
- **Задание цветовой и полутоновой модели.** Описывает команды OpenGL, используемые для определения требуемой цветовой или полутоновой модели.

Цветовосприятие

С точки зрения физики свет состоит из фотонов — мельчайших частиц света, каждая из которых перемещается по своему собственному пути и имеет индивидуальную частоту колебаний (или длину волны, или энергию — частота, длина волны, энергия зависят друг от друга). Фотон характеризуется положением, направлением и частотой/длиной волны/энергией. Фотоны, длина волны которых лежит в пределах от около 390 нм (фиолетовый цвет) до 720 нм (красный цвет), соответствуют цветам видимого спектра, они формируют цветовую радугу (фиолетовый, синий, голубой, зеленый, желтый, оранжевый, красный). Однако глаз различает множество цветов, не включенных в радугу, — например, белый, черный, коричневый и розовый. Как такое может быть?

То, что видит глаз, — смесь фотонов различных частот. Реальные источники света характеризуются частотой фотонов, которые они испускают. Идеальный белый свет состоит из равного количества фотонов всех частот. Луч лазера обычно беспримесный, и все фотоны имеют одинаковую частоту (а также направление и фазу). В свете от натриевой лампы больше желтой составляющей. Свет большинства звезд в космосе имеет распределение, зависящее преимущественно от их температуры (излучение абсолютно черных тел). Распределение частот от нескольких источников сложнее.

Человеческий глаз воспринимает цвет, когда стимулируются ячейки сетчатки, называемые зрительными клетками или колбочками, после удара фотонами. Существует три вида колбочек, реагирующих на три различные длины волны света: один вид реагирует на красный свет, другой — на зеленый и третий — на синий. (У людей, страдающих дальтонизмом, обычно отсутствуют колбочки одного из видов.) Когда смесь фотонов попадает в глаз, колбочки на сетчатке испытывают различную степень возбуждения, в зависимости от их вида.

Так как каждый цвет воспринимается глазом как уровень возбуждения колбочек фотонами, глаз различает цвета, отличные от спектрального разложения белого цвета призмой или радугой. Например, если послать смесь красных и синих фотонов, в которой фотоны обоих видов представлены одинаково, глаз увидит фуксиновый цвет маджента, отсутствующий в радуге[1]. Другие комбинации дают коричневый, бирюзовый и розовато-лиловый, ни один из которых не представлен в спектре.

Монитор эмулирует видимые цвета за счет свечения пикселов, комбинирующего красный, зеленый и синий цвета в пропорциях, стимулирующих колбочки сетчатки в той мере, чтобы получился желательный цвет. Если бы у человека было больше видов колбочек, например некоторые были бы чувствительны к желтому цвету, то тогда цветные мониторы имели бы желтую пушку и графическое оборудование использовало бы модель RGBY (red, green, blue, yellow — красный, зеленый, синий, желтый). А если бы мы все являлись одинаковыми дальтониками, эта глава была бы намного короче.

Для отображения нужного цвета монитор испускает нужную часть красного, зеленого и синего света (RGB), чтобы в глазу стимулировались колбочки различных видов. Цветной монитор может объединять красный, зеленый и синий цвета в различных пропорциях, формируя миллионы оттенков, различаемых глазом.

ПРИМЕЧАНИЕ

Существует множество других представлений цвета (цветовых моделей), например HLS, HSV и CMYK. Если они когда-нибудь вам понадобятся, то понадобится и преобразовать цвета из/в RGB. Формулы для преобразования описаны в книге: Foley, van Dam, et. al., Computer Graphics: Principles and Practice (Addison–Wesley, 1990).

Сказанное верно для восприятия глазом комбинации фотонов направленного света. Ситуация для света, отраженного от материалов и попавшего в глаз, более сложная — например, белые лучи, отражаясь от красного мячика, становятся красными, а желтые, освещающие синий стакан, кажутся черными. (См. раздел «Реальный мир и освещение в OpenGL» главы 5, где рассматривается этот эффект.)

Вычисление цвета

На цветном экране компьютера оборудование заставляет каждый пиксел экрана излучать красный, зеленый и синий свет различной интенсивности. Эти зна-

[1] В компьютерной литературе последних лет повелось называть эти цвета «голубым» и «пурпурным». Поскольку компьютерная графика была компьютерной графикой еще во времена CGA-мониторов, соответствующие названия цветов «циан» и «маджента» вошли в обиход в то же время. Но RGB-значения маджента (0xFF00FF) вовсе не то же, что RGB-значение традиционного пурпурного цвета (0x800080), и аналогично для циана и голубого, поэтому и в данном переводе фигурируют привычные для «компьютерщиков» термины. Если в плане визуального восприятия пурпурный и маджента еще более-менее похожи, то циан намного ближе к аквамарину, чем к голубому цвету. — *Примеч. перев.*

чения интенсивности обозначаются R, G и B. Часто они упаковываются вместе (иногда с четвертым значением A, называемым альфа-компонентом), а упакованное значение обозначается буквами RGB (или RGBA). (См. раздел «Смешивание» главы 6, где смысл альфа-составляющей объяснен подробно.) Информация о цвете каждого пиксела может храниться в *режиме RGBA*, когда определяются значения R, G, B и, возможно, A, или в режиме *индексированных цветов*, в котором единственное число, называемое *индексом цвета*, характеризует цвет каждого пиксела. Каждый индекс цвета указывает на запись в таблице, где определены значения R, G и B для каждого индекса. Эта таблица называется *цветовой палитрой*.

В режиме индексации цвета можно изменять значения цветовой карты (палитры). Но так как цветовые карты управляются оконной системой, команд OpenGL для этих целей не предусмотрено. Все примеры в данной книге инициализируют режим отображения цвета во время открытия окна с помощью команд библиотеки GLUT (см. приложение Г для получения дополнительной информации).

Количество пикселов в массиве и число цветов, которые в состоянии отобразить каждый пиксел, зависит от используемого графического оборудования. В любой графической системе каждому пикселу соответствует область памяти для хранения цвета. А вся эта память называется *цветовым буфером*. Размер буфера, зависящий от компьютера, обычно измеряется в битах, и восьмибитный буфер может хранить 8 бит данных (256 возможных различных цветов) для каждого пиксела. (См. главу 10 для получения более подробной информации.)

Значения R, G и B могут изменяться от 0,0 (нет) до 1,0 (максимальная интенсивность). Например, сочетание R = 0,0, G = 0,0 и B = 1,0 представляет максимально возможно яркий синий цвет. Если R, G и B равны 0,0, пиксел имеет черный цвет; если они все равны 1,0, пиксел рисуется самым ярким белым цветом, который только может быть отображен на экране. *Смешивание* зеленого и синего даст циановый оттенок. Комбинация синего и красного — маджента. Красный и зеленый дадут желтый. Для того чтобы вам было легче получать нужный цвет из R-, G- и B-составляющих, посмотрите на цветовой куб на цветном рисунке 12. По осям куба отложена интенсивность красной, синей и зеленой составляющих. Черно-белый вариант куба показан на рис. 4.1.

Для задания цвета объекта (в данном случае точки) могут использоваться команды, приведенные ниже:

```
glColor3f(1.0, 0.0, 0.0);      /* текущий RGB-цвет — красный: */
                               /* чистый красный, нет зеленого и синего */
glBegin(GL_POINTS);
   glVertex3fv(point_array);
glEnd();
```

В некоторых режимах (например, если включено вычисление освещенности или текстур) заданный цвет может изменяться другими операциями перед тем, как будет записан в видеобуфер в качестве цвета пиксела. Фактически цвет пиксела определяется в результате последовательности операций.

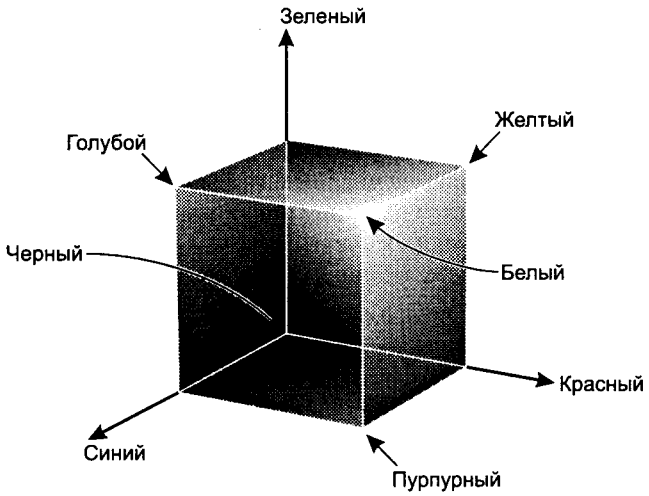


Рис. 4.1. Цветовой куб в черно-белом варианте

В начале выполнения программы режим отображения цвета задается как RGBA или режим индексации цвета. Режим, установленный один раз, не может быть изменен. После запуска программы цвет (индексы цветов или значения RGBA) определяется для каждой вершины каждого геометрического примитива. Этот цвет, если включено освещение, определен из взаимодействия матрицы преобразований и нормалей к поверхностям и остальных свойств материала. Другими словами, красный мяч с синим световым пятном отличается от красного мяча без освещения. (См. главу 5 для получения дополнительной информации.) После вычисления освещения применяется выбранная модель закраски. Как описано в разделе «Задание цветовой и полутоновой моделей», можно выбрать сплошную закраску или с плавными переходами, по-разному влияющую на окончательный цвет пикселов.

Затем производится *растеризация* примитива — его преобразование в двухмерное точечное изображение. При растеризации определяется, какие квадраты целочисленной сетки в оконных координатах покрыты примитивом, а затем назначаются цвет и другие значения для каждого квадрата. Квадрат сетки вместе со связанными с ним значениями цвета, *z* (глубины) и координат текстур называется *фрагментом*. Пикселы — это элементы видеобуфера (буфера кадров), а фрагменты связаны с примитивами, и их комбинируют с пикселами для получения нового цвета пиксела. После фрагментации выполняются текстурирование, наложение тумана и размытие (если они включены). Затем фрагменты проходят через операции альфа-смешивания, полутонового смешивания, побитовые логические операции и др. Наконец значения цветов фрагментов (индексы цвета или RGBA) записываются в пикселы и отображаются на экране.

RGBA в сравнении с индексированным цветом

Вне зависимости от режима для каждого пиксела хранится необходимое количество данных о цвете, определяемое числом битовых плоскостей в буфере кадров. В *битовой плоскости* выделяется один бит информации для каждого пиксела. Если существует восемь битовых плоскостей, для каждого пискела определены восемь цветовых битов и, соответственно, $2^8 = 256$ различных значений или цветов.

R-, G- и B-компоненты в битовых плоскостях часто хранятся раздельно (то есть, если определены 24 битовые плоскости, система выделяет 8 бит для каждого компонента — красного, зеленого и синего), но это не всегда так. Чтобы узнать, сколько битовых плоскостей доступно в вашей системе, вызовите функцию `glGetIntegerv()` с одним из значений аргумента `GL_RED_BITS`, `GL_GREEN_BITS`, `GL_BLUE_BITS`, `GL_ALPHA_BITS` и `GL_INDEX_BITS`.

ПРИМЕЧАНИЕ

Интенсивность цвета на большинстве компьютеров не воспринимается человеческим глазом линейно. Пусть цвет просто состоит только из красного компонента, а зеленый и синий равны нулю. При изменении интенсивности от 0.0 (отключена) до 1.0 (полная) число электронов, атакующих пиксел, увеличивается, но будет ли значение 0,5 давать среднюю интенсивность между 0.0 и 1.0? Для того чтобы проверить это, напишите программу, которая рисует пикселы в шахматном порядке с интенсивностью 0.0 и 1.0, и сравните ее с областью, залитой с интенсивностью 0.5. На определенном расстоянии от экрана две области должны выглядеть одинаково. Если они все-таки различаются, нужно с помощью механизма, реализованного в вашей системе, провести коррекцию. Например, на многих системах есть таблица для установки интенсивности 0.5, средней между 0.0 и 1.0. При отображении обычно имеется экспоненциальная зависимость, где экспонента связана с гаммой (отсюда термин — гамма-коррекция). Использование одинаковой гаммы для красного, зеленого и синего компонентов обычно дает хорошие результаты, но подстройка этих трех значений по отдельности еще более улучшает восприятие.[1] (Для получения более подробной информации по этой теме см. книгу: Foley, van Dam, et. al. Computer Graphics: Principles and Practice (Addison–Wesley, 1990).)

Режим RGBA

В режиме RGBA оборудование выделяет определенное число битовых плоскостей для каждого из R-, G-, B- и A-компонентов (не обязательно одинаковое число для каждого компонента), как показано на рис. 4.2. Значения R, G и B чаще хранятся в виде целых чисел, чем в виде вещественных чисел, и они подгоняются под доступное число битов для хранения и поиска. Например, если в системе восемь битов выделено для R-компонента, то значениями могут быть целые числа

[1] Интенсивность свечения люминофора в электронно-лучевых трубках нелинейно зависит от числа электронов в луче: $I = \alpha \times N^{\lambda}$, где α и λ — константы. Во избежание искажений применяется гамма-коррекция входного сигнала. Для этого входное значение возводится в степень $1/\lambda$ и затем передается монитору. Значения гаммы для различных мониторов могут быть различны для красного, зеленого и синего каналов. Поэтому чтобы изображения, полученные на другом мониторе, воспроизводились правильно, делается подбор гаммы монитора после ее оценки с помощью калибровочных тестовых изображений. — *Примеч. перев.*

от 0 до 256; то есть числа 0, 1, 2, ..., 255 в битовых плоскостях могут соответствовать значениям R 0/255 = 0.0, 1/255, 2/255,..., 255/255 = 1.0. Сколько бы ни было битовых плоскостей, 0.0 всегда определяет минимальную интенсивность, а 1.0 — максимальную.

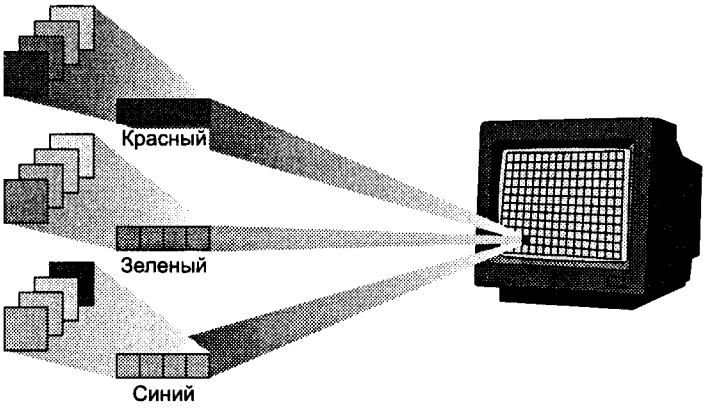


Рис. 4.2. Значения RGB в битовых плоскостях

ПРИМЕЧАНИЕ

Значение альфа-компонента (A в RGBA) непосредственно не влияет на цвет, видимый на экране, поскольку цветом он не является, а оказывает воздействие на значения RGB-составляющих. Оно может использоваться для получения различных эффектов, включая смешивание и полупрозрачность. (См. раздел «Смешивание» главы 6 для получения более подробной информации о значении альфа-компонента.)

Число цветов, в которые может быть окрашен пиксел, зависит от числа битовых плоскостей и возможностей оборудования по обработке этих битовых плоскостей. Таким образом, компьютер с 24 битовыми плоскостями в режиме RGB способен отобразить 16,77 млн различных оттенков цвета.

Передача полутонов

В некотором графическом оборудовании для увеличения числа видимых цветов применяется так называемое размытие (dithering). Размытие — это технология комбинирования каких-то цветов для имитирования другого цвета. Предположим, что в системе для каждого RGB-компонента цвета выделен только один бит, то есть она в состоянии отобразить только восемь цветов: черный, белый, красный, синий, зеленый, желтый, циан и мадженту. Для отображения розовой области оборудование может закрасить область красными и белыми пикселами в шахматном порядке. Область кажется розовой — средней между белой и красной, — когда глаз находится далеко от экрана и не различает отдельные пикселы. Красно-розовый образуется за счет увеличения количества красных пикселов, а бледно-розовый — увеличением числа белых.

Здесь в действительности розовых пикселов нет, и для такого оборудования создание области, состоящей из множества пикселов, не различимых глазом по

отдельности, — единственный выход. При задании RGB-значения неподдерживаемого напрямую цвета оборудование подсвечивает пикселы многоугольника, усредняя ближайшие цвета, которые позволят глазу увидеть тот оттенок, который нужен. (Заметьте, что при считывании информации из буфера кадров вы будете иметь дело с красным и белым цветами, а не с розовым. См. главу 8.)

Рисунок 4.3 демонстрирует простое смешивание черных и белых пикселов для передачи серого цвета. Здесь шаблоны 4 × 4 верхнего ряда в порядке слева направо соответствуют смешиванию с интенсивностью серого 50%, 19% и 69%. Под каждым шаблоном показан результат его применения. На рисунке черные и белые квадраты больше пикселов по размерам, но если посмотреть на это изображение с другого конца комнаты, можно увидеть нужные градации серого.

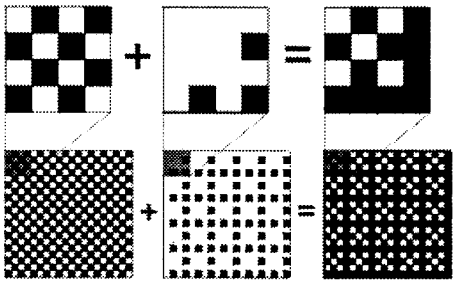


Рис. 4.3. Смешивание черного и белого для получения серого

Для получения высококачественного изображения в отсутствие смешивания достаточно выделить каждой R-, G- и B-составляющей по восемь битов информации. Однако, даже если компьютер имеет 24 битовые плоскости, это не значит, что от смешивания нужно отказаться. Например, при работе в режиме двойной буферизации число битовых плоскостей может быть удвоено посредством их деления на 12-битовые, то есть реально на каждый компонент будет выделено по 4 бита. Без полутонового смешивания четырехбитовые цветовые компоненты могут давать не тот результат, который нужен.

Включение и отключение смешивания выполняется командами `glEnable()` и `glDisable()` со значением аргумента `GL_DITHER`. Запомните, что передача полутонов, в отличие от многих остальных режимов, по умолчанию активна.

Режим индексации цвета

В режиме индексации цвета перед рисованием сцены OpenGL использует таблицу цветов. Это напоминает смешивание красок на палитре художника; аналогично цветовая палитра в компьютере ограничивает множество RGB-значений избранными, соответствующими заданным индексам, как показано на рис. 4.4.

Художник рисует сцену, выбирая цвета из палитры и закрашивая требуемые области нужным цветом. Компьютер хранит цветовые индексы в битовых плоскостях для каждого пиксела. Затем данное значение битовой плоскости сопоставля-

ется с цветовой таблицей, и на экране рисуется пиксел требуемого цвета, как показано на рис. 4.5.

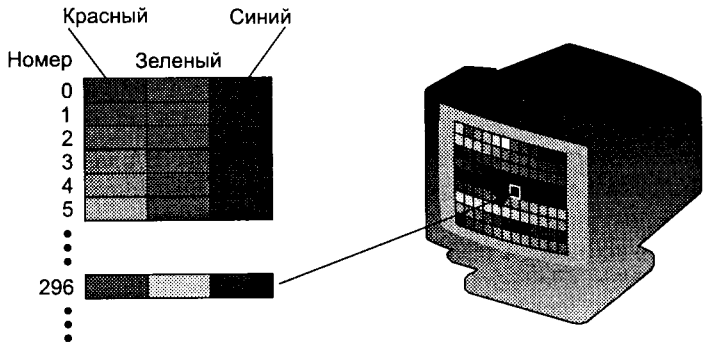


Рис. 4.4. Карта цветов

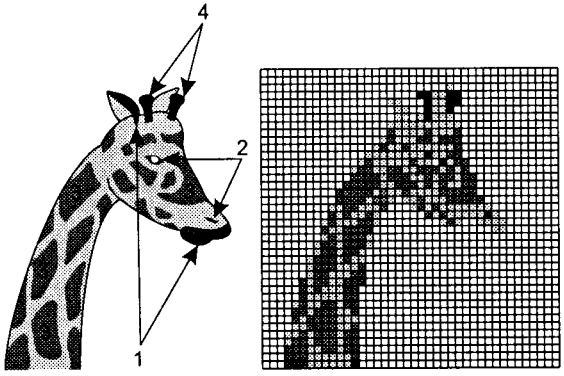


Рис. 4.5. Использование цветовой карты для рисования изображения

В режиме индексации цвета число доступных цветов ограничено размером цветовой карты и числом доступных битовых плоскостей. Размер цветовой карты определяется возможностями оборудования. Он всегда кратен двум и обычно лежит в пределах от 256 (2^8) до 4096 (2^{12}), где показатель степени — число битовых плоскостей. Если в цветовой карте содержится 2^n индексов и доступно m битовых плоскостей, количество доступных записей равно меньшему значению из 2^n и 2^m.

В режиме RGBA цвет каждого пиксела не зависит от цвета других пикселов. А вот в режиме индексов цвета каждый пиксел связан с соответствующим значением индекса в цветовой таблице. Если содержимое элемента цветовой таблицы изменяется, все пикселы с таким значением индекса меняют цвет.

Выбор между режимами RGBA и индексации цвета

Выбор между этими двумя режимами должен производиться на основе возможностей вашего оборудования и ваших требований. Для подавляющего количества систем большее число цветов доступно в режиме RGBA, чем в режиме индекса-

ции цвета. Также для некоторых эффектов, таких как затенение, освещение, текстурирование и туман, RGBA предоставляет более гибкие возможности, в сравнении с индексацией.

Использование индексации цвета оправдано в следующих случаях:

- Если приложение, оперирующее индексированными цветами, переносится на другую платформу.
- Если число битовых плоскостей невелико, режим RGBA дает достаточно жалкий набор цветов. Например, при наличии всего 8 битовых плоскостей в режиме RGBA доступны только 3 бита на красный, 3 бита на зеленый и 2 бита на синий. То есть возможны только 8 (2^3) градаций красного и зеленого, и только 4 градации синего. Переход между различными цветами чрезвычайно груб.

  Тогда, при ограниченных требованиях к передаче полутонов, предпочтительней использовать цветовую таблицу для загрузки большего числа набора цветов. Например, если нужна только синяя гамма, цветовая таблица позволяет хранить 256 (2^8) синих полутонов, что гораздо лучше, чем 4 градации в режиме RGBA. Конечно, таким образом вы заполните всю таблицу оттенками синего цвета, и у вас не будет ни красного, ни зеленого, ни каких других комбинаций.
- Режим индексации цвета очень удобен для различных хитростей, таких как анимация и рисование в слоях. (См. главу 14 для получения дополнительной информации.)

Вообще говоря, используйте режим RGBA везде, где это возможно. Он работает с текстурами и хорошо сочетается с освещением, затенением, туманом, размытием и смешиванием.

Переключение между режимами

Допустим, вы не можете выбрать между режимом RGBA и индексации цвета. Например, вам нужны и анимация с индексированными цветами, и текстурирование, то есть режим RGBA. Или же вам потребовалось переключиться между режимами одиночной и двойной буферизации. Например, у вас очень мало битовых плоскостей, скажем, восемь. В режиме одиночной буферизации в вашем распоряжении 256 (2^8) цветов, а режим двойной буферизации, нужный для устранения мерцания при анимации, дает только 16 (2^4) цветов. Предположим, что требуется нарисовать движущийся объект без мерцания и вы готовы пожертвовать цветностью ради двойной буферизации (может быть, объект движется слишком быстро, и точная детализация не нужна — кто различит детали на такой скорости). Но когда объект останавливается, надо нарисовать его в режиме одиночной буферизации, уже полноцветным.

К сожалению, большинство оконных систем не позволяют так легко переключаться между режимами. Так, в системе X Window режим отображения цвета — атрибут объекта X Visual. А объект X Visual должен быть определен до создания

окна, то есть не может меняться при жизни окна. Фактически вы «приклеиваетесь» либо к двойной буферизации, либо к режиму RGBA.

Хитроумное решение этой проблемы — создание более одного окна, работающего в собственном режиме. Вам остается только управлять видимостью окон (отображать или не отображать в системе X Window, управлять или не управлять виджетами Motif и Athena) и рисовать объект в подобающем, видимом окне.

Задание цветовой и полутоновой моделей

OpenGL поддерживает текущий цвет (в режиме RGBA) и текущий индекс цвета (в режиме индексации цвета). Пока не используется более сложная цветовая модель, такая как освещение или текстурирование, каждый объект рисуется этим текущим цветом (или индексом цвета). Посмотрите на следующий псевдокод:

```
set_color(RED);
draw_item(A);
draw_item(B);
set_color(GREEN);
set_color(BLUE);
draw_item(C);
```

Элементы A и B рисуются красным, а элемент C — синим. Четвертая строка, устанавливающая цвет в зеленый, не имеет смысла (кроме того, что дает задержку во времени). Без освещения или текстурирования, если текущий цвет установлен, впоследствии все элементы рисуются этим цветом, пока он не будет сменен.

Задание цвета в режиме RGBA

В режиме RGBA текущий цвет устанавливает команда `glColor*()`.

void `glColor3`{b s I f d ub us ui}(*TYPE r, TYPE g, TYPE b*);
void `glColor4`{b s I f d ub us ui}(*TYPE r, TYPE g, TYPE b, TYPE a*);
void `glColor3`{b s I f d ub us ui}v(const *TYPE* **v*);
void `glColor4`{b s I f d ub us ui}v(const *TYPE* **v*);
Определяет текущие значения для красной, зеленой, синей и альфа-составляющих. Эта команда может иметь три суффикса, обозначающих различные параметры. Первый суффикс — 3 или 4, признак задания альфа-значения. Если альфа-значение не задается, оно принимается равным 1.0. Второй суффикс определяет тип данных аргументов: байт (byte), короткое целое (short), целое (integer), вещественное (float), двойной точности (double), беззнаковый байт (unsigned byte), беззнаковое короткое целое (unsigned short) или беззнаковое целое (unsigned integer). Третий суффикс — `v`, отвечает за то, является ли аргумент указателем на массив значений заданного типа данных.

Для версий `glColor*()`, принимающих в аргументах вещественные данные, значения должны быть в диапазоне от 0.0 до 1.0, минимального и максимального значений, которые могут быть сохранены в видеобуфере. Компоненты типа целое

число без знака (unsigned integer) линейно преобразуются (clamped) к вещественному типу, максимальное значение приравнивается к 1.0, а 0 приравнивается к 0.0. Компоненты типа целое число со знаком (signed integer) линейно преобразуются к вещественному типу, причем ограничиваются сверху значением 1.0, а снизу — значением –1.0 (табл. 4.1).

Значения и вещественного типа, и знакового целого преобразуются к диапазону [0, 1] перед обновлением текущего цвета или параметров освещенности текущего материала. После вычисления освещенности результирующий цвет, попадающий вне диапазона [0, 1], приводится в рамки данного диапазона перед интерполяцией или записью в видеобуфер. Если освещенность отключена, ограничивание компонентов цвета делается перед растеризацией.

Таблица 4.1. Приведение значений цвета к вещественному типу

Суффикс	Тип данных	Минимальное значение	Минимальное значение приводится к	Максимальное значение	Максимальное значение приводится к
b	1-byte integer	–128	–1.0	127	1.0
s	2-byte integer	–32 768	–1.0	32 767	1.0
i	4-byte integer	–2 147 483 648	–1.0	2 147 483 647	1.0
ub	unsigned 1-byte integer	0	0.0	255	1.0
us	unsigned 2-byte integer	0	0.0	65 535	1.0
ui	unsigned 4-byte integer	0	0.0	4 294 967 295	1.0

Аналогичная команда `glSecondaryColor*()` предусмотрена для определения цвета, применяемого после текстурирования (если отключено освещение). (Для получения более подробной информации см. раздел «Применение вторичного цвета после текстурирования» главы 9.)

Задание цвета в режиме индексации цвета

В режиме индексации цвета для выбора индекса цвета в качестве текущего используется функция `glIndex*()`.

void `glIndex`{sifd ub}(*TYPE c*);
void `glIndex`{sifd ub}v(const *TYPE* **c*)
Устанавливает текущий индекс цвета в *c*. Первый суффикс команды определяет тип значений аргументов: короткое целое (short), целое (integer), вещественное (float), двойной точности (double) или беззнаковый байт (unsigned byte). Второй, опциональный, суффикс определяет, является ли аргумент массивом значений заданного типа данных (массив содержит только одно значение).

В разделе «Очистка окна» главы 2 было приведено описание функции `glClearColor()`. Для режима индексации цвета предусмотрена аналогичная функция `glClearIndex()`.

void `glClearIndex`(GLfloat *cindex*);
Устанавливает текущий цвет очистки в режиме индексации цвета. Значение индекса очистки по умолчанию равно 0,0. При включенной индексации цвета тот же эффект дает команда `glClear(GL_COLOR_BUFFER_BIT)`.

ПРИМЕЧАНИЕ

В OpenGL нет команд для загрузки значений цветов в цветовую таблицу. Эти операции обычно выполняет оконная система. GLUT предоставляет функцию glutSetColor(), которая вызывает специфичную для оконной системы команду.

Текущий индекс хранится как значение вещественного типа. Целые значения преобразуются к вещественным напрямую. Значения за пределами диапазона индексов цветов не подвергаются ограничиванию. Однако перед размытием (если включено) и записью в видеобуфер индекс преобразуется в вещественный формат с фиксированной точкой. Все биты в целой части результирующего значения с фиксированной точкой, которые не соответствуют битам в видеобуфере, игнорируются.

Задание модели закрашивания

Линия или залитый многоугольник могут быть нарисованы одним цветом (заливка) или множеством различных цветов (градиентная заливка, также называемая *закрашиванием по Гуро*). Нужный метод определяется функцией `glShadeModel()`.

`glShadeModel`(GLenum *mode*);
Устанавливает модель закрашивания. Аргумент *mode* может иметь значение `GL_SMOOTH` (по умолчанию) или `GL_FLAT`.

При сплошной закраске цвет одной определенной вершины независимого примитива дублируется в направлении всех вершин примитива. При градиентной заливке цвет каждой вершины рассматривается раздельно. Если это линия, цвета интерполируются между цветами вершин отрезка. Для многоугольника цвета внутренней области вычисляются путем интерполяции цветов вершин. Программа из листинга 4.1 рисует треугольник с градиентной заливкой, показанный на цветном рисунке 11.

Листинг 4.1. Рисование треугольника с градиентной заливкой: smooth.c

```
void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_SMOOTH);
}

void triangle(void)
{
```

продолжение ⇩

Листинг 4.1 (*продолжение*)

```
    glBegin(GL_TRIANGLES);
        glColor3f(1.0, 0.0, 0.0);
        glVertex2f(5.0, 5.0);
        glColor3f(0.0, 1.0, 0.0);
        glVertex2f(25.0, 5.0);
        glColor3f(0.0, 0.0, 1.0);
        glVertex2f(5.0, 25.0);
    glEnd();
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    triangle();
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
       gluOrtho2D(0.0, 30.0, 0.0, 30.0*(GLfloat) h/(GLfloat) w);
    else
       gluOrtho2D(0.0, 30.0*(GLfloat) w/(GLfloat) h, 0.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

При градиентной заливке цвета ближайших пикселов ненамного различаются. В режиме RGBA смежные пикселы с немного различающимися цветами выглядят одинаково, то есть смена цвета вдоль многоугольника происходит постепенно. А вот в режиме индексации цвета соседние пикселы могут ссылаться на различные значения в таблице цветов, составленной из произвольных цветов. Поскольку соседние элементы таблицы могут содержать совершенно различные цвета, то и градиентно залитый многоугольник в режиме индексации цвета может выглядеть необычайно психоделично.

Решить проблему поможет палитра с индексами, определяющими плавно изменяющиеся по линейному закону цвета. Помните, что загрузка цветов в таблицу цветов — работа для оконной системы, а не для OpenGL. Правда, в GLUT существует такая функция `glutSetColor()` для загрузки в таблицу цветов одиночного индекса. Первый ее аргумент — это индекс, остальные задают красную, зеленую и синюю составляющие. Поэтому, чтобы загрузить целых 32 смежных индекса цветов (от 16 до 47) желтой гаммы, выполните цикл:

```
for (i = 0; i < 32; i++) {
   glutSetColor(16+i, 1.0* (i/32.0), 1.0*(i/32.0), 0.0);
}
```

Тогда, когда потребуется градиентная заливка примитивов, нарисованных только цветами с индексами от 16 до 47, эти многоугольники будут плавно залиты желтым цветом.

При сплошной закраске цвет какой-то одной вершины задает цвет заливки примитива. Для отрезка это цвет конечной (второй) вершины. Для многоугольника используется цвет определенной по некоторым правилам вершины. Номер вершины можно узнать из табл. 4.2. Отсчет всех номеров вершин и многоугольников в ней начинается с 1. OpenGL следует этим правилам, но лучший способ избежать неизвестности в вопросе цвета заливки многоугольника — это установить одинаковый цвет для всех вершин примитива.

Таблица 4.2. Выбор цвета заливки для i-го многоугольника

Тип многоугольника	Вершина, цвет которой используется для заливки
Независимый многоугольник	1
Треугольник полосы (с общей гранью)	$i + 2$
Треугольник веера (с общей вершиной)	$i + 2$
Независимый треугольник	$3i$
Связанный четырехугольник (через грань)	$2i + 2$
Независимый четырехугольник	$4i$

5

Освещение

После прочтения этой главы вы сможете:

- Понимать, как OpenGL приближает сцену к условиям реального освещения
- Рисовать освещенные объекты путем задания материала, источников света и свойств модели освещения
- Задавать свойства материала освещаемого объекта
- Манипулировать стеком матриц для управления положением источников света

Как вы видели в главе 4, OpenGL вычисляет цвет пикселов в конечной отображаемой на экран сцене, удерживаемой в буфере кадров. Часть этих вычислений зависит от того, какое освещение используется и как объекты в сцене отражают или поглощают свет. В качестве примера можно привести океан, цвет которого яркий в солнечный день и серый в облачный. Наличие солнечного света или облаков определяет, выглядит океан ярко-бирюзовым или грязно-серо-зеленым. Фактически большинство объектов не похожи на трехмерные до тех пор, пока они не освещены. На рис. 5.1 показаны две версии одинаковой сцены (одиночной сферы): одна из них с освещением, другая — без.

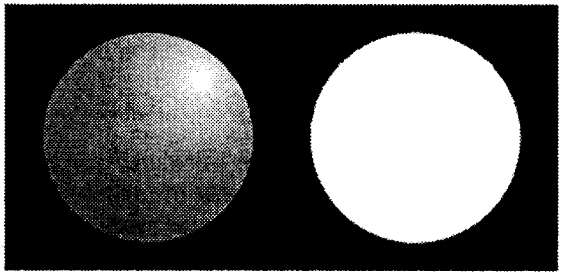


Рис. 5.1. Освещенная и неосвещенная сферы

Как видно из рисунка, неосвещенная сфера выглядит как плоский диск. Это показывает, насколько критично взаимодействие между объектами и светом при создании трехмерной сцены.

В OpenGL можно управлять освещением и объектами в сцене с помощью множества эффектов. Эта глава начинается с примера удаления невидимых поверхностей. Затем в ней объясняется, как управлять освещением в сцене, рассматривается модель освещения OpenGL и детально описывается, как установить множество параметров освещения для получения нужного эффекта. В конце главы приведены математические вычисления, показывающие, как освещение влияет на цвет.

Эта глава состоит из следующих основных разделов:

- **Удаление невидимых поверхностей.** Описывает основы удаления невидимых поверхностей из сцены.
- **Реальный мир и освещение в OpenGL.** Раскрывает в общих чертах, как работает освещение в реальном мире и как OpenGL реализует эту модель.
- **Пример: рисование освещенной сферы.** Знакомит с возможностями освещения в OpenGL путем демонстрации небольшой программы, рисующей освещенную сферу.
- **Создание источников света.** Описывает элементы модели освещения и управление ими.
- **Выбор модели освещения.** Рассказывает, как определить и расположить источники света.
- **Задание свойств материала.** Показывает, как описать свойства объектов, чтобы они взаимодействовали со светом так, как это нужно.

- **Математика освещения.** Представляет математические вычисления, проводимые OpenGL для определения воздействия освещения на сцену.
- **Освещение в режиме индексации цвета.** Рассматривает различия между режимами RGBA и индексации цвета в условиях освещения.

Вычисления освещенности обычно происходят до текстурирования. В версии 1.2 появился режим освещения (GL_SEPARATE_SPECULAR_COLOR), в котором вычисление отраженного цвета отделено от излучательности, рассеянного и фонового света и где оно выполняется после текстурирования. Данный режим, как правило, увеличивает яркость освещения, делая ее менее зависимой от цвета текстурного рисунка.

Удаление невидимых поверхностей

Дойдя до конца этого раздела, вы начнете рисовать закрашенные трехмерные объекты. Работая с ними, очень важно уметь отделять объекты, находящиеся ближе к положению наблюдения, от тех, которые закрыты другими, более близкими объектами.

Изменение точки обзора может привести к изменению взаимного перекрытия объектов. Например, если посмотреть на сцену с противоположной стороны, все объекты, ранее находившиеся перед другими, оказываются за ними. Для получения реалистичной сцены эти взаимные перекрытия должны быть выявлены. Допустим, код выглядит следующим образом:

```
while (1)  {
     get_viewing_point_from_mouse_position();
     glClear(GL_COLOR_BUFFER_BIT);
     draw_3d_object_A();
     draw_3d_object_B();
}
```

При некоторых положениях мыши объект A может закрывать объект B. При других — наоборот, объект B может заслонять объект A. Если специально ничего не предпринять, приведенный код всегда будет рисовать вторым объект B (то есть поверх объекта A) вне зависимости от выбранного положения просмотра. В худшем сценарии, когда объекты A и B пересекаются так, что часть объекта B скрыта объектом A, а часть объекта B закрывает собой объект A, не даст правильного результата и изменение порядка рисования.

Исключение из сцены частей залитых объектов, которые заслонены другими, называется *удалением невидимых поверхностей*. (Удаление невидимых линий, то есть то же самое применительно к каркасным объектам, немного сложнее и не обсуждается в этой главе. См. раздел «Удаление невидимых линий» главы 14.) Наиболее легкий путь удаления невидимых поверхностей — использование буфера глубины (иногда называемого *z*-буфером) (см. главу 10).

Глубина — расстояние от плоскости просмотра (обычно ближней плоскости отсечения) до каждого пиксела окна. Изначально значения глубины для всех пикселов устанавливаются равными максимальному расстоянию (обычно дальняя

плоскость отсечения) с помощью команды `glClear()` со значением аргумента `GL_BUFFER_BIT`. Затем объекты в сцене рисуются в произвольном порядке.

Аппаратные или программные графические вычисления преобразуют каждую рисуемую поверхность в набор пикселов окна, где эта поверхность визуализуется, если не перекрывается чем-то еще. Кроме того, вычисляется расстояние от плоскости просмотра. При включении буферизации глубины перед рисованием каждого пиксела выполняется сопоставление глубины со значением, уже хранящимся в буфере. Если новый пиксел оказывается ближе (впереди), в буфер записываются новые значения цвета пиксела и глубины. Если же глубина больше текущей, новый пиксел отбрасывается без изменения старой информации.

Для использования буфера глубины его необходимо включить. Это делается только один раз. Впоследствии при любой перерисовке сцены необходимо очищать буфер глубины, а затем можно рисовать объекты сцены в любом порядке.

Чтобы внести в приведенный выше код возможность работы с удалением скрытых поверхностей, измените его следующим образом:

```
glutInitDisplayMode(GLUT_DEPTH |.... );
glEnable(GL_DEPTH_TEST);
...
while (1)  {
    get_viewing_point_from_mouse_position();
    glClear(GL_COLOR_BUFFER_BIT);
    draw_3d_object_A();
    draw_3d_object_B();
}
```

При задании указанного значения аргумента функции `glClear()` очищаются сразу оба буфера: глубины и цвета.

Тест на глубину может отрицательно повлиять на производительность приложения. Но так как информация отбрасывается, а не используется для рисования, удаление невидимых поверхностей способно несколько компенсировать это падение производительности. Вместе с тем степень воздействия буфера глубины на быстродействие существенно зависит от реализации. «Программный» буфер глубины (размещенный в памяти процессора) явно будет медленнее, чем специально реализованный на аппаратном уровне.

Реальный мир и освещение в OpenGL

При взгляде на физические поверхности глаз воспринимает цвет в зависимости от распределения энергии фотонов, попадающих на сетчатку глаза и раздражающих ее. (См. раздел «Цветовосприятие» главы 4.) Эти фотоны поступают от источника света или нескольких источников, некоторые из них поглощаются, а некоторые отражаются от поверхности. Кроме того, различные поверхности могут иметь различные свойства: некоторые отполированы и прекрасно отражают свет в нужных направлениях, а другие — рассеивают фотоны одинаково во все стороны. Большинство поверхностей представляют собой что-то среднее.

OpenGL представляет свет так, как будто он разбит на красную, синюю и зеленую составляющие. То есть цвет источника света характеризуется интенсивностью RGB-компонентов, а в свойства материала поверхности входят процентные соотношения красной, синей и зеленой составляющих, отражаемых в разных направлениях. В OpenGL вычисления освещения представляют собой приближение к действительности, но оно дает хорошие результаты и быстро работает. Если необходимы более точные результаты (или другая модель освещения), вы должны все высчитать сами программным способом. Но такая программная реализация довольно сложна, в чем можно убедиться, потратив пару часов на чтение любой книги по оптике.

В модели освещения OpenGL свет в сцене исходит от нескольких источников, которыми можно управлять по отдельности. Часть света доходит из определенного направления или положения, а часть представляет собой рассеянный свет. Например, если включить в комнате электрическую лампочку, большая часть света будет поступать от лампочки, а часть — будет являться отраженным светом от одной, двух, трех или более стен. Этот отраженный свет (называемый *фоновым*, ambient) предполагается настолько распределенным, что нет никакой возможности определить его исходное направление; в то же время при отключении источника света он исчезает.

Наконец, в сцене может присутствовать *общий фон* (фоновое освещение, general ambient), свет, который не исходит от определенного источника, то есть как будто он столько раз отражался, что источник его излучения невозможно определить.

В модели OpenGL источники света влияют на внешний вид сцены, только если определены поверхности, поглощающие и отражающие свет. Каждая поверхность может быть задана как материал со своими свойствами. Материал может излучать свой собственный свет (как, например, фары автомобиля), отражать какой-то приходящий свет во всех направлениях или отражать часть приходящего света в определенном направлении (как, например, зеркало или любая другая полированная поверхность).

В модели освещения OpenGL свет разбивается на четыре независимые составляющие: фоновое освещение, рассеянный свет, отраженный свет и излучаемый свет. Все четыре составляющие вычисляются независимо, а затем складываются друг с другом.

Фоновый, рассеянный, отраженный и излучаемый свет

Фоновое рассредоточенное освещение — это свет, отраженный окружающими объектами столько раз, что его направление невозможно установить, — он выглядит приходящим со всех направлений. Освещение комнаты имеет большую часть фоновой составляющей, поскольку большая часть света, достигающего глаза, сначала отразилась от множества поверхностей. Для внешнего наблюдателя освещенный дверной проем содержит маленькую общую составляющую, так как большая часть света исходит из одного места, и до тех пор, пока наблюдатель находится вне помещения, отраженная от других объектов составляющая невелика.

Когда фоновый свет попадает на поверхность, он отражается одинаково во всех направлениях.

Рассеянный (или диффузный, diffuse) свет — это свет, поступающий из одного направления. При прямом падении на поверхность он более яркий, чем при легком прикосновении к ней вскользь. При столкновении с поверхностью происходит отражение диффузного цвета, независимое от направления, то есть яркость света не зависит от местоположения наблюдателя. Любой свет, приходящий из определенного места или направления, скорее всего, имеет рассеянную диффузную составляющую.

Отраженный (или зеркальный, specular) свет приходит из конкретного направления и отражается от поверхности также в определенном направлении. Направленный лазерный пучок лучей имеет практически 100-процентное отражение от высококачественного зеркала. Освещенный солнцем металл или пластмасса также имеют высокие коэффициенты отражения, а отражающая способность мела или ковра практически равна нулю. Отражающую способность можно представить как блеск (shininess).

Кроме общего, рассеянного и отраженного цветов материалы могут обладать *излучаемым* (или эмиссионным, emissive) цветом, который имитирует свет, исходящий от объекта. В модели освещения OpenGL цвет самосвечения поверхности добавляет яркости объекту и не зависит от любых других источников света. Кроме того, излучение не вносит дополнительного освещения в сцену.

Хотя источник света вырабатывает единое частотное распределение, фоновая, рассеянная и отраженная составляющие могут быть разными. Так, если в комнате с белым потолком находится источник красного света, рассредоточенный свет будет ближе к белому, в то время как на объекты попадает красный свет. OpenGL позволяет каждой световой составляющей присваивать различные независимые цвета.

Цвета материала

Согласно модели освещения OpenGL, цвет материала задается в зависимости от того, какой процент приходящего красного, зеленого или синего света поверхность отражает. Например, абсолютно красный мяч отбрасывает весь приходящий красный свет и полностью поглощает зеленый и синий свет. Соответственно, если посмотреть на этот мяч при белом освещении (которое состоит из одинакового количества красного, зеленого и синего света), весь красный свет отразится и будет виден красный мяч. Освещенный беспримесным красным светом, он также выглядит красным. Но, если на мяч посмотреть при абсолютно зеленом свете, мяч будет черным (весь зеленый свет поглощается, а красной составляющей нет, то есть ничего не отражается).

Как и источники света, материалы имеют различные фоновые, рассеянные и отраженные цвета, которые определяют способности материала в отношении аналогичных составляющих света. При расчетах отдельные компоненты объединяются: фоновый свет и фоновый цвет материала, диффузный свет и рассеивающий

цвет, и аналогично коэффициент отражения объединяется с отраженной составляющей. «Фоновое» и «рассеивающее» свойства определяют цвет материала и обычно если не одинаковы, то похожи. Отражающая способность обычно соответствует белому или серому свету, но так, чтобы наиболее яркие места по цвету совпадали с зеркальной составляющей источника. Представим красную полированную сферу в ярких солнечных лучах. Тогда большая часть сферы будет красной, а самые освещенные точки — белыми.

Значения RGB для источников света и материалов

Значения компонентов цвета для источников света означают не то же, что для материалов. Для источника света эти числа зависят от процентной доли к полной интенсивности каждого цвета. Так, если все составляющие (R, G и B) равны 1.0, свет будет настолько белым, насколько это возможно. Если значения равны 0.5, цвет по-прежнему будет белым, но будет уже половинной интенсивности, то есть серым. Если R = G = 1, а B = 0 (красный и зеленый, без синего), свет желтый.

Для материалов числа берутся исходя из отражающей способности данного цвета. То есть значения R = 1, G = 0.5 и B = 0 подразумевают полное отражение красного света, половинное — зеленого, и полное поглощение синего света. Иными словами, если свет (L) OpenGL состоит из LR, LG, LB, а материал (M) имеет соответствующие составляющие MR, MG, MB, то без учета всех прочих эффектов отражения свет, доходящий до глаза, рассчитывается как (LR × MR, LG × MG, LB × MB).

В то же время, если рассматривать два источника света (R1, G1, B1) и (R2, G2, B2), их компоненты будут просто суммированы, и глаз будет видеть свет (R1 + R2, G1 + G2, B1 + B2). Если какая-то из этих сумм превысит 1 (то есть цвет более яркий, чем может быть отображен оборудованием), она ограничивается 1.

Пример: рисование освещенной сферы

Для добавления освещения в сцену нужно сделать следующее:

1. Определить векторы нормалей для каждой вершины каждого объекта. Они устанавливают ориентацию объекта относительно источников света.
2. Создать, выбрать и расположить один или несколько источников света.
3. Создать и выбрать *модель освещения*, которая определяет уровень фонового света и правильное местоположение точки обзора (подходящее для вычисления освещенности).
4. Определить свойства материала для объектов в сцене.

Программа, приведенная в листинге 5.1, решает эти задачи. Она рисует сферу, освещенную единственным источником света, как показано на рис. 5.1.

Листинг 5.1. Рисование освещенной сферы: light.c

```
void init(void)
{
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 50.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
   GLfloat white_light[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat lmodel_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_SMOOTH);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
   glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient)

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glutSolidSphere(1.0, 20, 16);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
              1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho(-1.5*(GLfloat)w/(GLfloat)h, 1.5*(GLfloat)w/(GLfloat)h, -1.5,
              1.5, -10.0, 10.0);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
```

продолжение ⇩

Листинг 5.1 (*продолжение*)

```
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

Вызовы функций, связанные с освещением, находятся в процедуре `init()`. Они вкратце описаны в ближайшем разделе и более подробно — далее в этой главе. Заметьте, что в листинге 5.1 используется режим RGBA, а не индексации цвета. Вычисление освещенности в OpenGL различно для двух режимов, и фактически возможности освещения в режиме индексации цвета более ограничены. Поэтому режим RGBA более предпочтителен при работе со светом, и все примеры в этой главе написаны с его использованием. (См. раздел «Освещение в режиме индексации цвета» этой главы для получения дополнительной информации.)

Задание векторов нормалей для каждой вершины каждого объекта

Нормали определяют расположение объекта относительно источников света. Чтобы рассчитать количество света, получаемого вершиной от каждого источника света, OpenGL отталкивается от ее нормали. В данном примере нормали для сферы устанавливаются функцией `glutSolidSphere()`.

Чтобы освещенность вычислялась правильно, векторы нормали должны иметь единичную длину. Но это качество по ходу теряется, так как матрица модельно-видовых преобразований не масштабирует нормали к поверхности. Чтобы нормали всегда оставались нормализованными, используйте функцию `glEnable()` со значением аргумента `GL_NORMALIZE` или `GL_RESCALE_NORMAL`.

В режиме `GL_RESCALE_NORMAL` выполняется умножение каждой составляющей нормали к поверхности на одно и то же значение, определяемое из матрицы модельно-видовых преобразований. Такая процедура правильно работает при условиях, если нормаль до того масштабирована равномерно и изначально была единичной. Режим `GL_NORMALIZE` более универсален, здесь вычисляется длина вектора нормали, а затем каждый компонент делится на полученное значение. Это гарантирует, что нормали останутся единичными, но данный способ более трудоемок, чем простое перемасштабирование нормалей. (См. разделы «Векторы нормали» главы 2 и приложение Д, «Вычисление векторов нормалей», для получения дополнительной информации.)

ПРИМЕЧАНИЕ

В некоторых реализациях OpenGL в режиме GL_RESCALE_NORMAL действительно выполняется именно нормализация векторов нормалей, а не обычное масштабирование. Но полагаться на это не стоит, так как здесь нет способа получить достоверную информацию относительно поведения функции.

Создание, размещение и включение одного или нескольких источников света

В листинге 5.1 используется только один, белый, источник света — GL_LIGHT0. Его положение определяется функцией glLightfv(), и соответственно вычисляются рассеивание и отражение. Если нужен другой цвет источника света, измените значения аргументов glLight*().

В сцене может присутствовать до восьми различных источников света произвольного цвета. (Некоторые реализации OpenGL поддерживают более восьми источников света.) По умолчанию цвет источников, отличных от GL_LIGHT0, — черный. Источники можно размещать там, где это необходимо: рядом со сценой (например, настольная лампа) или бесконечно далеко (имитация солнца). Кроме того, можно управлять толщиной луча — делать его сфокусированным или широким. Помните, что добавление даже одного источника света требует значительных вычислений при рисовании, то есть сказывается на производительности. (См. раздел «Создание источников света» для получения дополнительной информации о том, как создать источники света с заданными свойствами.)

После определения свойств эти источники света необходимо включить с помощью функции glEnable(). Кроме того, чтобы OpenGL принимал освещение во внимание, нужно вызвать функцию glEnable() со значением аргумента GL_LIGHTING. (См. раздел «Включение источников света» для получения дополнительной информации.)

Выбор модели освещения

Как вы уже могли догадаться, функция glLightModel*() описывает параметры модели освещения. В листинге 5.1 явно определен только один ее элемент — фоновый рассеянный свет (global ambient). Модель освещения также включает в себя местоположение наблюдателя, который может быть локальным или бесконечно удаленным, и то, как выполняются вычисления для передней и задней поверхностей объектов сцены, одинаково или индивидуально. В листинге 5.1 для обоих параметров приняты значения по умолчанию — наблюдатель находится в бесконечности, а освещение одностороннее. В таких условиях результаты несколько менее реалистичны, поскольку не принимаются во внимание углы между точкой обзора и каждым объектом. Если наблюдатель находится рядом со сценой, знание этих углов необходимо, что значительно увеличивает сложность вычислений. Так как в приведенной программе задняя поверхность сферы никогда не видна (как бы находится внутри сферы), одностороннего освещения вполне достаточно. (См. раздел «Выбор модели освещения» для получения подробной информации).

Задание свойств материала для объектов в сцене

Свойства материала объекта определяют характер рассеивания света, то есть — из какого материала этот объект «изготовлен». Поскольку взаимодействие между

поверхностью объекта и падающим светом довольно сложно, правильное задание свойств материала, дающих нужный результат, — целое искусство. В вашем распоряжении не только фоновый, рассеянный и отраженный цвета материала, но и блеск, то есть насколько хорошо материал отполирован. В обсуждаемом примере явно, с помощью функции `glMaterialfv()`, определяются лишь два свойства — отражение и блеск. (Для получения информации обо всех свойствах материала и примерах их использования см. раздел «Задание свойств материала» этой главы).

Замечания на память

При написании собственной программы с использованием освещения учтите, что для одних параметров в самый раз будут значения по умолчанию, а для других их потребуется изменить. Кроме того, не забудьте включить источники света, которые вы определили, и вычисление освещенности. Наконец, помните, что при изменении условий освещенности увеличить эффективность помогают списки отображения. (См. раздел «Философия проектирования списков отображения» главы 7.)

Создание источников света

Источники света характеризуются набором свойств, включая цвет, положение и ориентацию. Ниже описывается, как управлять этими параметрами и каким окажется результирующий свет. Свойства источников света устанавливаются функцией `glLight*()`.

void `glLight`{if}(GLenum *light*, GLenum *pname*, *TYPE param*);
void `glLight`{if}`v`(GLenum *light*, GLenum *pname*, *TYPE* **param*);
Создает нумерованный источник света *light*, в диапазоне `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7`. Свойство источника света указывается символическим именем *pname*, согласно табл. 5.1. Аргумент *param* — это собственно значение какого-то одного свойства *pname* или указатель на набор значений — в векторной версии.

Использование функции `glLight*()` продемонстрировано в листинге 5.2.

Листинг 5.2. Задание цвета и положения источников света

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Таблица 5.1. Значения по умолчанию для аргумента pname функции glLight*()

Параметр	Значения по умолчанию	Описание
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	Цвет фонового освещения
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0) для LIGHT0 — белый или (0.0, 0.0, 0.0, 1.0) для остальных	Цвет рассеянного освещения
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0) для LIGHT0 — белый или (0.0, 0.0, 0.0, 1.0) для остальных	Цвет отраженного света
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	Координаты (*x, y, z, w*) источника света
GL_SPOT_DIRECTION	(0.0, 0.0, –1.0)	Направление (*x, y, z*) распространения света
GL_SPOT_EXPONENT	0.0	Распределение интенсивности светового пучка по экспоненте
GL_SPOT_CUTOFF	180.0	Угол разброса световых лучей
GL_CONSTANT_ATTENUATION	1.0	Ослабление интенсивности света, вне зависимости от направления и расстояния
GL_LINEAR_ATTENUATION	0.0	Линейное ослабление интенсивности света
GL_QUADRATIC_ATTENUATION	0.0	Квадратичное затухание света

Видно, что значения параметров помещены в массивы, а функция `glLightfv()` вызывается несколько раз. В этом примере первые три вызова являются лишними, так как устанавливаемые ими значения GL_AMBIENT, GL_DIFFUSE и GL_SPECULAR совпадают с предопределенными неявно.

ПРИМЕЧАНИЕ

Помните, что каждый источник света необходимо включить с помощью команды glEnable(). (Для получения более подробной информации о том, как это сделать, см. раздел «Включение источников света».)

В следующих разделах описываются все аргументы функции `glLight*()` и их возможные значения. Они связаны с моделью освещения сцены и свойствами материалов объектов. (См. разделы «Выбор модели освещения» и «Задание свойств материала» для получения дополнительной информации по этим двум вопросам раздельно и раздел «Математика освещения» на предмет их взаимосвязи.)

Цвет

OpenGL позволяет ассоциировать с каждым источником света три различных параметра цвета: GL_AMBIENT, GL_DIFFUSE и GL_SPECULAR. Параметр GL_AMBIENT выражает интенсивность фонового RGBA-освещения, которое привносит в сцену

какой-то источник света. Как видно из табл. 5.1, по умолчанию фоновое освещение источника отключено — GL_AMBIENT равно (0.0, 0.0, 0.0, 1.0). Это значение использовалось в листинге 5.1. Если в программу добавить синее общее освещение:

```
GLfloat light_ambient[] = { 0.0, 0.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
```

получится изображение, показанное на цветном рисунке 13.

Параметр GL_DIFFUSE приближен по значению к обычному естественному «цвету света». Он определяет RGBA-цвет рассеянного освещения, вносимого в сцену некоторым источником света. По умолчанию значение GL_DIFFUSE равно (1.0, 1.0, 1.0, 1.0) для GL_LIGHT0, что соответствует яркому белому свету, как показано на цветном рисунке 13. Для всех прочих источников света (GL_LIGHT1, ..., GL_LIGHT7) значение по умолчанию равно (0.0, 0.0, 0.0, 0.0).

Параметр GL_SPECULAR задает цвет отраженного света объекта. Обычно реальные объекты, такие как стеклянная бутылка, в своей наиболее освещенной части выдают блик, то есть имеют цвет отражающегося от них света (часто это белый цвет). Поэтому при создании реалистичных сцен GL_SPECULAR приравнивается GL_DIFFUSE. По умолчанию значение GL_SPECULAR равно (1.0, 1.0, 1.0, 1.0) для GL_LIGHT0 и (0.0, 0.0, 0.0, 0.0) для всех остальных источников света.

ПРИМЕЧАНИЕ

Альфа-компонент не используется до тех пор, пока не включено смешивание (см. главу 6). До этого момента его можно смело игнорировать.

Положение и ослабление

Как было сказано ранее, можно выбирать между источником света, который бесконечно удален от сцены, и между источником, расположенным рядом с ней. Источник первого типа называют *направленным*. При бесконечном удалении можно считать, что лучи света от него при падении на объект параллельны друг другу. Примером направленного источника света в реальном мире является солнце. Источники второго типа называют *позиционными*, так как положение такого источника качественно влияет на сцену, и в особенности направление, откуда приходят световые лучи. Настольная лампа — пример позиционного источника света. Различия между направленным и позиционным источниками света показаны на цветном рисунке 13. Источник света, используемый в листинге 5.1, является направленным:

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Видно, что для параметра GL_POSITION определяется вектор из четырех значений (x, y, z, w). Если последнее значение, w, равно нулю, имеет место направленный источник света, а координаты (x, y, z) описывают его ориентацию. Эта ориентация преобразуется модельно-видовой матрицей. По умолчанию значение параметра GL_POSITION равно (0, 0, 1, 0), то есть в сцену вводится направленный источник света, причем направленный в сторону отрицательных координат z.

(Кстати, ничто не запрещает нам создать направленный источник света (0; 0; 0), но смысл в этом напрочь отсутствует.)

Если значение w не равно нулю, задается позиционный источник света, а значения (x, y, z) определяют его положение в однородной системе координат объекта (см. приложение Е). В последующем положение преобразуется модельно-видовой матрицей и сохраняется в координатах наблюдателя. (Для получения дополнительной информации см. раздел «Управление положением и направлением источников света».) Также по умолчанию позиционный источник света излучает во всех направлениях, но ограничить разброс лучей конусом можно, если сделать этот источник света «прожектором». (Для получения более подробной информации по данном вопросу смотрите следующий раздел.)

ПРИМЕЧАНИЕ

Помните, что цвета лицевых многоугольников с градиентной заливкой определяются цветами вершин. Поэтому часто большие многоугольники несовместимы с локальным освещением. Так, если расположить источник света рядом с центром многоугольника, не исключено, что дальние вершины не получат достаточно света, то есть многоугольник будет выглядеть более темным, чем мы от него ждем. Для решения этой проблемы нужно разбить большой многоугольник на несколько более мелких.

Для реальных источников света интенсивность освещения падает с увеличением расстояния до источника. Поскольку направленный источник располагается бесконечно далеко, ослабление для него априори отключено. Однако для локального источника света оно может потребоваться. OpenGL реализует ослабление интенсивности источника света, умножая вклад этого источника на коэффициент ослабления:

$$\text{коэффициент ослабления} = \frac{1}{k_c + k_l d + k_q d^2},$$

где d — расстояние от источника света до вершины, k_c — GL_CONSTANT_ATTENUATION, k_l — GL_LINEAR_ATTENUATION, k_q — GL_QUADRATIC_ATTENUATION.

По умолчанию k_c равен 1, а k_l и k_q равны нулю, но любой из коэффициентов может иметь другое значение:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

Обратите внимание, что ослаблять допускается фоновую, рассеянную и отраженную составляющие, но никак не излучение и общий фоновый свет. Также заметьте, что раз ослабление требует дополнительных математических операций — деления для каждого вычисляемого цвета, то оно может ухудшить производительность приложения.

Прожекторы

Уже говорилось, что можно вынудить источник света действовать как прожектор, ограничив испускаемый свет конусом. Чтобы получить прожектор, нужно

задать размах этого конуса. (Помните, что, поскольку прожектор является позиционным источником света, его необходимо расположить в требуемом месте. Кроме того, OpenGL не мешает нам создать направленный прожектор, но это действие не оправдано с точки зрения смысла.) Для задания угла между осью конуса и луча по краю конуса используется параметр GL_SPOT_CUTOFF. Размах конуса равен удвоенному этому значению, как показано на рис. 5.2.

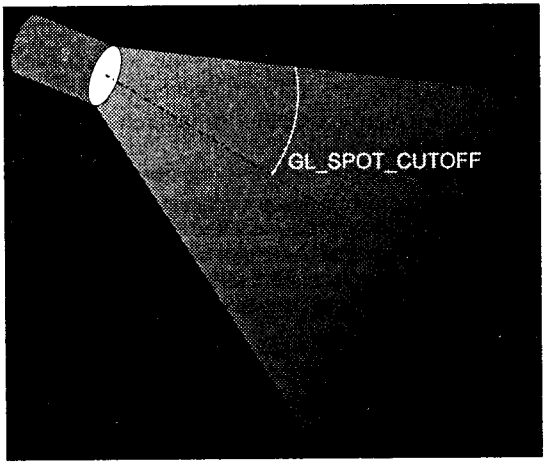


Рис. 5.2. Параметр GL_SPOT_CUTOFF

Свет не выходит за границы конуса — чего мы и ждали. По умолчанию величина GL_SPOT_CUTOFF равна 180.0. Это означает, что свет испускается во всех направлениях (размах в вершине конуса равен 360°). За исключением этого специального значения 180.0, допустимые значения лежат в диапазоне [0.0, 90.0]. Следующая команда ограничивает направление распространения света от источника углом 45°:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

Кроме того, необходимо задать направление освещения, то есть ось светового конуса:

```
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

Направление указывается в координатах объектов. По умолчанию это (0.0, 0.0, –1.0), то есть когда режим GL_SPOT_DIRECTION не включен, свет распространяется вдоль отрицательной оси *z*. Помните, что направление впоследствии преобразуется модельно-видовой матрицей так же, как и векторы нормалей, а результат сохраняется в видовых координатах. (Для получения дополнительной информации об этих преобразованиях см. раздел «Управление положением и направлением источников света».)

Помимо размаха и ориентации, существует еще два способа управления интенсивностью распределения света в конусе. Во-первых, можно ввести в действие коэффициент ослабления, описанный ранее, умножаемый на интенсивность света. Кроме того, есть параметр GL_SPOT_EXPONENT, по умолчанию равный нулю,

управляющий концентрацией светового пучка так, что интенсивность света больше на оси конуса и ослабляется к границам. Коэффициент ослабления равен косинусу угла между направлением света и направлением от источника света к освещаемой вершине, возведенному в степень (показатель экспоненты). Таким образом, больший коэффициент GL_SPOT_EXPONENT соответствует более сфокусированному источнику света. (Для получения дополнительной информации о вычислении освещенности см. раздел «Математика освещения».)

Несколько источников света

Как уже было отмечено, на сцене могут присутствовать до восьми источников света (или больше, в зависимости от реализации OpenGL). Поскольку OpenGL требуется вычислить, сколько света получает каждая вершина от каждого источника, увеличение числа источников света весомо влияет на быстродействие. Константы, ссылающиеся на восемь источников света, — это GL_LIGHT0, GL_LIGHT1, GL_LIGHT2, GL_LIGHT3 и т. д. Ранее нами устанавливались параметры для источника GL_LIGHT0. Если требуются дополнительные источники света, необходимо задать и их свойства, которые по умолчанию не совпадают со свойствами GL_LIGHT0. В листинге 5.3 задается белый прожектор с затуханием.

Листинг 5.3. Второй источник света

```
GLfloat light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat light1_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_position[] = { -2.0, 2.0, 1.0, 1.0 };
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };

glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
glLight f(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLight f(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.2);

glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);

glEnable(GL_LIGHT1);
```

Если эти строки добавить в листинг 5.1, сфера окажется освещена двумя источниками: одним направленным и одним прожектором.

Упражнение

Измените листинг 5.1 следующим образом:

- сделайте первый источник света цветным позиционированным, вместо белого направленного;

- добавьте дополнительный цветной прожектор. Подсказка: используйте код, аналогичный коду из предыдущего раздела;
- оцените, как эти изменения влияют на производительность.

Управление положением и направлением источников света

OpenGL работает с положением и направлением источника света так же, как обращается с геометрическими примитивами. Другими словами, источник света является субъектом тех же матричных преобразований, что и примитив. И говоря более определенно, когда команда `glLight*()` вызывается для указания позиции или ориентации источника света, это положение или направление преобразуется текущей модельно-видовой матрицей и сохраняется в видовых координатах. Это означает, что можно управлять положением и направлением источников света, изменяя содержимое модельно-видовой матрицы. (Проекционная матрица не влияет на эти параметры.) В данном разделе показано, как добиться трех перечисленных ниже эффектов, перенеся задание источников света в программе в другое место (относительно модельных и видовых преобразований):

- положение источника света не меняется;
- источник света движется вокруг неподвижного объекта;
- источник света движется вместе с точкой обзора.

Неподвижный источник света

В простейшем случае, как в листинге 5.1, положение источника света не изменяется. В таком варианте оно задается после всех используемых видовых и/или модельных преобразований. Программа из листинга 5.4, включающая в себя подпрограммы `init()` и `reshape()`, показывает, как может выглядеть соответствующий код.

Листинг 5.4. Неподвижный источник света

```
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
    glOrtho(-1.5, 1.5, -1.5*h/w, 1.5*h/w, -10.0, 10.0);
else
    glOrtho(-1.5*w/h, 1.5*w/h, -1.5, 1.5, -10.0, 10.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

/* далее в процедуре init() */
GLfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

Видно, что сначала устанавливаются окно просмотра и проекционная матрица. Затем в модельно-видовую матрицу загружается единичная матрица, и только

после этого задается положение источника света. Поскольку используется единичная матрица, явно установленное положение источника света (1.0, 1.0, 1.0) сохраняется и при умножении на модельно-видовую матрицу. А раз дальше ни положение источника света, ни модельно-видовая матрица не изменяются, направление распространения света остается прежним — (1.0, 1.0, 1.0).

Независимо движущийся источник света

Теперь предположим, что нужно вращать или перемещать источник света вокруг неподвижного объекта. Один из способов сделать это — задать местоположение источника света после изменяющего эту позицию модельно-видового преобразования. Можно начать с вызовов тех же команд процедуры `init()`, что описаны в предыдущем разделе. Затем надо выполнить желаемое модельно-видовое преобразование (на стеке модельно-видовых матриц) и сбросить позицию источника, скажем, в процедуре отображения `display()`. В листинге 5.5 показан примерный код процедуры `display()`.

Листинг 5.5. Независимо движущийся источник света

```
static GLdouble spin;

void display(void)
{
   GLfloat light_position[]  = { 0.0, 0.0, 1.5, 1.0 };
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix();
      gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
      glPushMatrix();
         glRotated(spin, 1.0, 0.0, 0.0);
         glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      glPopMatrix();
      glutSolidTorus(0.275, 0.85, 8, 15);
   glPopMatrix();
   glFlush();
}
```

Глобальная переменная *spin*, вероятнее всего, инициируется с устройства ввода. Процедура `display()` вызывает перерисовку сцены с источником света, повернутым вокруг неподвижного тора на *spin*°. Заметьте, что две пары команд, `glPushMatrix()` и `glPopMatrix()`, изолируют видовые и модельные преобразования. Так как в программе из листинга 5.5 точка обзора остается неизменной, текущая матрица помещается в стек, а затем командой `gluLookAt()` задается требуемое видовое преобразование. Далее получившаяся матрица снова заносится в стек перед преобразованием поворота командой `glRotated()`. После этого задается положение источника света в новой повернутой системе координат таким образом, что источник света оказывается повернутым относительно своего первоначального положения. (Помните, что позиция источника света сохраняется в видовых координатах, получаемых после преобразования модельно-видовой матрицей.) Когда матрица выгружается из стека, рисуется тор.

В листинге 5.6 приведена программа, поворачивающая источник света вокруг неподвижного объекта. При нажатии левой кнопки мыши угол поворота источника света увеличивается на 30°. Положение источника света на сцене обозначено маленьким неосвещенным каркасом куба.

Листинг 5.6. Перемещение источника света с помощью модельных преобразований: movelight.c

```
static int spin = 0;

void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_SMOOTH);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
}
/* Здесь происходит сброс после модельного преобразования (glRotated).
/* Источник света получает новые мировые координаты.
/* Куб обозначет положение источника света.
*/
void display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glTranslatef(0.0, 0.0, -5.0);

    glPushMatrix();
    glRotated((GLdouble) spin, 1.0, 0.0, 0.0);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glTranslated(0.0, 0.0, 1.5);
    glDisable(GL_LIGHTING);
    glColor3f(0.0, 1.0, 1.0);
    glutWireCube(0.1);
    glEnable(GL_LIGHTING);
    glPopMatrix();

    glutSolidTorus(0.275, 0.85, 8, 15);
    glPopMatrix();
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
```

```
    glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
{
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN) {
                spin = (spin + 30) % 360;
                glutPostRedisplay();
            }
            break;
        default:
            break;
    }
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutMainLoop();
    return 0;
}
```

Перемещение источника света вместе с точкой обзора

Чтобы создать источник света, который перемещается вместе с точкой обзора, нужно установить его позицию до видового преобразования. Тогда видовое преобразование будет одинаково влиять и на источник света, и на точку обзора. Помните, что положение источника света сохраняется в видовых координатах — это один из небольшого числа случаев, когда видовые координаты весьма важны. В листинге 5.7 положение источника света задается подпрограммой `init()`, которая сохраняет его в видовых координатах (0, 0, 0). Другими словами, свет испускается из оптической системы фотокамеры.

Листинг 5.7. Источник света, перемещающийся вместе с точкой обзора

```
GLfloat light_position[]  = { 0.0,  0.0,  0.0,  1.0 };

    glViewport(0, 0, (GLint) w, (GLint) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Теперь, если точка обзора переместится, источник света последует за ней, с сохранением дистанции (0, 0, 0) относительно глаза. Расширим программу в листинге 5.7 глобальными переменными *ex*, *ey*, *ez*, управляющими положением точки обзора, и *upx*, *upy*, *upz*, указывающими вектор ориентации. Процедура `display()`, вызываемая из цикла обработки сообщений для перерисовки сцены, может выглядеть следующим образом:

```
static GLdouble ex, ey, ez, upx, upy, upz;

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
        gluLookAt(ex, ey, ez, 0.0, 0.0, 0.0, upx, upy, upz);
        glutSolidTorus(0.275, 0.85, 8, 15);
    glPopMatrix();
    glFlush();
}
```

Когда перерисовывается освещенный тор, и источник света, и точка обзора перемещаются в одно и то же место. Благодаря изменению данных, передаваемых функции `gluLookAt()`, и перемещению позиции наблюдения, объект никогда не будет темным, поскольку он всегда освещается с позиции наблюдателя. Даже если не изменять координаты позиции источника света явно, он все равно будет перемещаться, поскольку изменяется видовая система координат.

Этот метод может быть весьма полезен при имитации фонаря на каске шахтера. Другим примером может быть свеча или лампа, которую несут в руке. Позиция источника света, задаваемая вызовом функции `glLightfv(GL_LIGHTi, GL_POSITION, position)`, по смыслу будет соответствовать указанию расстояний *x*, *y* и *z* источника от наблюдателя. Затем при изменении положения наблюдателя источник света будет оставаться на том же относительном расстоянии.

Упражнение

Измените программу в листинге 5.6 следующим образом:

- заставьте источник света перемещаться, а не вращаться вокруг объекта. Подсказка: в процедуре `display()` вместо первого вызова функции `glRotated()` используйте `glTranslated()`, и выберите подходящее значение вместо *spin*;
- измените ослабление так, чтобы интенсивность света уменьшалась при удалении от объекта. Подсказка: для установки нужного ослабления добавьте вызов функции `glLight*()`.

Пособие от Нэйта Робинса по освещению

Если у вас есть набор обучающих программ Нэйта Робинса, запустите пособие по позиционированию источников света. (О том, где взять эти программы, написано в разделе «Набор обучающих программ по OpenGL от Нэйта Робинса».) С помощью средства обучения можно поэкспериментировать с положением источника света и оценить влияние модельно-видовой матрицы.

Выбор модели освещения

В OpenGL понятие модели освещения слагается из четырех компонентов:

- интенсивности общего фонового освещения;
- положения точки обзора — локального по отношению к сцене или бесконечно удаленного;
- расчета освещенности для лицевых и обратных граней объектов;
- того, должен или нет отраженный цвет быть отделен от фонового и рассеянного и наложен на объект после операций текстурирования.

Этот раздел объясняет способы задания модели освещения. Здесь также обсуждается, как включать освещение, — то есть как сказать OpenGL, что она должна выполнять расчет освещенности.

Для задания всех параметров модели освещения используется команда `glLightModel*()`. Функция `glLightModel*()` имеет два аргумента: имя параметра модели освещения и значение для этого параметра.

void `glLightModel`{if}(GLenum *pname*, *TYPE param*);
void `glLightModel`{if}v(GLenum *pname*, *TYPE *param*);
Устанавливает свойство модели освещения, идентифицируемое аргументом *pname* (табл. 5.2). Аргумент *param* — это собственно значение свойства; а если используется векторная версия команды — указатель на набор значений. Невекторная версия команды используется только для установки одиночных параметров и неприменима для `GL_LIGHT_MODEL_AMBIENT`.

Таблица 5.2. Значения по умолчанию для аргумента pname функции glLightModel*()

Символическое имя	Значение по умолчанию	Описание
GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Фоновая RGBA-интенсивность всей сцены
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0 или GL_FALSE	Способ вычисления углов зеркального отражения (точка обзора локальная или в бесконечности)
GL_LIGHT_MODEL_TWO_SIDE	0.0 или GL_FALSE	Выбор между односторонним и двухсторонним освещением
GL_LIGHT_MODEL_COLOR_CONTROL	GL_SINGLE_COLOR	Вычисляется ли отраженный цвет отдельно от фонового и рассеянного

Общее фоновое освещение

Как обсуждалось ранее, каждый источник света может добавлять к сцене фоновое освещение. Кроме того, может присутствовать другой фоновый свет, не принадлежащий никакому конкретному источнику. Чтобы задать RGBA-интенсивность такого общего рассеянного света, используется константа `GL_LIGHT_MODEL_AMBIENT`:

```
Glfloat lmodel_ambient[]={0.2,0.2,0.2,1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

В этом примере значения для lmodel_ambient совпадают со значениями по умолчанию для GL_LIGHT_MODEL_AMBIENT. Поскольку эти числа задают небольшое количество белого фонового освещения, объекты сцены видны всегда, в отсутствие каких-либо дополнительных источников света. На цветном рисунке 14 показано изменение объекта вследствие изменения общей освещенности.

Локальная или бесконечно удаленная точка обзора

Положение точки обзора влияет на расчет отблеска, получаемого вследствие зеркального отражения. Точнее, интенсивность блика в конкретной вершине зависит от вектора нормали в этой вершине, направления из вершины к источнику света и направления из вершины к точке обзора. Имейте в виду, что на самом деле точка обзора не перемещается никакими командами, связанными с освещением (необходимо изменить преобразование проецирования, как описано в разделе «Преобразования проецирования» главы 3), вместо этого выполняются разные расчеты освещения на основании предположения о том, что она перемещается.

В случае бесконечно удаленной точки обзора направления из нее ко всем вершинам считаются одинаковыми. Локальная точка обзора позволяет получать более реалистичный результат, но, поскольку должны быть рассчитаны направления между ней и каждой вершиной, общее быстродействие уменьшается. По умолчанию используется бесконечно удаленная точка обзора. Ее можно сменить на локальную следующим вызовом:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

Данная команда помещает точку обзора в позицию (0, 0, 0) в видовых координатах. Для переключения обратно к бесконечно удаленной точке обзора используйте GL_FALSE.

Двухстороннее освещение

Расчет освещенности производится для всех многоугольников, независимо от того, являются они лицевыми или обратными. Поскольку обычно источники света настраиваются в первую очередь в расчете на лицевые многоугольники, обратные могут быть освещены неверно. В листинге 5.1, где в качестве объекта фигурирует сфера, всегда видны только лицевые грани, поскольку именно они находятся «снаружи» сферы. В данном случае неважно, как выглядят задние. Однако если часть сферы будет отсечена и ее внутренняя поверхность станет видимой, возможно, эту поверхность потребуется обработать в соответствии с заданными условиями освещения; и возможно, также потребуется задать для обратной поверхности иные свойства материала, чем для лицевой. При включении двухстороннего освещения командой:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

OpenGL обращает нормали для задних многоугольников; обычно это означает, что векторы нормалей для видимых обратных и лицевых многоугольников направлены в сторону, а не от наблюдателя. В результате все многоугольники освещаются правильно. Однако дополнительные операции по обеспечению двухсто-

роннего освещения делают его более медленным, чем принятое по умолчанию одностороннее освещение.

Чтобы выключить двухстороннее освещение, используйте GL_FALSE в предыдущем вызове. Также можно управлять тем, какие грани OpenGL считает лицевыми, с помощью команды glFrontFace(). (Для получения дополнительной информации см. раздел «Обращение и отбраковка поверхностей многоугольников» главы 2.)

Вторичный отраженный цвет

При обычном расчете освещенности фоновая, рассеянная, отраженная и излучающая составляющие источника света вычисляются и просто складываются вместе. По умолчанию наложение текстуры производится после применения освещенности, в результате зеркальный блик может выглядеть мутным, а текстура может быть искажена. Чтобы отложить применение отраженного цвета на время после текстурирования, вызовите команду

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

В этом режиме расчет освещенности производится для двух цветов на каждую вершину: первичного, то есть суммы всех неотраженных вкладов от всех источников света, и вторичного — суммы всех отраженных вкладов. Во время текстурирования с цветами текстуры комбинируется только первичный цвет. После операции текстурирования к комбинации первичного и текстурных цветов добавляется вторичный цвет (см. раздел «Применение вторичного цвета после текстурирования» главы 9). Объекты, освещенные и текстурированные с помощью отделенного отраженного цвета, как правило, имеют более рельефный зеркальный блик.

Чтобы вернуться к методу расчета освещенности по умолчанию, выполните команду:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
```

Первичный цвет снова будет включать в себя вклады от всех компонентов: фонового, рассеянного, отраженного и излучаемого цвета материала; а после текстурирования в освещенность ничего более не добавляется. (Если текстурирования нет, отделять отраженный цвет от остальных компонентов не требуется.)

Включение источников света

OpenGL предоставляет право включения (или выключения) механизма расчета освещенности вам. Если данный механизм не активен, текущая вершина просто отображается текущим цветом, и никаких вычислений, касающихся нормалей, источников света, модели освещения и свойств материала, не производится. Расчет освещенности включается командой:

```
glEnable(GL_LIGHTING);
```

и выключается командой

```
glDisable(GL_LIGHTING);
```

Кроме того, необходимо после задания параметров каждого источника света включать и его, как это сделано в листинге 5.1, где используется только один источник света — GL_LIGHT0:

```
glEnable(GL_LIGHT0);
```

Задание свойств материала

Вы уже знаете, как создать источник света с определенными характеристиками и как описать нужную модель освещения. В этом разделе рассказывается, как установить свойства материала для объектов сцены: фоновый, рассеянный и отраженный цвета, уровень блеска и цвет испускаемого объекта света. (Информация об уравнениях, используемых при расчете освещения и вычислении свойств материала, приведена в разделе «Математика освещения».) Большинство свойств материала концептуально похожи на те, которые обсуждались в разделах, посвященных созданию источников света. Механизм их указания также подобен установке параметров источника света за исключением разве что названия команды — glMaterial*().

void glMaterial{if}(GLenum *face*, GLenum *pname*, *TYPE param*);
void glMaterial{if}v(GLenum *face*, GLenum *pname*, *TYPE *param*);
Задает текущее свойство материала для использования при расчете освещенности. Аргумент *face* может принимать значения GL_FRONT, GL_BACK или GL_FRONT_AND_BACK в соответствии с тем, к каким поверхностям объекта применяется это свойство. Устанавливаемое свойство материала идентифицируется именем *pname*, а его значение передается в аргументе *param* в виде указателя на вектор значений или отдельным значением (в невекторном варианте). Невекторная версия команды работает только для параметра GL_SHININESS. Возможные значения *pname* перечислены в табл. 5.3. Заметьте, что константа GL_AMBIENT_AND_DIFFUSE позволяет одновременно установить фоновый и рассеянный цвета материала в одно и то же RGBA-значение.

Таблица 5.3. Значения по умолчанию аргумента pname функции glMaterial*()

Имя параметра	Значение по умолчанию	Описание
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Фоновый цвет материала
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Рассеянный цвет материала
GL_AMBIENT_AND_DIFFUSE	–	Фоновый и рассеянный цвета материала
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	Отраженный цвет материала
GL_SHININESS	0.0	Коэффициент зеркального отражения
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	Излучаемый цвет материала
GL_COLOR_INDEXES	(0, 1, 1)	Индексы фонового, рассеянного и отраженного цветов

Как было описано ранее в разделе «Выбор модели освещения», если обратные грани могут быть видимыми, допускается освещенность для лицевых и обратных многоугольников объекта вычислять раздельно, с помощью аргумента *face* функции `glMaterial*()`. На цветном рис. 14 изображен объект, нарисованный с применением разных свойств материала для его лицевых и обратных граней.

Для того чтобы иметь представление относительно влияния свойств материалов на результат, посмотрите на цветной рисунок 16. На нем изображено несколько экземпляров одного и того же объекта, нарисованных с применением разных свойств материала. В последующих разделах обсуждаются примененные при рисовании этих сфер свойства.

Помните, что большинство свойств материалов, устанавливаемых с помощью функции `glMaterial*()`, являются цветами (R, G, B, A). Независимо от значений альфа-компонентов других цветов, в любой отдельно взятой вершине альфа-значение представляет собой только значение рассеянного цвета материала (определенное для параметра `GL_DIFFUSE` функцией `glMaterial*()`, как описано в следующем разделе). (Для получения полной информации о значениях альфа-компонентов см. раздел «Смешивание» главы 6.) Кроме того, RGBA-свойства материалов неприменимы в режиме индексации цвета. (См. раздел «Освещение в режиме индексации цвета»).

Рассеянное и фоновое отражение

Параметры `GL_DIFFUSE` и `GL_AMBIENT`, устанавливаемые командой `glMaterial*()`, влияют на цвета рассеянного и фонового света, отражаемого объектом. Рассеянное отражение играет наиболее важную роль в восприятии цвета наблюдателем. На наш глазомер оказывают влияние цвет падающего рассеянного света и угол между этим светом и вектором нормали к поверхности. (Диффузное отражение наиболее заметно, если рассеянный свет падает перпендикулярно поверхности.) Положение точки обзора вообще никак не чувствуется на диффузном отражении.

Фоновое отражение одинаково воздействует на все цвета объекта. Рассеянное отражение наиболее заметно на непосредственно освещаемых частях объекта, в то время как фоновое более проявляется в местах, которые напрямую не освещаются. Суммарное фоновое отражение объекта складывается из общего фонового освещения и фонового света, излучаемого конкретными источниками. Так же как и рассеянное, фоновое отражение не зависит от точки обзора.

Для объектов реального мира рассеянное и фоновое отражение от одного и того же объекта, как правило, имеет одинаковый цвет. По этой причине OpenGL предоставляет упрощенный способ для задания обоих параметров материала в команде `glMaterial*()`:

```
GLfloat mat_amb_diff[]={ 0.1, 0.5, 0.8, 1.0 }
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat_amb_diff);
```

В этом примере RGBA-цвет (0.1, 0.5, 0.8, 1.0) — темно-синий — задается в качестве текущих цветов фонового и рассеянного отражения для лицевых и обратных поверхностей многоугольников.

На цветном рисунке 16 первый ряд сфер не имеет общего отражения (0.0, 0.0, 0.0, 0.0), а во втором ряду оно отчетливо выражено (0.7, 0.7, 0.7, 1.0).

Зеркальное отражение

Зеркальное отражение от объекта производит блики. В отличие от фонового и рассеянного отражения, то, что видит наблюдатель здесь, зависит от положения точки обзора. Представим, что вы смотрите на металлический шарик под солнечным светом. При движении головы блик, создаваемый солнечным светом на шарике, до некоторой степени будет передвигаться вместе с вами. Однако если вы слишком далеко закинете голову, блик вообще исчезнет.

OpenGL позволяет управлять воздействием, оказываемым материалом на отраженный свет (с помощью параметра `GL_SPECULAR`), и контролировать площадь и яркость блика (с помощью параметра `GL_SHININESS`). Параметр `GL_SHININESS` принимает значения в диапазоне [0.0, 128.0]: чем больше значение, тем меньше и ярче (более сфокусирован) зеркальный блик. (Для получения более подробной информации см. раздел «Математика освещения» этой главы.)

На цветном рисунке 16 сферы в первом столбце не имеют зеркального отражения. Во втором столбце для параметров `GL_SPECULAR` и `GL_SHININESS` заданы следующие значения:

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[] = { 5.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
```

В третьем столбце значение параметра `GL_SHININESS` увеличено до 100.0.

Излучение

Задавая RGBA-цвет для параметра `GL_EMISSION`, можно изобразить объект так, как будто он испускает свет данного цвета. Поскольку большинство реальных объектов (помимо источников света) сами не излучают свет, этот эффект используется в основном для имитации ламп и других источников света. На цветном рисунке 16 сферам в четвертом столбце посредством параметра `GL_EMISSION` придан красновато-серый цвет:

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

Хотя кажется, что сферы несколько светятся, на самом деле они не действуют как источники света. Для того чтобы это было так, необходимо создать источник света и поместить его в то же место, что и сферу.

Изменение свойств материала

В примере, приведенном в листинге 5.1, используются одни и те же свойства материала для всех вершин единственного объекта сцены (сферы). В других ситуациях может потребоваться ассоциировать различные свойства материала с раз-

личными вершинами одного и того же объекта. Более вероятно, что сцена будет скомпонована из нескольких объектов с различными свойствами материала. Например, это может быть такое изображение, как на цветном рисунке 16. На рисунке представлены 12 различных объектов (все являются сферами), каждый — с собственными свойствами. Листинг 5.8 содержит часть кода подпрограммы `display()`.

Листинг 5.8. Различные свойства материала: material.c

```
   GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
   GLfloat mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0 };
   GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat no_shininess[] = { 0.0 };
   GLfloat low_shininess[] = { 5.0 };
   GLfloat high_shininess[] = { 100.0 };
   GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/*  Рисуем сферу на пересечении первого столбца и первого ряда
 *  с рассеянным диффузным цветом, без фонового или зеркального цвета
 */
   glPushMatrix();
   glTranslatef(-3.75, 3.0, 0.0);
   glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
   glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
   glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
   glutSolidSphere(1.0, 16, 16);
   glPopMatrix();

/*  Рисуем сферу на пересечении первого ряда и второго столбца
 *  с диффузным и зеркальным цветом, с небольшим блеском, без фонового цвета
 */
   glPushMatrix();
   glTranslatef(-1.25, 3.0, 0.0);
   glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
   glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
   glutSolidSphere(1.0, 16, 16); glPopMatrix();

/*  Рисуем сферу на пересечении первого ряда и третьего столбца
 *  с рассеянным и зеркальным светом, ярким блеском, без фонового цвета
 */
   glPushMatrix();
   glTranslatef(1.25, 3.0, 0.0);
   glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
```

продолжение ⇩

Листинг 5.8 (*продолжение*)

```
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    glutSolidSphere(1.0, 16, 16); glPopMatrix();

/*  Рисуем сферу на пересечении первого ряда и четвертого столбца
 *  с диффузным светом, излучением, без фонового и зеркального цвета
 */
    glPushMatrix();
    glTranslatef(3.75, 3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
    glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
    glutSolidSphere(1.0, 16, 16); glPopMatrix();
```

Как видно, функция `glMaterial*()` вызывается несколько раз для установки требуемого свойства материала для каждой сферы. Заметьте, что эту функцию необходимо вызывать только для тех параметров, которые изменяются. Вторая, третья и четвертая сферы рисуются с одинаковыми «фоновым» и «рассеянным» свойствами материала, так что нет смысла их переопределять. Поскольку существуют некоторые временные затраты на выполнение функции `glMaterial*()`, пример в листинге 5.8 может быть переписан для минимизации изменения свойств материала.

Пособие от Нэйта Робинса по свойствам материала

Если у вас есть набор обучающих программ от Нэйта Робинса, запустите пособие по свойствам материала. Используя пособие можно экспериментировать со свойствами материала, включая фоновый, рассеянный и отраженный цвета и блеск.

Режим цвета материала

Другая техника снижения затрат производительности заключается в использовании для изменения свойств материала функции `glColorMaterial()`.

void `glColorMaterial`(GLenum *face*, GLenum *mode*);

Заставляет свойство (или свойства) материала, заданное аргументом *mode*, при обработке поверхности (или поверхностей), заданной аргументом *face*, все время принимать значение текущего цвета. При изменении текущего цвета (командой `glColor*()`) указанные свойства материала также незамедлительно меняются. Аргумент *face* может принимать значения `GL_FRONT`, `GL_BACK` или `GL_FRONT_AND_BACK` (по умолчанию). Аргумент *mode* выбирается из списка `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_AMBIENT_AND_DIFFUSE` (по умолчанию) или `GL_EMISSION`. В каждый момент времени активным является только один из режимов. Функция `glColorMaterial()` не работает при использовании освещения в режиме индексирования цветов.

Заметим, что функция `glColorMaterial()` задает две независимые величины: первая определяет, какие поверхности изменяются с изменением текущего цвета, а вторая — какие свойства материала или поверхности изменяются. Однако OpenGL не поддерживает отдельные значения *mode* для каждой поверхности!

Если по завершении работы glColorMaterial() вызвать функцию glEnable() со значением аргумента GL_COLOR_MATERIAL, это позволит в последующем изменять текущий цвет командой glColor*() (а также другие свойства материала командой glMaterial*()) в соответствии с потребностями рисования:

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
/* теперь функция glColor* изменяет диффузный цвет */
glColor3f(0.2, 0.5, 0.8);
/* здесь рисуем объекты */
glColorMaterial(GL_FRONT, GL_SPECULAR);
/* функция glColor* больше не меняет диффузный цвет,
 * теперь она изменяет отраженный цвет */
glColor3f(0.9, 0.0, 0.2);
/* здесь рисуем другие объекты */
glDisable(GL_COLOR_MATERIAL);
```

Используйте функцию glColorMaterial() в тех случаях, когда для большинства вершин на сцене требуется подстраивать значение лишь одного параметра материала. Если изменяемых параметров несколько, как это было в случае с цветным рисунком 16, применяйте функцию glMaterial*(). При исчерпании потребности в функциональности glColorMaterial() обязательно убедитесь, что данный механизм отключен командой glDisable(). Тем самым вы избежите изменений свойств материала «по неустановленным причинам», а также снизите затраты в плане производительности. Быстродействие приложения при работе с механизмом glColorMaterial() зависит от реализации OpenGL. Некоторые реализации могут оптимизировать обработку вершин с целью ускоренного изменения свойств материала на основе текущего цвета.

Пример, приведенный в листинге 5.9, представляет собой интерактивную программу, использующую функцию glColorMaterial() для изменения свойств материала. Нажатие любой из трех кнопок мыши изменяет рассеянный цвет.

Листинг 5.9. Применение функции glColorMaterial(): colormat.c

```
GLfloat diffuseMaterial[4] = { 0.5,  0.5, 0.5, 1.0 };

void init(void)
{
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_SMOOTH);
   glEnable(GL_DEPTH_TEST);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMaterial);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glColorMaterial(GL_FRONT, GL_DIFFUSE);
   glEnable(GL_COLOR_MATERIAL);
}
```

продолжение ⇩

Листинг 5.9 (*продолжение*)

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSolidSphere(1.0, 20, 16);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho(-1.5*(GLfloat)w/(GLfloat)h, 1.5*(GLfloat)w/(GLfloat)h, -1.5,
1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void mouse(int button, int state, int x, int y)
{
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN) {    /*  меняем красный */
                diffuseMaterial[0] += 0.1;
                if (diffuseMaterial[0] > 1.0)
                    diffuseMaterial[0] = 0.0;
                glColor4fv(diffuseMaterial);
                glutPostRedisplay();
            }
            break;
        case GLUT_MIDDLE_BUTTON:
            if (state == GLUT_DOWN) {      /*  меняем зеленый */
                diffuseMaterial[1] += 0.1;
                if (diffuseMaterial[1] > 1.0)
                    diffuseMaterial[1] = 0.0;
                glColor4fv(diffuseMaterial);
                glutPostRedisplay();
            }
            break;
        case GLUT_RIGHT_BUTTON:
            if (state == GLUT_DOWN) {     /*  меняем синий */
                diffuseMaterial[2] += 0.1;
                if (diffuseMaterial[2] > 1.0)
                    diffuseMaterial[2] = 0.0;
                glColor4fv(diffuseMaterial);
                glutPostRedisplay();
            }
            break;
```

Листинг 5.9 (*продолжение*)

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSolidSphere(1.0, 20, 16);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho(-1.5*(GLfloat)w/(GLfloat)h, 1.5*(GLfloat)w/(GLfloat)h, -1.5,
1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void mouse(int button, int state, int x, int y)
{
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN) {    /*  меняем красный */
                diffuseMaterial[0] += 0.1;
                if (diffuseMaterial[0] > 1.0)
                    diffuseMaterial[0] = 0.0;
                glColor4fv(diffuseMaterial);
                glutPostRedisplay();
            }
            break;
        case GLUT_MIDDLE_BUTTON:
            if (state == GLUT_DOWN) {      /*  меняем зеленый */
                diffuseMaterial[1] += 0.1;
                if (diffuseMaterial[1] > 1.0)
                    diffuseMaterial[1] = 0.0;
                glColor4fv(diffuseMaterial);
                glutPostRedisplay();
            }
            break;
        case GLUT_RIGHT_BUTTON:
            if (state == GLUT_DOWN) {     /*  меняем синий */
                diffuseMaterial[2] += 0.1;
                if (diffuseMaterial[2] > 1.0)
                    diffuseMaterial[2] = 0.0;
                glColor4fv(diffuseMaterial);
                glutPostRedisplay();
            }
            break;
```

```
        default:
            break;
    }
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[ 0 ]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutMainLoop();
    return 0;
}
```

Упражнение

Модифицируйте программу в листинге 5.8 следующим образом:

- Измените общее освещение сцены. Подсказка: используйте параметр `GL_LIGHT_MODEL_AMBIENT`.
- Измените рассеянный, фоновый и отраженный цвета, блеск и цвет излучения. Подсказка: используйте функцию `glMaterial*()`, но не злоупотребляйте ею.
- Примените двухсторонний материал и добавьте пользовательские плоскости отсечения таким образом, чтобы видеть сферы «изнутри» и «снаружи». (См. раздел «Дополнительные плоскости отсечения» главы 3.) Подсказка: включите двухстороннее освещение с помощью `GL_LIGHT_MODEL_TWO_SIDE`, установите нужные свойства материала и добавьте плоскости отсечения.
- Вместо вызова функции `glMaterialfv()` для установки значения параметра `GL_DIFFUSE` используйте более эффективный вызов функции `glColorMaterial()` с сохранением той же освещенности сцены.

Математика освещения

В этом разделе представлены уравнения, привлекаемые OpenGL для расчетов освещенности, чтобы задать цвета в режиме RGBA. (См. раздел «Математика освещения в режиме индексации цвета» для получения информации о соответствующих вычислениях.) Вам не нужно читать этот раздел, если вы хотите добиться определенного освещения экспериментальным путем. Даже со знанием этих уравнений вам наверняка придется экспериментировать, но вы будете иметь лучшее представление о том, как значения определенных параметров влияют на цвет вершины. Помните, что если освещение выключено, цвет вершины просто принимает значение текущего цвета, если же освещение включено, расчеты, связанные с ним, производятся в видовых координатах.

В следующих уравнениях математические расчеты производятся отдельно для R-, G- и B-компонентов цвета. Таким образом, если в уравнении складываются три величины, следует понимать, что значения R, G и B складываются друг с другом попарно, формируя результирующий RGB-цвет ($R_1 + R_2 + R_3$, $G_1 + G_2 + G_3$, $B_1 + B_2 + B_3$). Когда же три величины перемножаются, результатом будет цвет ($R_1R_2R_3$,$G_1G_2G_3$, $B_1B_2B_3$). (Напомним, что значение альфа-компонента цвета вершины равно значению альфа-компонента цвета рассеивания материала этой вершины.)

Цвет освещенной вершины вычисляется следующим образом:

цвет вершины = излучаемый цвет материала вершины +
+ общий фоновый цвет, масштабированный согласно значению фонового цвета материала вершины +
+ фоновые, рассеянные и отраженные составляющие всех источников света, ослабленные должным образом

После расчета освещенности значения цветов приводятся (в режиме RGBA) к диапазону [0, 1].

Не забудьте, что OpenGL не принимает в расчет возможность того, что один объект может загораживать свет для другого и, как следствие, автоматически не отбрасываются тени. (См. раздел «Тени» главы 14 для получения информации о технологии создания теней.) Также учтите, что в OpenGL самосветящиеся объекты не освещают другие объекты.

Излучение материала

Об излучаемой составляющей $\text{emission}_{\text{material}}$ достаточно сказать только, что это RGB-значение, присваиваемое параметру `GL_EMISSION`.

Масштабированный общий фоновый свет

Второй составляющей уравнения является значение, вычисленное путем перемножения общего фонового света (определенного параметром `GL_LIGHT_MODEL_AMBIENT`) на значение фонового цвета материала (`GL_AMBIENT`, заданного функцией `glMaterial*()`):

$$\text{ambient}_{\text{light model}} \times \text{ambient}_{\text{material}}$$

Каждая пара значений R, G и B для этих двух параметров перемножается независимо, формируя результирующий RGB-цвет данной составляющей: (R_1R_2, G_1G_2, B_1B_2).

Вклады от источников света

Каждый источник света может вносить свой пай в освещенность вершины, и все эти взносы от всех источников суммируются. Далее приведено уравнение, по которому выполняется вычисление вклада от каждого источника света:

вклад = коэффициент ослабления × влияние прожектора ×
× (фоновая составляющая + рассеянная составляющая +
+ отраженная составляющая)

Коэффициент ослабления

Фактор ослабления был рассмотрен в разделе «Положение и ослабление». Формулу мы приведем повторно:

$$\text{коэффициент ослабления} = \frac{1}{k_c + k_l d + k_q d^2},$$

где d — расстояние от источника света до вершины, k_c — GL_CONSTANT_ATTENUATION, k_l — GL_LINEAR_ATTENUATION, k_q — GL_QUADRATIC_ATTENUATION.

Если источник света является направленным, коэффициент затухания равен 1.

Эффект прожектора

Эффект прожектора (spotlight effect — в формуле) выражается в добавлении одного из трех значений в зависимости от того, является ли в действительности источник света прожектором и находится ли вершина внутри или снаружи конуса излучения прожектора:

- 1, если источник не является прожектором (параметр GL_SPOT_CUTOFF равен 180.0);
- 0, если источник света является прожектором, но вершина лежит вне его конуса излучения;
- $(\max\{\mathtt{v} \cdot \mathtt{d}, 0\})^{\text{GL_SPOT_EXPONENT}}$, где $\mathtt{v} = (v_x, v_y, v_z)$ — единичный вектор из прожектора (GL_POSITION) в вершину, $\mathtt{d} = (d_x, d_y, d_z)$ — ориентация прожектора (GL_SPOT_DIRECTION) при условии, что источник света является прожектором и вершина находится внутри его конуса излучения.

  Скалярное произведение векторов v и d меняется в соответствии с косинусом угла между ними; таким образом, объекты на линии освещения получают максимум освещенности, на объекты вне этой линии света поступает меньше, в соответствии с косинусом угла.

Чтобы выяснить, лежит ли вершина внутри конуса излучения прожектора, OpenGL вычисляет значение $(\max\{\mathtt{v} \cdot \mathtt{d}, 0\})$, где v и d определены аналогично. Если эта величина меньше косинуса угла при вершине конуса излучения прожектора (GL_SPOT_CUTOFF) — вершина находится вне конуса излучения, в противном случае она попадает внутрь конуса.

Фоновая составляющая

Этот член уравнения представляет собой фоновый цвет источника, умноженный на фоновый цвет материала вершины:

$$\text{ambient}_{\text{light}} \times \text{ambient}_{\text{material}}.$$

Рассеянная составляющая

Диффузный компонент необходимо принимать в расчет независимо от того, падает ли свет непосредственно на вершину или нет. Он вычисляется как

$$(\max\{\mathtt{L}\cdot\mathtt{n},0\})\times \mathrm{diffuse}_{\mathrm{light}}\times \mathrm{diffuse}_{\mathrm{material}},$$

где $\mathtt{L}=(L_x, L_y, L_z)$ — единичный вектор, указывающий из вершины в направлении положения источника света (GL_POSITION), $\mathtt{n}=(n_x, n_y, n_z)$ — вектор нормали единичной длины к вершине.

Отраженная составляющая

Зеркальная составляющая также зависит от того, падает ли свет непосредственно на вершину. Когда $\mathtt{L}\cdot\mathtt{n}$ меньше или равно 0, это говорит, что в вершине нет отраженной составляющей. (Если эта величина отрицательна, источник света находится с другой стороны поверхности.) Если же отраженная составляющая присутствует, она зависит от следующего:

- нормали к вершине единичной длины (n_x, n_y, n_z);
- суммы двух векторов единичной длины: вектора между вершиной и источником света (положением или направлением) и вектора, соединяющего вершину и точку наблюдения (в случае если выбран режим GL_LIGHT_MODEL_LOCAL_VIEWER; если это не так, в качестве второго вектора суммы берется (0, 0, 1)). Эта векторная сумма нормализуется (посредством деления каждого компонента на длину получившегося вектора). В результате получаем $\mathtt{s}=(s_x, s_y, s_z)$;
- показателя степени shininess для учета блеска (GL_SHININESS);
- цвета зеркального компонента источника света ($\mathtt{GL_SPECULAR}_{\mathrm{light}}$);
- цвета зеркального компонента материала ($\mathtt{GL_SPECULAR}_{\mathrm{material}}$).

На основе приведенных выше определений OpenGL рассчитывает отраженную составляющую по следующей формуле:

$$(\max\{\mathtt{s}\cdot\mathtt{n},0\})^{\mathrm{shininess}}\times \mathrm{specular}_{\mathrm{light}}\times \mathrm{specular}_{\mathrm{material}}.$$

Однако если $\mathtt{L}\cdot\mathtt{n}=0$, отраженная составляющая также равна 0.

Все вместе

Теперь, отталкиваясь от определений, сформулированных выше, запишем уравнение освещенности в режиме RGBA в окончательном виде:

$$\begin{aligned}&\text{цвет вершины} = \mathrm{emission}_{\mathrm{material}} + \mathrm{ambient}_{\mathrm{light\ model}}\times \mathrm{ambient}_{\mathrm{material}} + \\ &+\sum_{i=0}^{n-1}\left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i \times (\text{spotlight effect})_i \times [\mathrm{ambient}_{\mathrm{light}}\times \mathrm{ambient}_{\mathrm{material}} + \\ &+ (\max\{\mathtt{L}\cdot\mathtt{n},0\})\times \mathrm{diffuse}_{\mathrm{light}}\times \mathrm{diffuse}_{\mathrm{material}} + \\ &+ (\max\{\mathtt{s}\cdot\mathtt{n},0\})^{\mathrm{shininess}}\times \mathrm{specular}_{\mathrm{light}}\times \mathrm{specular}_{\mathrm{material}}]_i\end{aligned}$$

Вторичный отраженный цвет

Если для некоторой модели освещения цветовой режим установлен в значение GL_SEPARATE_SPECULAR_COLOR, то для каждой вершины вычисляются два цвета (первичный и вторичный) в соответствии со следующими формулами:

первичный цвет = излучаемый свет материала вершины + общее фоновое освещение, умноженное на фоновую составляющую цвета материала вершины + фоновая и диффузная составляющие всех источников света, ослабленные соответствующим образом;

вторичный цвет = отраженные составляющие всех источников света, ослабленные соответствующим образом.

Следующие два уравнения применяются OpenGL для расчета первичного и вторичного цветов:

$$\text{первичный цвет} = \text{emission}_{\text{material}} + \text{ambient}_{\text{light model}} \times \text{ambient}_{\text{material}} + \sum_{i=0}^{n-1}\left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i \times (\text{spotlight effect})_i \times \times [\text{ambient}_{\text{light}} \times \text{ambient}_{\text{material}} + (\max\{L \cdot n, 0\}) \times \text{diffuse}_{\text{light}} \times \text{diffuse}_{\text{material}}]_i;$$

$$\text{вторичный цвет} = \sum_{i=0}^{n-1}\left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i \times (\text{spotlight effect})_i \times \times [(\max\{s \cdot n, 0\})^{\text{shininess}} \times \text{specular}_{\text{light}} \times \text{specular}_{\text{material}}]_i.$$

В процессе текстурирования с цветами текстуры комбинируется только первичный цвет. После текстурирования вторичный цвет добавляется к комбинации первичного цвета и цвета текстуры.

Освещение в режиме индексации цвета

В режиме индексации цвета параметры, задаваемые RGBA-величинами, либо не оказывают влияния, либо интерпретируются особым образом. Поскольку в режиме индексации цвета намного сложнее достигнуть определенных эффектов, связанных с освещением, следует применять режим RGBA везде, где это возможно. Фактически в режиме индексированных цветов параметры источника света, модели освещения или свойств материала задаются тоже в форме RGBA-значений. Используемые параметры источника света — `GL_DIFFUSE` и `GL_SPECULAR`, а также значение блеска `GL_SHININESS`. С помощью параметров `GL_DIFFUSE` и `GL_SPECULAR` (d_l и s_l соответственно) вычисляются интенсивности рассеянной и отраженной составляющих (d_{cl} и s_{cl}):

$$d_{cl} = 0{,}30\text{R}(d_l) + 0{,}59\text{G}(d_l) + 0{,}11\text{B}(d_l);$$

$$s_{cl} = 0{,}30\text{R}(s_l) + 0{,}59\text{G}(s_l) + 0{,}11\text{B}(s_l),$$

где $\text{R}(x)$, $\text{G}(x)$ и $\text{B}(x)$ представляют красный, зеленый и синий компоненты цвета x, соответственно. Весовые коэффициенты 0.30, 0.59 и 0.11 отражают степень восприятия компонентов цвета человеческим глазом — наши глаза наиболее чувствительны к зеленому цвету и наименее восприимчивы к синему.

Для задания цвета материала в режиме индексации цвета `glMaterial*()` вызывается со специальным параметром `GL_COLOR_INDEXES`:

```
GLufloat mat_colormap[] = { 16.0, 47.0, 79.0 };
glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, mat_colormap);
```

Три числа карты задают соответственно индексы фонового, рассеянного и отраженного цветов материала. Другими словами, OpenGL считает цвет с индексом, заданным в первом числе (в данном случае 16.0), — чистым фоновым цветом, цвет с индексом, заданным во втором числе (47.0), — чистым диффузным цветом, а цвет с индексом, заданным в третьем числе (79.0), — чистым отраженным цветом. (По умолчанию фоновый цвет имеет индекс 0.0, а рассеянный и отраженный цвета — индекс 1.0. Заметим, что команда `glColorMaterial()` не работает в режиме индексирования цветов.)

Во время рисования сцены OpenGL использует цвета, ассоциированные с индексами данного диапазона, для закраски объектов сцены. Таким образом, необходимо верно построить цветовую таблицу между этими индексами (в данном примере между 16 и 47 и затем между 47 и 79). Обычно цветовая карта строится из близких RGB-значений, но достижение определенных эффектов оправдывает и другие соображения. Далее приведен пример «плавной» цветовой карты, начинающейся с черного фонового цвета, на пути к белому отраженному цвету минующей фиолетовый рассеянный цвет:

```
for (i=0;i<32;i++){
    glutSetColor(16 + i, 1.0 * (i/32.0), 0.0, 1.0 * (i/32.0));
    glutSetColor(48 + i, 1.0, 1.0 * (i/32.0), 1.0);
}
```

Функция `glutSetColor()` библиотеки GLUT принимает четыре аргумента. Она ассоциирует индекс, задаваемый первым аргументом, с RGB-тройкой, задаваемой вторым, третьим и четвертым аргументами. При $i = 0$ цветовой индекс 16 связывается с черным RGB-цветом (0, 0, 0). Цвет в карте монотонно изменяется до рассеянного цвета материала с индексом 47 (при $i = 31$), которому соответствует фиолетовый RGB-цвет (1.0, 0.0, 1.0). Вторая команда в теле цикла строит карту между фиолетовым рассеянным цветом и белым (1.0, 1.0, 1.0) отраженным цветом (индекс 79).

Математика освещения в режиме индексации цвета

Как и следовало ожидать, поскольку в режиме индексации цвета доступно другое подмножество параметров освещения, нежели в RGBA-режиме, расчеты также производятся иначе. Так как в этом режиме исходящий от материала свет, а также фоновое освещение не задаются, здесь представляют интерес только диффузная и отраженная составляющие источников света. Но даже эти элементы рассчитываются иначе: так, как показано ниже.

Начнем с диффузной и отраженной составляющих из RGBA-уравнений. В рассеянной составляющей заменим $\text{diffuse}_{\text{light}} \times \text{diffuse}_{\text{material}}$ на d_{cl}, вычисленное в предыдущем разделе для режима индексирования цветов. Аналогично, для отраженной составляющей вместо $\text{specular}_{\text{light}} \times \text{specular}_{\text{material}}$ подставим s_{cl} из предыдущего раздела. (Ослабление, эффект прожектора и другие компоненты вычисляются

так же, как в RGBA-режиме.) Назовем эти модифицированные элементы d и s соответственно. Теперь пусть $s' = \min\{s, 1\}$. Вычислим

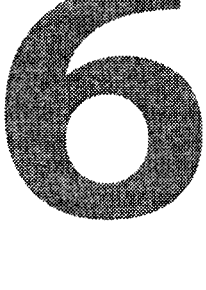
$$c = a_m + d(1 - s')(d_m - a_m) + s'(s_m - a_m),$$

где a_m, d_m и s_m представляют собой индексы фонового, рассеянного и отраженного цветов материала, установленные командой `glMaterial*()` со значением аргумента `GL_COLOR_INDEXES`. Индекс результирующего цвета

$$c' = \min\{c, s_m\}.$$

После того как расчет освещенности произведен, значения индексов преобразуются в формат с фиксированной запятой (с произвольным количеством битов справа от двоичной запятой). Далее целая часть индекса маскируется (с помощью побитового И) числом $2^n - 1$, где n — количество битов на цвет в индексном буфере.

Смешивание, сглаживание, туман и отклонение



После прочтения этой главы вы сможете:

- Смешивать цвета для получения эффекта полупрозрачности объекта
- Устранять ступенчатость линий и многоугольников
- Создавать сцены с реальными атмосферными эффектами
- Рисовать скругленные (сглаженные) точки с размером, зависящим от расстояния до точки наблюдения
- Рисовать объекты с заданной глубиной без дефектов изображения, возникающих по причине наложения объектов друг на друга

В предыдущих главах мы учили вас создавать сцены компьютерной графики. Значит, на этот момент вы умеете:

- рисовать геометрические объекты;
- преобразовывать их так, чтобы они выглядели правильно с любой точки обзора в нужной перспективе;
- задавать их цвет и вид раскраски;
- добавлять источники света и определять их воздействие на объекты в сцене.

Теперь вы подготовлены для того, чтобы идти дальше. Эта глава предлагает к рассмотрению пять технологий, позволяющие ввести дополнительные детали в сцену и придать ей большую реалистичность. Все эти технологии довольно просты – фактически труднее их объяснить, чем программировать. Каждая из них описывается в своем разделе:

- **Смешивание.** Рассказывает, как определить функцию смешивания, которая комбинирует цвета приемника и получателя. В результате некоторые части сцены оказываются полупрозрачными.
- **Сглаживание.** Описывает технологию незначительного изменения цвета (antialiasing), за счет которой границы точек, линий и многоугольников кажутся более ровными, а не угловатыми и ступенчатыми. Более мощная технология мультивыборки (multisampling) предназначена для сглаживания любых примитивов в сцене.
- **Туман.** Объясняет, как создать иллюзию глубины, вычисляя значения цветов объекта на основе расстояния до точки обзора. Цель — добиться того, чтобы объекты, находящиеся далеко, растворялись в фоне, как это происходит в реальной жизни.
- **Параметры точки.** Рассматривает эффективную технологию рисования примитива точки с размером и цветом, изменяющимися в зависимости от расстояния до наблюдателя. Параметры точки используются в системах частиц — при моделировании дождя, снега и прочего в графических программах.
- При попытке рисования каркаса поверх закрашенного объекта по тем же вершинам могут возникнуть ненужные артефакты. Раздел **«Отклонение многоугольников по глубине»** показывает, как подогнать (отклонить) значения глубины так, чтобы векторный, тонированный объект был неотличим от прототипа.

Поддержка девиации многоугольников появилась в OpenGL версии 1.1.

В версию 1.2 была введена возможность выбора выражения смешивания (функция `glBlendEquation()`), константные цвета (`glBlendColor()`) и типы смешивания: `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA` и `GL_ONE_MINUS_CONSTANT_ALPHA`. В версиях 1.2 и 1.3 эти возможности поддерживаются, только если реализация включает в себя расширение Imaging Subset. Начиная с версии 1.4, наличие выражений смешивания, константных цветов и факторов влияния стало обязательным.

В версии 1.3 поддержка мультивыборки стала частью ядра OpenGL.

В версии 1.4 также были добавлены следующие возможности, затрагиваемые в данной главе:

- использование GL_SRC_COLOR и GL_ONE_MINUS_SRC_COLOR в качестве факторов смешивания для источника (до версии 1.4 GL_SRC_COLOR и GL_ONE_MINUS_SRC_COLOR были факторами смешивания для получателя);
- использование GL_DST_COLOR и GL_ONE_MINUS_DST_COLOR как типов смешивания для получателя (до версии 1.4 GL_DST_COLOR и GL_ONE_MINUS_DST_COLOR были факторами смешивания для источника);
- явное определение координат тумана;
- параметры точки для управления характеристиками точечных примитивов;
- возможность смешивания RGB-компонентов и альфа-канала цвета раздельными функциями смешивания.

Смешивание

Что же такое «альфа»? Мы неоднократно употребляли этот термин (или как букву A в аббревиатуре RGBA), но до сих пор не конкретизировали. Альфа-значения указывались для функции glColor*(), задавались в функции glClearColor() для определения цвета очистки и использовались при задании некоторых параметров освещения, таких как свойства материала и интенсивность света источника. В главе 4 говорилось, что пиксел на экране монитора излучает красный, зеленый и синий света, и интенсивность света складывается из значений красной, зеленой и синей составляющих — RGB. Так каким же образом альфа-значение (A) влияет на рисование в окне на экране?

Когда включено смешивание, значение альфа-компонента часто используется для комбинирования значения цвета обрабатываемого фрагмента со значением цвета пиксела, которое уже хранится в буфере кадров. Смешивание происходит после растеризации сцены и разбивки ее на фрагменты, но до того, как конечные пикселы сохраняются в видеобуфере. Значения альфа-компонента также играют свою роль при выполнении альфа-теста, принимающего или отбраковывающего фрагмент на основе его альфа-значений. (См. главу 10 для получения дополнительной информации.)

В отсутствие смешивания информация о цвете в буфере цвета перезаписывается, и ни о какой прозрачности говорить не приходится. Смешивание позволяет управлять тем, как (и в каком соотношении) сохраненный цвет должен быть объединен со значением цвета нового фрагмента. И, соответственно, с помощью альфа-компонента можно получить светопроницаемые фрагменты, через которые будет «проступать» ранее сохраненный цвет. Смешивание цвета — основа технологии полупрозрачности, цифровой композиции и закрашивания.

ПРИМЕЧАНИЕ

Альфа-значения не определяются в режиме индексации цвета, то есть операции смешивания в этом режиме не выполняются.

Наиболее естественный способ понять смешивание — думать об RGB-компонентах фрагмента как о цвете, а альфа-составляющую считать показателем непрозрачности (opacity). Прозрачные или светопроницаемые поверхности имеют меньшую непрозрачность, чем непрозрачные, и, соответственно, меньшие значения альфа-составляющей. Например, если вы смотрите на объект через зеленое стекло, цвет, который вы увидите, будет частично зеленым от стекла, а частично — цветом объекта. Процентное соотношение зависит от пропускной способности стекла: если стекло пропускает 80% света, который на него падает (то есть непрозрачно на 20%), этот цвет будет содержать 20% цвета стекла и 80% цвета объекта за ним. Легко представить ситуацию с несколькими прозрачными поверхностями. Так, линия взгляда до некоторой точки в салоне автомобиля проходит через одно стекло, а при взгляде сквозь автомобиль объекты видны через два стекла.

Факторы влияния источника и получателя

Во время смешивания цвета входящего фрагмента (*источника*) комбинируются с соответствующими текущими сохраненными значениями пикселов (*получателя*) в два этапа. Сначала вы указываете, как вычисляются факторы влияния источника и получателя. Эти коэффициенты представляют собой четверки значений, которые умножаются на каждую составляющую R, G, B и A источника и получателя соответственно. Затем компоненты двух наборов RGBA попарно объединяются. Для того, чтобы показать это математически, представим коэффициенты смешивания в виде (S_r, S_g, S_b, S_a) и (D_r, D_g, D_b, D_a) соответственно, а значения RGBA для источника и получателя пусть обозначаются подстрочными символами s и d. Наконец, вычисляются смешанные значения RGBA:

$$(R_sS_r + R_dD_r,\ G_sS_g + G_dD_g,\ B_sS_b + B_dD_b,\ A_sS_a + A_dD_a).$$

И последнее, что делается, — это каждая составляющая четверки ограничивается диапазоном [0, 1].

По умолчанию объединение фрагментов источника и получателя происходит путем сложения их значений ($C_sS + C_dD$). См. раздел «Объединение пикселов на основе выражений», где описан выбор нужной математической операции.

Есть два способа задания коэффициентов смешивания источника и получателя. Первый: функция `glBlendFunc()`, которая определяет два коэффициента смешивания — для источника и для получателя. Второй: функция `glBlebdFuncSeparate()`, в расчете на четыре коэффициента, чтобы использовать для альфа-значений функцию смешивания, отличную от функции для RGB-компонентов.

void `glBlendFunc`(GLenum *srcfactor*, GLenum *destfactor*);

Управляет смешиванием значений цветов обрабатываемого фрагмента (источник) и фрагмента, уже сохраненного в видеобуфере (получатель). Значения аргументов приведены в табл. 6.1. Аргумент *srcfactor* определяет способ вычисления фактора смешивания источника, а параметр *destfactor* — получателя.

Коэффициенты смешивания лежат в пределах [0, 1]. Значения, полученные в результате объединения цветов источника и получателя, также попадают в пределы [0, 1].

void `glBlendFuncSeparate`(GLenum *srcRGB*, GLenum *destRGB*, GLenum *srcalpha*, GLenum *destalpha*);

Так же, как и функция `glBlendFunc()`, функция `glBlendFuncSeparate()` управляет смешиванием цветов источника (фрагмента) и получателя (буфера кадров); и аргументы принимают те же значения (табл. 6.1). Аргумент *srcRGB* задает коэффициент смешивания источника для значений цвета, *destRGB* – коэффициент смешивания получателя, тоже для цвета. Аргументы *srcalpha* и *detsalpha* определяют коэффициенты смешивания для значений альфа-компонента источника и получателя соответственно.

Коэффициенты смешивания не выходят за диапазон [0, 1]. Значения, полученные в результате объединения значений цветов источника и получателя, также лежат в пределах [0, 1].

ПРИМЕЧАНИЕ

В табл. 6.1 значения RGBA источника, получателя и цветовых констант обозначены индексами s, d и c соответственно. Разность четверок означает разность их составляющих. Значение GL_SRC_ALPHA_SATURATE может использоваться только как коэффициент смешивания для источника.

Таблица 6.1. Коэффициенты смешивания источника и получателя

Константа	Коэффициент смешивания RGB	Коэффициент смешивания альфа-компонентов
GL_ZERO	(0, 0, 0)	0
GL_ONE	(1, 1, 1)	1
GL_SRC_COLOR	(R_s, G_s, B_s)	A_s
GL_ONE_MINUS_SRC_COLOR	(1, 1, 1) – (R_s, G_s, B_s)	$1 - A_S$
GL_DST_COLOR	(Rd, G_d, B_d)	A_d
GL_ONE_MINUS_DST_COLOR	(1, 1, 1) – (R_d, G_d, B_d)	$1 - A_d$
GL_SRC_ALPHA	(A_s, A_s, A_s)	A_s
GL_ONE_MINUS_SRC_ALPHA	(1, 1, 1) – (A_s, A_s, A_s)	$1 - A_S$
GL_DST_ALPHA	(A_d, A_d, A_d)	A_d
GL_ONE_MINUS_DST_ALPHA	(1, 1, 1) – (A_d, A_d, A_d)	$1 - A_d$
GL_CONSTANT_COLOR	(Rc, G_c, B_c)	A_c
GL_ONE_MINUS_CONSTANT_COLOR	(1, 1, 1) – (R_c, G_c, B_c)	$1 - A_C$
GL_CONSTANT_ALPHA	(A_c, A_c, A_c)	A_c
GL_ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1) – (A_c, A_c, A_c)	$1 - A_c$
GL_SRC_ALPHA_SATURATE	(f, f, f); f = min(A_s, 1 – A_d)	1

Использование одной из функций смешивания `GL*CONSTANT*` подразумевает предшествующий вызов функции `glBlendColor()`.

void `glBlendColor`(GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);

Устанавливает значения аргументов *red*, *green*, *blue* и *alpha* в качестве составляющих цветовой константы (R_c, G_c, B_c, A_c), используемой в операциях смешивания.

Включение смешивания

Вне зависимости от того, как определяется функция смешивания, чтобы она работала, следует включить режим смешивания:

```
glEnable(GL_BLEND);
```

Отключается данный режим вызовом `glDisable` со значением аргумента `GL_BLEND`. Заметьте, что использование констант `GL_ONE` (для источника) и `GL_ZERO` (для получателя) дает тот же результат, что и отключение смешивания; эти значения приняты по умолчанию.

Объединение пикселов на основе выражений

Обычно при смешивании цвета в видеобуфере складываются с цветами обрабатываемого фрагмента и так получается новый цвет. Но вы можете использовать функцию `glBlendEquation()` для активации иной математической операции — вычитания, подмены на минимальное или максимальное значение и др.

> void `glBlendEquation`(GLenum *mode*);
>
> Задает, как смешиваются цвета буфера кадров и источника. Аргумент *mode* выбирается из списка: `GL_FUNC_ADD` (по умолчанию), `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN` и `GL_MAX`. Доступные режимы описаны в табл. 6.2.

В табл. 6.2 буквы C_s и C_d обозначают цвета источника и получателя; индексы S и D – это факторы влияния источника и получателя, определенные с помощью функций `glBlendFunc()` и `glBlendFuncSeparate()`.

Таблица 6.2. Математические выражения для смешивания

Параметр	Операция
GL_FUNC_ADD	$C_sS + C_dD$
GL_FUNC_SUBTRACT	$C_sS - C_dD$
GL_FUNC_REVERSE_SUBTRACT	$C_dD - C_sS$
GL_MIN	$\min(C_sS, C_dD)$
GL_MAX	$\max(C_sS, C_dD)$

Листинг 6.1 демонстрирует практическое применение различных выражений смешивания. Тип смешивания выбирается клавишами `a`, `s`, `r`, `m` и `x`. В качестве источника используется синий квадрат, а получателем является желтый фон. Коэффициенты смешивания для каждого цвета устанавливаются в `GL_ONE` посредством `glBlendFunc()`.

Листинг 6.1. Использование выражений смешивания: blendeqn.c

```
/*  Клавиши, используемые для замены выражения смешивания
*
*   'a'     ->     GL_FUNC_ADD
*   's'     ->     GL_FUNC_SUBTRACT
*   'r'     ->     GL_FUNC_REVERSE_SUBTRACT
```

продолжение

Листинг 6.1 (*продолжение*)

```
*     'm'     ->     GL_MIN
*     'x'     ->     GL_MAX
*/

void init(void)
{
    glClearColor(1.0, 1.0, 0.0, 0.0);

    glBlendFunc(GL_ONE, GL_ONE);
    glEnable(GL_BLEND);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 0.0, 1.0);
    glRectf(-0.5, -0.5, 0.5, 0.5);
    glFlush();
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'a': case 'A':
            /* Цвета складываются: (1,1,0) + (0,0,1) = (1,1,1).
             * В результате получается белый квадрат на желтом фоне */
              glBlendEquation(GL_FUNC_ADD);
              break;

        case 's': case 'S':
            /* Цвета вычитаются: (0,0,1) – (1,1,0) = (-1,-1,1)
             * После ограничения к (0, 0, 1) получается синий квадрат
             * на желтом фоне */
              glBlendEquation(GL_FUNC_SUBTRACT);
              break;
        case 'r': case 'R':
            /* Цвета вычитаются: (1,1,0) - (0,0,1) = (1,1,-1).
             * После ограничения к (1, 1, 0) получаем желтый цвет
             * и для квадрата и для фона */
              glBlendEquation(GL_FUNC_REVERSE_SUBTRACT);
              break;

        case 'm': case 'M':
            /* Находится минимум по всем компонентам
             * как [min(1,0),min(1,0),min(0,1)], что дает (0,0,0).
             * Поэтому имеем черный квадрат на желтом фоне */
              glBlendEquation(GL_MIN);
              break;

        case 'x': case 'X':
            /* Находится максимум по всем компонентам
             * как [max(1, 0), max(1, 0), max(0, 1)], что дает (1, 1, 1).
             * Результат - белый квадрат на желтом фоне */
```

```
            glBlendEquation(GL_MAX);
            break;

        case 27:
            exit(0);
    }    break;

    glutPostRedisplay();
}
```

Область применения смешивания

Не все комбинации коэффициентов источника и получателя имеют смысл. Большинство приложений нуждается в ограниченном наборе сочетаний. Ниже описано типичное использование некоторых комбинаций факторов влияния источника и получателя. Некоторые из этих примеров требуют только входное значение альфа-компонента, поэтому их работа не зависит от того, хранятся ли альфа-величины в буфере кадров. Как и везде, здесь обычно один и тот же результат достижим различными способами.

- Один из способов получения картинки, состоящей наполовину из одного изображения и наполовину из другого, — это установка коэффициента источника в GL_ONE, а значения коэффициента получателя — в GL_ZERO, после чего рисуется первое изображение. Затем фактор влияния источника устанавливается в GL_SRC_ALPHA, а коэффициент получателя — в GL_ONE_MINUS_SRC_ALPHA, и рисуется второе изображение с альфа-компонентом, равным 0.5. Эта пара значений обычно используется для большинства операций смешивания. Если изображение должно состоять на три четверти из одного рисунка и на четверть из другого рисунка, сначала выводится первое изображение, а затем — второе с альфа-значением 0.25.
- Чтобы смешать три рисунка в равной пропорции, нужно установить коэффициент получателя в GL_ONE, а коэффициент источника — в GL_SRC_ALPHA. Затем рисуется каждый рисунок с A = 0.3333333. Таким образом, яркость каждого рисунка равна одной трети оригинальной, что заметно в местах, где рисунки не перекрываются.
- Допустим, необходимо написать графический редактор и в нем имитировать кисть для сгущения красок, когда каждый мазок добавляет немного больше цвета к текущему цвету изображения (скажем, 10 % цвета на 90 % изображения при каждом проходе). Для этого изображение рисуется кистью с A = 10 %, а факторы влияния устанавливаются в GL_SRC_ALPHA (для источника) и GL_ONE_MINUS_SRC_ALPHA (для получателя). Кроме того, можно изменять альфа-значение внутри следа для того, чтобы придать ему свойство сглаживания, когда кисть добавляет больше цвета в центре и меньше по краям (см. раздел «Сглаживание»). Точно так же создается ластик путем установки цвета стирания в цвет фона.
- Функция смешивания, которая использует цвета источника и получателя — значения GL_DST_COLOR или GL_ONE_MINUS_DST_COLOR для коэффициентов

источника и GL_SRC_COLOR или GL_ONE_MINUS_SRC_COLOR для коэффициентов получателя, — позволяет корректировать каждый компонент цвета в отдельности. Эта операция эквивалентна применению простого фильтра; так, умножение красной составляющей на 80 %, зеленой составляющей на 40 %, а синей составляющей на 72 % имитирует просмотр сцены через светофильтр, блокирующий 20 % красного света, 60 % зеленого и 28 % синего.

- Предположим, что нужно нарисовать рисунок, составленный из трех полупрозрачных поверхностей на залитом фоне, частично перекрывающихся. Пусть самая дальняя поверхность пропускает 80 % цвета за ней, следующая пропускает 40 %, а ближняя — 90 %. Для получения этого изображения сначала рисуется фон с факторами влияния источника и получателя по умолчанию, затем меняем коэффициенты смешивания на GL_SRC_ALPHA (для источника) GL_ONE_MINUS_SRC_ALPHA (для получателя). Теперь рисуем дальнюю поверхность с A = 0.2, затем промежуточную поверхность с A = 0.6 и наконец ближнюю поверхность с A = 0.1.

- Если ваше оборудование имеет альфа-плоскости, можно построить объекты один раз (включая их альфа-значения), вернуться назад, а затем выполнить требуемые операции матирования и компоновки на уже полностью построенных объектах. (Пример использования этой техники см. в работе Тома Даффа *Compositing 3D Rendered Images* (труды конференции SIGGRAPH, 1985, с. 41–44.) Объекты, составляющие изображение, могут быть получены из различных источников: они могут быть построены командами OpenGL, при помощи технологии трассировки лучей или метода излучательности, реализованных в других графических библиотеках, или могут просто являться сканированными копиями.

- Задавая различные значения альфа-компонента для отдельных фрагментов изображения, можно создавать растровые изображения непрямоугольной формы. В большинстве случаев для этого нужно обнулить альфа-значение каждой «невидимой» части и присвоить максимальное (1.0) каждому непрозрачному участку. Например, так можно нарисовать многоугольник в форме дерева и применить к нему лиственный орнамент. Наблюдатель сможет смотреть сквозь фрагменты прямоугольной текстуры, которые будут прозрачными, если их значения альфа-компонента равны 0. Этот метод, иногда называемый *«обшивкой»* (billboarding), работает быстрее, чем конструирование дерева из «трехмерных» многоугольников. Пример приведен на рис. 6.1, где дерево является одиночным прямоугольником, способным вращаться вокруг оси ствола, как показано с помощью контуров, причем оставаясь всегда обращенным «лицом» к наблюдателю. (См. раздел «Функции текстур» главы 9 для получения более подробной информации о смешивании текстур.)

- Смешивание также применяется для *сглаживания* (antialiasing), под которым понимается технология, «разглаживающая» шероховатости во внешнем виде примитивов, рисуемых на растровом экране. (См. раздел «Сглаживание» для получения дополнительной информации.)

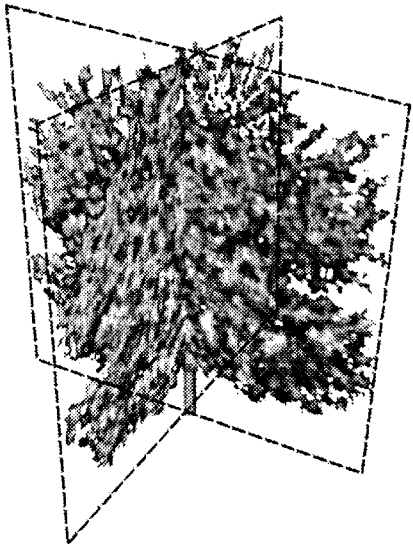


Рис. 6.1. Создание непрямоугольного растрового изображения

Пример смешивания

Программа, представленная в листинге 6.2, строит два перекрывающихся треугольника, со значениями альфа-компонента 0.75. Смешивание включено, а факторы влияния источника и получателя установлены в GL_SRC_ALPHA и GL_ONE_MINUS_SRC_ALPHA соответственно.

При запуске программы слева появляется желтый треугольник, а затем справа рисуется цианowый треугольник таким образом, чтобы они перекрывались в центре окна и циан смешивался с изначальным желтым цветом. Смена порядка рисования треугольников производится с помощью клавиши t.

Листинг 6.2. Пример смешивания: alpha.c

```
static int leftFirst = GL_TRUE;
/* Инициализируем функцию смешивания */
static void init(void)
{
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glShadeModel(GL_FLAT);
   glClearColor(0.0, 0.0, 0.0, 0.0);
}

static void drawLeftTriangle(void) {
/* Рисуем желтый треугольник в левой части экрана */
   glBegin(GL_TRIANGLES);
       glColor4f(1.0, 1.0, 0.0, 0.75);
       glVertex3f(0.1, 0.9, 0.0);
       glVertex3f(0.1, 0.1, 0.0);
       glVertex3f(0.7, 0.5, 0.0);
```

продолжение ⇩

Листинг 6.2 (*продолжение*)

```
    glEnd();
}

static void drawRightTriangle(void)
{
/* Рисуем цианový треугольник в правой части экрана */
     glBegin(GL_TRIANGLES);
        glColor4f(0.0, 1.0, 1.0, 0.75);
        glVertex3f(0.9, 0.9, 0.0);
        glVertex3f(0.3, 0.5, 0.0);
        glVertex3f(0.9, 0.1, 0.0);
    glEnd();
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    if (leftFirst) {
        drawLeftTriangle();
        drawRightTriangle();
    }
    else {
        drawRightTriangle();
        drawLeftTriangle();
    }
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D(0.0, 1.0, 0.0, 1.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D(0.0, 1.0*(GLfloat)w/(GLfloat)h, 0.0, 1.0);
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 't':
        case 'T' :
            leftFirst = !leftFirst;
            glutPostRedisplay();
            break;
        case 27:  /* Клавиша Esc */
            exit(0);
            break;
        default:
            break;
    }
}
```

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(200, 200);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Последовательность рисования треугольников влияет на цвет их пересечения. В случае, когда сначала рисуется левый треугольник, циановый фрагмент (источник) смешивается с желтым фрагментом, который уже хранится в видеобуфере (получатель). Когда же первым рисуется правый треугольник, желтый накладывается на циан. Так как значения альфа-компонента равны 0.75, реальный коэффициент смешивания равен 0.75 для источника и 1.0 – 0.75 = 0.25 для получателя. Другими словами, фрагменты-источники слегка прозрачны, и они оказывают большее влияние на конечный цвет, чем фрагменты-получатели.

Трехмерное смешивание с использованием буфера глубины

Как было показано в предыдущем примере, порядок отрисовки многоугольников оказывает существенное влияние на результат смешивания. При рисовании трехмерных полупрозрачных объектов результат зависит от способа рисования многоугольников — от задней грани к передней или от передней к задней. При определении правильной последовательности необходимо также учесть эффект буфера глубины. (См. раздел «Удаление невидимых поверхностей» главы 5, а также см. раздел «Тест глубины» главы 10.) Буфер глубины хранит расстояние от точки обзора до части объекта, занимающей заданный пиксел окна на экране. Если данный пиксел принадлежит еще и другой части этого или другого объекта, он будет сохранен только в том случае, если текущий объект находится ближе к наблюдателю; тогда значение в буфере глубины изменится. Таким способом выявляются невидимые (или скрытые) части поверхностей, не подлежащие рисованию и которые, соответственно, не используются при смешивании.

Если необходимо совместить непрозрачные и прозрачные объекты в одной сцене, буфер глубины поможет удалить невидимые поверхности любых объектов, заслоненные непрозрачными объектами. Если непрозрачный объект скрывает прозрачный или другой непрозрачный объект, буфер глубины должен отсеять более удаленный объект. Однако если прозрачный объект находится ближе, его нужно смешать с непрозрачным объектом. Конечно, можно определить должный порядок рисования многоугольников, когда сцена неподвижна, но при перемещении наблюдателя или объекта эта проблема вырастает в практически неразрешимую.

Решение кроется в использовании буфера глубины в режиме «только для чтения» при рисовании прозрачных объектов. Сначала рисуются все непрозрачные объекты в обычном режиме доступа к буферу глубины. Затем устанавливается запрет на изменение значений в буфере глубины. Когда рисуются прозрачные объекты, их глубина также сравнивается с сохраненными в буфере значениями для непрозрачных объектов, то есть прозрачные объекты не рисуются, если расположены за непрозрачными. Если же они ближе к точке обзора, это не отбрасывает непрозрачные объекты, так как буфер глубины не может быть изменен. Для управления режимом доступа к буферу глубины предназначена функция `glDepthMask()`. Если в качестве аргумента ей передать значение `GL_FALSE`, буфер переходит в режим «только для чтения», а значение `GL_TRUE` восстанавливает нормальный режим, разрешая запись.

Пример использования этого метода для рисования непрозрачных и прозрачных трехмерных объектов приведен в листинге 6.3. В данной программе клавиша `a` переключает последовательность анимации на состояние, где прозрачный куб перемещается через непрозрачную сферу. Нажатие клавиши `r` переводит объекты в сцене в первоначальное состояние. Для получения хорошего результата при пересечении прозрачных объектов их нужно рисовать от задней грани к передней.

Листинг 6.3. Трехмерное смешивание: alpha3D.c

```
#define MAXZ 8.0
#define MINZ -8.0
#define ZINC 0.4
static float solidZ = MAXZ;
static float transparentZ = MINZ;
static GLuint sphereList, cubeList;

static void init(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 0.15 };
    GLfloat mat_shininess[] = { 100.0 };
    GLfloat position[]  = { 0.5, 0.5, 1.0, 0.0 };

    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);

    sphereList = glGenLists(1);
    glNewList(sphereList, GL_COMPILE);
        glutSolidSphere(0.4, 16, 16);
    glEndList();
    cubeList = glGenLists(1);
    glNewList(cubeList, GL_COMPILE);
        glutSolidCube(0.6);
    glEndList();
}
```

```
void display(void)
{
   GLfloat mat_solid[] = { 0.75, 0.75, 0.0, 1.0 };
   GLfloat mat_zero[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat mat_transparent[] = { 0.0, 0.8, 0.8, 0.6 };
   GLfloat mat_emission[] = { 0.0, 0.3, 0.3, 0.6 };

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix();
      glTranslatef(-0.15, -0.15, solidZ);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_zero);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_solid);
      glCallList(sphereList);
   glPopMatrix();

   glPushMatrix();
      glTranslatef(0.15, 0.15, transparentZ);
      glRotatef(15.0, 1.0, 1.0, 0.0);
      glRotatef(30.0, 0.0, 1.0, 0.0);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_transparent);
      glEnable(GL_BLEND);
      glDepthMask(GL_FALSE);
      glBlendFunc(GL_SRC_ALPHA, GL_ONE);
      glCallList(cubeList);
      glDepthMask(GL_TRUE);
      glDisable(GL_BLEND);
   glPopMatrix();

   glutSwapBuffers();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLint) w, (GLint) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho(-1.5*(GLfloat)w/(GLfloat)h, 1.5*(GLfloat)w/(GLfloat)h, -1.5,
1.5, -10.0, 10.0);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity();
}

void animate(void)
{
   if (solidZ <= MINZ || transparentZ >= MAXZ)
      glutIdleFunc(NULL);
   else {
```

продолжение ⇩

Листинг 6.3 (*продолжение*)

```
        solidZ -= ZINC; transparentZ += ZINC; glutPostRedisplay();
    }
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'a':
        case 'A':
            solidZ = MAXZ;
            transparentZ = MINZ;
            glutIdleFunc(animate);
            break;
        case 'r':
        case 'R':
            solidZ = MAXZ;
            transparentZ = MINZ;
            glutPostRedisplay();
            break;
        case 27:
            exit(0);
    }    break;
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow(argv[0]);
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Сглаживание

Как вы уже могли заметить, в некоторых написанных вами программах линии на рисунках, особенно почти горизонтальные или почти вертикальные, кажутся зубчатыми. Эта «зубчатость» возникает из-за того, что идеальная в теории линия на практике составлена из групп пикселов, которые выстраиваются на пиксельной сетке. В данном разделе описаны технологии для устранения этой ступенчатости (ее уменьшения). На рис. 6.2 изображены две пересекающиеся линии, как в исправленном, так и в исходном виде. Изображения масштабированы для большей наглядности.

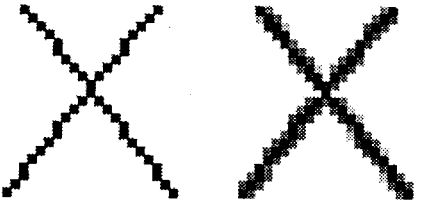


Рис. 6.2. Ступенчатые и сглаженные линии

На рис. 6.3 показано, что диагональная линия толщиной в 1 пиксел некоторые пикселы перекрывает больше, чем остальные. Фактически при выполнении сглаживания OpenGL вычисляет значения *покрытия* для каждого фрагмента на основании площади пикселов, охваченной линией. На рис. 6.3 также приведены эти значения покрытия для рисуемой линии. В режиме RGBA OpenGL умножает значение альфа-компонента для фрагмента на значение покрытия. Затем получившееся значение может быть применено для смешивания фрагмента с соответствующим пикселом, хранимым в буфере кадров. В режиме индексации цвета OpenGL устанавливает младшие четыре бита в индексе цвета на основе степени покрытия фрагмента (0000, если покрытие отсутствует, и 1111 — при полном охвате). Задачей программиста является заполнение карты цветов и использование этого свойства для получения нужного эффекта.

Подробности вычисления значений покрытия достаточно сложны, их трудно представить в общем виде, так как они существенно зависимы от реализации OpenGL. Выработать компромисс между качеством изображения и скоростью выполнения помогает функция `glHint()`; с ней полезно поупражняться, но не все реализации ее поддерживают.

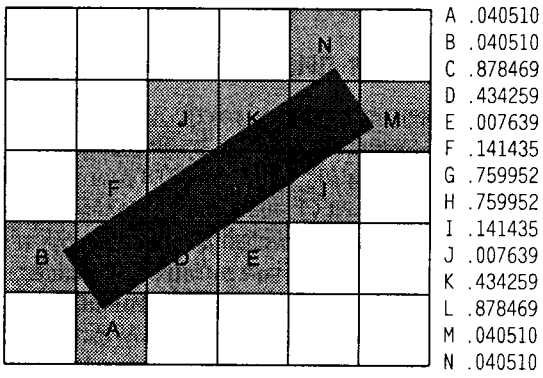


Рис. 6.3. Определение степени покрытия

void `glHint`(GLenum *target*, GLenum hint);

Управляет рядом режимов работы OpenGL. Аргумент *target* идентифицирует подконтрольный режим, его возможные значения перечислены в табл. 6.3. Аргумент *hint* может принимать следующие значения: `GL_FASTER` для отдания предпочтения скорости обработки, `GL_NICEST` с целью достижения наивысшего качества или `GL_DONT_CARE` для снятия всех требований. Интерпретация

подсказок зависит от реализации. Некоторые реализации OpenGL могут на них не реагировать.

Значение аргумента *target* GL_PERSPECTIVE_CORRECTION_HINT связано с интерполяцией значений цвета и координат текстур вдоль примитива: или линейно в пространстве экрана (относительно простые вычисления), или с нахождением правильной перспективы (что требует больших усилий). Часто системы выполняют линейную интерполяцию цвета, поскольку результаты, являющиеся технически не совсем корректными, визуально приемлемы. Однако в большинстве случаев текстуры требуют максимально пригодную интерполяцию по перспективе. Таким образом, реализация OpenGL может задействовать этот параметр для управления методом интерполяции. (См. главу 3 для справки о перспективной проекции, главу 4 для получения информации о цвете и главу 9 по вопросам текстурирования.)

Таблица 6.3. Значения аргумента target функции glHint()

Значение	Применение
GL_POINT_SMOOTH_HINT, GL_LINE_SMOOTH_HINT, GL_POLYGON_SMOOOTH_HINT	Управление качеством сглаживания точек, линий или многоугольников
GL_FOG_HINT	Указывает, выполнять ли вычисления при наложении тумана по пикселам (GL_NICEST) или по вершинам (GL_FASTEST)
GL_PERSPECTIVE_CORRECTION_HINT	Качество интерполяции координат при вычислении цветов и наложении текстур
GL_GENERATE_MIPMAP_HINT	Качество и производительность автоматической генерации MIPMAP-текстур
GL_TEXTURE_COMPRESSION_HINT	Качество и производительность сжатых текстурных изображений

Сглаживание точек или линий

Одним из способов устранения ступенчатости для точек или линий является включение сглаживания с помощью вызова функции `glEnable()` со значением аргумента GL_POINT_SMOOOTH или GL_LINE_SMOOTH для точек и линий соответственно. Кроме того, функция `glHint()` позволяет управлять качеством сглаживания. (Напомним, что размер точки и толщина линии являются настраиваемыми. Кроме того, можно создавать прерывистые линии. См. раздел «Подробно о линиях» главы 2.) Затем выполняется алгоритм, описанный в одном из следующих разделов, в зависимости от режима работы (RGBA или индексированных цветов).

Другой техникой сглаживания точек или линий является технология мультивыборки, описанная в разделе «Сглаживание геометрических примитивов с помощью мультивыборки» этой главы.

Сглаживание в режиме RGBA

В режиме RGBA необходимо включить смешивание. Обычно используются следующие факторы влияния: GL_SRC_ALPHA для источника и GL_ONE_MINUS_SRC_

ALPHA для получателя. Альтернативный способ: если значения альфа-компонента для получателя установить в GL_ONE, линии в местах пересечения будут более яркими. Сглаживание тем более заметно, если используются большие альфа-значения. Помните, что поскольку вы выполняете смешивание, вероятно, нужно учесть порядок рисования, как было показано в разделе «Трехмерное смешивание с использованием буфера глубины».

Однако в большинстве случаев порядок рисования можно не учитывать, и это не окажет каких-то нежелательных эффектов. Программа в листинге 6.4 инициализирует нужные режимы для сглаживания, а затем рисует две пересекающиеся диагональные линии. Когда программа запущена, с помощью клавиши r можно вращать линии для того, чтобы оценивать их выравнивание при различных углах наклона. Обратите внимание, что в этом примере мы обходимся без буфера глубины и отключили его.

Листинг 6.4. Сглаженные линии: aargb.c

```
static float rotAngle = 0.;

/*    Инициализируем сглаживание для режима RGBA, включая альфа-смешивание,
 *    указания режима работы и толщины линии. Выведем значения детализации
 *    и толщины линии, зависимые от реализации
 */

void init(void)
{
   GLfloat values[2];
   glGetFloatv(GL_LINE_WIDTH_GRANULARITY, values);
   printf("GL_LINE_WIDTH_GRANULARITY равно %3.1f\n", values[ 0 ]);
   glGetFloatv(GL_LINE_WIDTH_RANGE, values);
   printf("GL_LINE_WIDTH_RANGE равно %3.1f %3.1f\n", values[0], values[1]);

   glEnable(GL_LINE_SMOOTH);
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
   glLineWidth(1.5);

   glClearColor(0.0, 0.0, 0.0, 0.0);
}

/* Рисуем две диагональные линии в виде буквы X */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 1.0, 0.0);
   glPushMatrix();
   glRotatef(-rotAngle, 0.0, 0.0, 0.1);
   glBegin(GL_LINES);
      glVertex2f(-0.5, 0.5);
      glVertex2f(0.5, -0.5);
   glEnd();
   glPopMatrix();
```

продолжение ⇩

Листинг 6.4 (*продолжение*)

```
    glColor3f(0.0, 0.0, 1.0);
    glPushMatrix();
    glRotatef(rotAngle, 0.0, 0.0, 0.1);
    glBegin(GL_LINES);
        glVertex2f(0.5, 0.5);
        glVertex2f(-0.5, -0.5);
    glEnd();
    glPopMatrix();

    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLint) w, (GLint) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D(-1.0, 1.0, -1.0*(GLfloat)h/(GLfloat)w,
1.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D(-1.0*(GLfloat)w/(GLfloat)h, 1.0*(GLfloat)w/(GLfloat)h,
-1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'r':
        case 'R':
            rotAngle += 20.;
            if (rotAngle >= 360.) rotAngle = 0.;
            glutPostRedisplay();
            break;
        case 27:    /* Клавиша Esc */
            exit(0);
            break;
        default:
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSi ze(200, 200);
    glutCreateWindow(argv[ 0 ] );
    init();
    glutReshapeFunc(reshape);
```

```
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Сглаживание в режиме индексации цвета

Для устранения ступенчатости в режиме индексации цвета необходимо загрузить и использовать карту цветов. Так как последние 4 бита в индексе цвета определяют степень покрытия, необходимо инициировать 16 соседних индексов цветами, изменяющимися по линейному закону от цвета фона до цвета объекта. (Изменение должно начаться с индекса, кратного 16.) Затем производится очистка цветового буфера первым из этих 16 цветов гребенки, и рисуются точки или линии с использованием данного набора индексов. В листинге 6.5 представлена программа, демонстрирующая создание цветовых гребенок для рисования сглаженных линий в режиме индексации цвета. В этом примере создаются два набора: один для зеленого цвета и другой для синих теней.

Листинг 6.5. Сглаживание в режиме индексации цвета: aaindex.c

```
#define RAMPSIZE 16
#define RAMP1START 32
#define RAMP2START 48

static float rotAngle = 0.;

/* Инициализируем сглаживание для режима индексации цвета,
 * включая загрузку зеленой гребенки, начиная с индекса
 * RAMP1START, и синей гребенки, начиная с индекса RAMP2START.
 * Количество цветов в наборе должно быть кратно 16
 */

void init(void)
{
    int i ;
    for (i = 0; i < RAMPSIZE; i++) {
        GLfloat shade;
        shade = (GLfloat) i/(GLfloat) RAMPSIZE;
        glutSetColor(RAMP1START+(GLint)i, 0., shade, 0.);
        glutSetColor(RAMP2 START+(GLint)i, 0., 0., shade);
    }
    glEnable(GL_LINE_SMOOTH);
    glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glLineWidth(1.5);

    glClearIndex((GLfloat) RAMP1START);
}

/* Рисуем две диагональные линии в виде буквы X */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glIndexi (RAMP1START);
```

продолжение ⇩

Листинг 6.5 (*продолжение*)

```
    glPushMatrix();
    glRotatef(-rotAngle, 0.0, 0.0, 0.1);
    glBegin(GL_LINES);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    glPopMatrix();

    glIndexi(RAMP2START);
    glPushMatrix();
    glRotatef(rotAngle, 0.0, 0.0, 0.1);
    glBegin(GL_LINES);
        glVertex2f(0.5, 0.5);
        glVertex2f(-0.5, -0.5);
    glEnd();
    glPopMatrix();

    glFlush();
}
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D(-1.0, 1.0, -1.0*(GLfloat)h/(GLfloat)w,
            1.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D(-1.0*(GLfloat)w/(GLfloat)h, 1.0*(GLfloat)w/(GLfloat)h,
            -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'r':
        case 'R':
            rotAngle += 20.;
            if (rotAngle >= 360.) rotAngle = 0.;
            glutPostRedisplay();
            break;
        case 27:    /* Клавиша Esc */
            exit(0);
            break;
        default:
            break;
    }
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
```

```
    glutInitDisplayMode(GLUT_SINGLE | GLUT_INDEX);
    glutInitWindowSi ze(200, 200);
    glutCreateWindow(argv[0]);
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Так как цвет меняется от цвета фона к цвету объекта, сглаженные линии выглядят правильно лишь тогда, когда они рисуются на фоне. При рисовании синей линии она затирает часть зеленой линии в точке пересечения. Для того чтобы исправить это упущение, нужно перерисовать область пересечения линий с помощью зелено-синей гребенки (от цвета линии в видеобуфере до цвета рисуемой линии). Однако это влечет дополнительные расчеты и обычно не имеет смысла, поскольку область пересечения невелика по размеру. Заметьте, что в режиме RGBA проблемы, в общем-то, и нет, так как цвета рисуемых объектов смешиваются с цветами, хранящимися в видеобуфере.

При рисовании выровненных точек и линий в режиме индексации цвета может потребоваться включение теста глубины. В приведенном примере тест глубины выключен по той причине, что обе линии лежат в одной *z*-плоскости. Однако при рисовании трехмерных сцен буфер глубины необходим, так как конечные цвета пикселов соответствуют ближайшим объектам.

Метод, описанный в разделе «Трехмерное смешивание с использованием буфера глубины», применим и для совмещения сглаженных точек и линий с несглаженными многоугольниками. Для этого сначала рисуем многоугольники, затем переводим буфер глубины в состояние «только для чтения» и рисуем точки и линии. Пусть точки и линии «пересекаются», это будет не видно за закрывающими пересечения многоугольниками.

Упражнение

Возьмите одну из написанных ранее программ типа руки робота или Солнечной системы из раздела «Примеры объединения нескольких преобразований» главы 3 и нарисуйте каркасные объекты со сглаживанием. Попробуйте сделать это как в режиме RGBA, так и в режиме индексации цвета. Также посмотрите на результат изменения толщины линий и размеров точек.

Сглаживание геометрических примитивов с помощью мультивыборки

Усреднение по множественной выборке, или, короче, множественная выборка, или, еще короче, мультивыборка (multisampling) — технология, привлекающая дополнительную информацию о цвете, глубине и трафарете при сглаживании примитивов OpenGL: точек, линий, многоугольников, битовых образов и изображений. При усреднении по множественной выборке в определении цвета, глуби-

ны и координат текстур фрагмента участвуют цвета, глубины и координаты текстур других фрагментов и пикселов, согласно размеру выборки. Вычисления проводятся не относительно неподвижного центра каждого пиксела (выборка из одного элемента), а отталкиваются от нескольких отсчетов (множественная выборка). Степень покрытия примитивом пиксела рассчитывается не по альфа-значениям, а на основе содержимого буфера мультивыборки.

Мультивыборка особенно хороша при сглаживании ребер многоугольников, так как она не требует строгого порядка обработки. (Но порядок рисования полупрозрачных объектов, подразумевающих использование альфа-компонентов, влияет на цвет результата.) В случаях, считающихся трудными по определению, наподобие пересечения или прилегания многоугольников, мультивыборки следует опасаться.

Мультивыборку очень просто добавить в приложение. Это делается в три шага:

1. Инициализация окна для поддержания множественной выборки. Это делается командой GLUT:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_MULTISAMPLE);
```

2. После открытия окна необходимо убедиться, доступна ли мультивыборка. То, что с помощью библиотеки GLUT можно создать окно «на любой вкус», еще не доказательство доступности мультивыборки в данной реализации. Если запрос параметра состояния `GL_SAMPLE_BUFFERS` возвращает значение 1, а `GL_SAMPLES` — возвращает значение больше единицы, мультивыборка поддержана. (Запрос со значением `GL_SAMPLES` возвращает количество элементов выборки. Если оно равно единице, мультивыборка невозможна.)

```
Glint bufs, samples;
glGetIntegerv(GL_SAMPLE_BUFFERS, &bufs);
glGetIntegerv(GL_SAMPLES, &samples);
```

3. Для активизации мультивыборки используется команда:

```
glEnable(GL_MULTISAPLE);
```

Программа из листинга 6.6 рисует два набора примитивов, демонстрируя эффект от применения мультивыборки. Процедура `init()` проверяет, доступен ли режим множественной выборки, а затем создает два списка отображения: один в виде «солнечного колеса» из линий различной толщины и треугольников (закрашенных многоугольников), и второй с «шахматным» фоном. В процедуре `display()` выводится «солнечное колесо» в двух видах: с мультивыборкой (слева) и без нее (справа). Таким образом, можно сравнить ступенчатость двух объектов. Контрастный фон иногда акцентирует зубчатость, а иногда скрадывает ее. С помощью клавиши b можно перерисовать изображение — с «шахматным» фоном или без него.

Листинг 6.6. Применение мультивыборки: multisamp.c

```
static int bgtoggle = 1;

/*
* Выводим параметры состояния, связанные с мультивыборкой.
* Создаем список отображения с "колесом" из линий и треугольников.
*/
```

```
void init(void)
{
   GLint buf, sbuf;
   int i, j;

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glGetIntegerv(GL_SAMPLE_BUFFERS_ARB, &buf);
   printf("Количество буферов выборок равно %d\n", buf);
   glGetIntegerv(GL_SAMPLES_ARB, &sbuf);
   printf("Количество элементов выборки равно %d\n", sbuf);

   glNewList(1, GL_COMPILE);
   for (i = 0; i < 19; i++) {
       glPushMatrix();
       glRotatef(360.0*(float)i/19.0, 0.0, 0.0, 1.0);
       glColor3f (1.0, 1.0, 1.0);
       glLineWidth((i%3)+1.0);
       glBegin(GL_LINES);
           glVertex2f(0.25, 0.05);
           glVertex2f(0.9, 0.2);
       glEnd();
       glColor3f(0.0, 1.0, 1.0);
       glBegin(GL_TRIANGLES);
           glVertex2f(0.25, 0.0);
           glVertex2f(0.9, 0.0);
           glVertex2f(0.875, 0.10);
       glEnd();
       glPopMatrix();
   }
   glEndList();

   glNewList(2, GL_COMPILE);
   glColor3f(1.0, 0.5, 0.0);
   glBegin(GL_QUADS);
       for (i = 0; i < 16; i++) {
           for (j = 0; j < 16; j++) {
               if (((i + j) % 2) == 0) {
                   glVertex2f(-2.0 + (i * 0.25), -2.0 + (j * 0.25));
                   glVertex2f(-2.0 + (i * 0.25), -1.75 + (j * 0.25));
                   glVertex2f(-1.75 + (i * 0.25), -1.75 + (j * 0.25));
                   glVertex2f(-1.75 + (i * 0.25), -2.0 + (j * 0.25));
               }
           }
       }
   glEnd();
   glEndList(); }

/* Рисуем два набора примитивов для сравнения результатов
 * при использовании мультивыборки и в ее отсутствие.
 * Этот код включает сглаживание и рисует один список отображения,
 * затем отключает сглаживание и рисует второй список отображения
 */
void display(void)
```

продолжение ⇩

Листинг 6.6 (*продолжение*)

```
{
    glClear(GL_COLOR_BUFFER_BIT);

    if (bgtoggle)
        glCallList(2);

    glEnable(GL_MULTISAMPLE_ARB);
    glPushMatrix();
    glTranslatef(-1.0, 0.0, 0.0);
    glCallList(1);
    glPopMatrix();

    glDisable(GL_MULTISAMPLE_ARB);
    glPushMatrix();
    glTranslatef(1.0, 0.0, 0.0);
    glCallList(1);
    glPopMatrix();
    glutSwapBuffers();
}
void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'b':
        case 'B':
            bgtoggle = !bgtoggle;
            glutPostRedisplay();
            break;
        case 27: /* Клавиша Esc */
            exit(0);
            break;
        default:
            break;
    }
}
```

В стандартной реализации возможности управления мультивыборкой весьма ограничены. Нельзя изменить число элементов выборки или задать (или даже узнать) их местоположение.

Если мультивыборка включена и ее буфер присутствует, то точки, линии и многоугольники «генерируют» фрагменты, для которых выполняется сглаживание. Например, примитив точки большого размера округляется, а не выглядит как квадрат; вне зависимости от того, включен или выключен режим `GL_POINT_SMOOTH`. Состояния `GL_LINE_SMOOTH` и `GL_POLYGON_SMOOTH` также игнорируются. Другие атрибуты примитивов, такие как размер точки и толщина линии, множественной выборкой не блокируются.

Альфа-компонент и покрытие мультивыборки

По умолчанию мультивыборка вычисляет степень покрытия тех фрагментов, которые не зависят от значения альфа-компонента. Однако есть специальные режи-

мы, где альфа-значения для фрагментов участвуют в расчете покрытия. Ниже перечислены эти режимы:

- `GL_SAMPLE_ALPHA_TO_COVERAGE` использует значение альфа-компонента фрагмента для вычисления зоны покрытия;
- `GL_SAMPLE_ALPHA_TO_ONE` устанавливает значение альфа-компонента фрагмента в максимально возможное значение — единицу и использует это значение при вычислении покрытия;
- `GL_SAMPLE_COVERAGE` — набор значений, установленных функцией `glSampleCoverage()`, комбинируется (операцией И) с вычисленным значением покрытия. Кроме того, этот режим может быть «инвертирован» посредством установки признака *invert* в функции `glSampleCoverage()`.

void `glSampleCoverage`(GLclampf *value*, GLboolean *invert*);

Устанавливает значения параметров, влияющих на интерпретацию значений альфа-компонента при вычислении покрытия мультивыборки. Аргумент *value* — значение покрытия, используемое, если включен режим `GL_SAMPLE_COVERAGE` или `GL_ALPHA_TO_COVERAGE`. Аргумент *invert* — значение логического типа, определяющее, должно ли промежуточное значение покрытия *value* быть поразрядно инвертировано (логическое И).

Сглаживание границ многоугольников

Устранение ступенчатости ребер закрашенных многоугольников аналогично сглаживанию точек и линий. Если многоугольники имеют общие границы, необходимо смешать соответствующие значения цветов. Это можно сделать методом, предлагаемым в данном разделе; другой вариант — сглаживание всей сцены с помощью буфера накопления. Работа с буфером накопления (см. главу 10) проще с точки зрения программиста, но требует большего объема вычислений и поэтому замедляет рисование. С другой стороны, как вы дальше увидите, описываемый метод более громоздкий.

ПРИМЕЧАНИЕ

Само собой, сглаживание точек или линий выполняется, если многоугольник рисуется в виде точек его вершин или рамки, то есть включен режим GL_POINT или GL_LINE в функции glPolygonMode(). Цель этого раздела — показать сглаживание именно закрашенного многоугольника (режим GL_FILL).

Теоретически сглаживание многоугольника можно выполнять и в режиме RGBA, и в режиме индексации цвета. Однако пересечение многоугольников влияет на сглаживание больше, чем совпадение точек и пересечение линий, поэтому порядок рисования и точность смешивания становятся более критичными. Фактически они настолько критичны, что при устранении ступенчатости для более чем одного многоугольника необходимо упорядочить многоугольники от ближнего к дальнему, а затем использовать функцию `glBlendFunc()` со значениями аргументов `GL_SRC_ALPHA_SATURATE` для источника и `GL_ONE` для получателя. Соответственно, устранение ступенчатости многоугольников в режиме индексации цвета обычно не используется.

Чтобы выровнять цвет границ многоугольников в режиме RGBA, а значит, определить степень покрытия для границ многоугольника, нужно использовать значения альфа-компонента. Включение сглаживания многоугольника выполняется командой `glEnable(GL_POLYGON_SMOOTH)`. Тогда пикселам на границах многоугольника будут присваиваться дробные значения альфа-компонента, основанные на зоне покрытия, как если бы эти границы были просто сглаженными линиями. Также, по желанию, можно задать значение для `GL_POLYGON_SMOOTH_HINT`.

Теперь необходимо «смешать» накладывающиеся границы. Вначале выключим буфер глубины, так как требуется управлять рисованием пикселов, совпадающих по положению. Затем установим коэффициенты смешивания в значения `GL_SRC_ALPHA_SATURATE` (для источника) и `GL_ONE` (для получателя). Мы хотим составить конечный цвет из цвета получателя и масштабированного цвета источника. Масштабирование осуществляется путем выбора наименьшего из входных значений альфа-компонента источника и вычтенного из единицы значения альфа-компонента получателя. Входящие пикселы получателя с большим значением альфа-компонента оказывают малое влияние на результирующий цвет, поскольку значение «1 минус альфа» близко к нулю. В итоге с помощью этого метода пикселы на границах многоугольника должны измениться в соответствии с цветами пикселов другого многоугольника, рисуемого позже. Наконец, нужно отсортировать многоугольники в сцене в порядке от ближнего к дальнему, перед их рисованием.

Туман

Компьютерные изображения иногда кажутся нереалистично четкими и ярко выраженными. Устранение ступенчатости делает объекты более реалистичными за счет размытия их границ. Кроме того, изображение можно приблизить к реальности посредством добавления *тумана*, который заставляет объекты постепенно «растворяться» при удалении. «Туман» — это общий термин, описывающий различные формы атмосферных явлений. Он может употребляться при моделировании дымки, мглы, смога или загрязнения воздуха (см. цветной рисунок 9). Туман необходим в приложениях, где требуется создать ограниченную видимость. Он часто используется в авиасимуляторах.

Когда туман включен, объекты, располагающиеся вдали от точки обзора, начинают растворяться в цвете тумана. Можно управлять плотностью тумана, которая определяет степень таяния объектов при увеличении расстояния, и его цветом. Кроме того, можно явно определять координаты на основе вершин для вычисления расстояния тумана вместо автоматического вычисления значения глубины.

Туман поддерживается как в режиме RGBA, так и в режиме индексации цвета, с некоторой разницей в вычислениях. Так как туман применяется после матричных преобразований, освещения и текстурирования, он оказывает действие на преобразованные, освещенные и текстурированные объекты. Обратите внимание, что туман способен увеличить производительность больших программ моделирования, поскольку можно не рисовать объекты, скрытые им.

Все виды геометрических примитивов могут быть затуманены, включая точки и линии. Использование тумана для точек и линий также называется *воздушной перспективой* (depth-cuing) (как показано на цветном рисунке 2). Этот эффект находит применение в молекулярном моделировании и других приложениях.

Применение тумана

Моделировать туман чрезвычайно просто. Он включается с помощью команды `glEnable(GL_FOG)`, а затем с помощью функции `glFog*()` задаются цвет и выражение, управляющее плотностью тумана. При желании можно задействовать значение `GL_FOG_HINT` в функции `glHint()`, как описывалось в табл. 6.3. Программа из листинга 6.7 рисует пять красных сфер, находящихся на различных расстояниях от точки просмотра. Нажатием клавиши f выбирается одно из трех уравнений, описанных в следующем разделе.

Листинг 6.7. Пять затуманенных сфер в режиме RGBA: fog.c

```
static GLint fogMode;
static void init(void)
{
   GLfloat position[] = { 0.5, 0.5, 3.0, 0.0 };

   glEnable(GL_DEPTH_TEST);

   glLightfv(GL_LIGHT0, GL_POSITION, position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   {
       GLfloat mat[3] = {0.1745, 0.01175, 0.01175};
       glMaterialfv(GL_FRONT, GL_AMBIENT, mat);
       mat[0] = 0.61424; mat[1] = 0.04136; mat[2] = 0.04136;
       glMaterialfv(GL_FRONT, GL_DIFFUSE, mat);
       mat[0] = 0.727811; mat[1] = 0.626959; mat[2] = 0.626959;
       glMaterialfv(GL_FRONT, GL_SPECULAR, mat);
       glMaterialf(GL_FRONT, GL_SHININESS, 0.6*128.0);
   }

   glEnable(GL_FOG);
   {
       GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};

       fogMode = GL_EXP; glFogi(GL_FOG_MODE, fogMode);
       glFogfv(GL_FOG_COLOR, fogColor);
       glFogf(GL_FOG_DENSITY, 0.35);
       glHint(GL_FOG_HINT, GL_DONT_CARE);
       glFogf(GL_FOG_START, 1.0);
       glFogf(GL_FOG_END, 5.0);
   }
   glClearColor(0.5, 0.5, 0.5, 1.0);  /* цвет тумана */
}

static void renderSphere(GLfloat x, GLfloat y, GLfloat z)
{
   glPushMatrix();
```

продолжение ⇩

Листинг 6.7 (*продолжение*)

```
    glTranslatef(x, y, z);
    glutSolidSphere(0.4, 16, 16);
    glPopMatrix();
}
/* Процедура display() рисует 5 сфер на разной глубине
 */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    renderSphere(-2., -0.5, -1.0);
    renderSphere(-1., -0.5, -2.0);
    renderSphere(0., -0.5, -3.0);
    renderSphere(1., -0.5, -4.0);
    renderSphere(2., -0.5, -5.0);
    glFlush();
}
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.5, 2.5, -2.5*(GLfloat)h/(GLfloat)w,
           2.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho(-2.5*(GLfloat)w/(GLfloat)h, 2.5*(GLfloat)w/(GLfloat)h, -2.5,
           2.5, -10.0, 10.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
}
void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'f':
        case 'F':
            if (fogMode == GL_EXP) {
                fogMode = GL_EXP2;
                printf("Режим тумана GL_EXP2\n");
            }
            else if (fogMode == GL_EXP2) {
                fogMode = GL_LINEAR;
                printf("Режим тумана GL_LINEAR\n");
            }
            else if (fogMode == GL_LINEAR) {
                fogMode = GL_EXP;
                printf("Режим тумана GL_EXP\n");
            }
            glFogi(GL_FOG_MODE, fogMode);
            glutPostRedisplay();
            break;
        case 27:
            exit(0);
```

```
            break;
        default:
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow(argv[0]);
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Уравнения тумана

Смешивание цвета тумана с цветом входящего фрагмента выполняется согласно коэффициенту смешивания. Этот коэффициент, f, вычисляется по одному из трех уравнений и ограничивается диапазоном [0, 1]:

$$f = e^{-(density \times z)} \quad (\texttt{GL_EXP});$$

$$f = e^{-(density \times z)^2} \quad (\texttt{GL_EXP2});$$

$$f = \frac{end - z}{end - start} \quad (\texttt{GL_LINEAR}).$$

В этих трех уравнениях z — расстояние в координатах глаза между точкой обзора и центром фрагмента. Им можно управлять с помощью основанных на вершинах координат тумана, как описано в разделе «Координаты тумана». Значения *density*, *start* и *end* устанавливаются функцией `glFog*()`. Использование коэффициента f зависит от режима работы — RGBA или индексации цвета — и описано в следующих подразделах.

void `glFog{if}`(GLenum *pname*, *TYPE param*);
void `glFog{if}v`(GLenum *pname*, *TYPE *param*);
Задает параметры и функции для вычисления тумана. Если значение *pname* равно `GL_FOG_MODE`, *param* выбирается из трех коэффициентов тумана: `GL_EXP` (по умолчанию), `GL_EXP2` или `GL_LINEAR`. Если *pname* равно `GL_FOG_DENSITY`, `GL_FOG_START` или `GL_FOG_END`, то *param* — собственно значение (или указатель — для векторной версии команды) *density*, *start* или *end*. (Значения по умолчанию равны 1, 0 и 1 соответственно.) В режиме RGBA для *pname* действует константа `GL_FOG_COLOR`, тогда *param* указывает на четыре значения цветовых компонентов (RGBA) тумана. Соответствующее значение для *pname* в режиме индексированных цветов равно `GL_FOG_INDEX`, при этом аргумент *param* равен индексу цвета тумана.

На рис. 6.4 представлены графики уравнений плотности тумана для различных значений параметров.

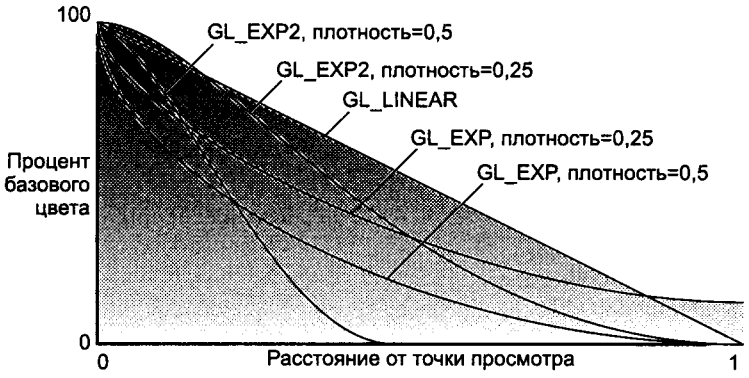


Рис. 6.4. Уравнения плотности тумана

Туман в режиме RGBA

В режиме RGBA формула для результирующего затуманенного цвета выглядит следующим образом:

$$C = f\,C_i + (1 - f)\,C_f,$$

где C_i представляет собой RGBA-цвет входящего фрагмента, а C_f — цвет тумана, заданный с помощью `GL_FOG_COLOR`.

Туман в режиме индексации цвета

В режиме индексированного цвета цвет с учетом тумана вычисляется по формуле

$$I = I_i + (1 - f)\,I_f,$$

где I_i — индекс цвета входящего фрагмента, а I_f — индекс цвета тумана, заданный с помощью `GL_FOG_INDEX`.

Для использования тумана в режиме индексированных цветов следует загрузить требуемый набор значений в палитру. Первый цвет в ней — цвет объекта без тумана, а последний — цвет полностью затуманенного объекта. Возможно, понадобится функция `glClearIndex()` для установки индекса цвета фона таким образом, чтобы он соответствовал последнему цвету в наборе. Этим способом можно заставить полностью затуманенные объекты сливаться с фоном. Аналогично, перед рисованием объектов с помощью вызова `glIndex*()` передается индекс первого цвета в палитре (незатуманенный цвет). Наконец, чтобы применить туман к различно окрашенным объектам в сцене, необходимо создать несколько цветовых наборов и вызвать `glIndex*()` перед каждым рисованием объекта для установки индекса текущего цвета на первый цвет соответствующего набора. Программа, приведенная в листинге 6.8, показывает, как создать нужные условия, а затем применить туман в режиме индексированных цветов.

Листинг 6.8. Туман в режиме индексирования цветов: fogindex.c

```
/* Инициализируем карту цветов и туман.
 * Устанавливаем цвет очистки экрана равным
 * последнему цвету в цветовом наборе
 */
#define NUMCOLORS 32
#define RAMPSTART 16

static void init(void)
{
   int i;
   glEnable(GL_DEPTH_TEST);
   for (i = 0; i < NUMCOLORS; i++) {
      GLfloat shade;
      shade = (GLfloat) (NUMCOLORS-i)/(GLfloat) NUMCOLORS;
      glutSetColor(RAMPSTART + i, shade, shade, shade);
   }
   glEnable(GL_FOG);

   glFogi(GL_FOG_MODE, GL_LINEAR);
   glFogi(GL_FOG_INDEX, NUMCOLORS);
   glFogf(GL_FOG_START, 1.0);
   glFogf(GL_FOG_END, 6.0);
   glHint(GL_FOG_HINT, GL_NICEST);
   glClearIndex((GLfloat) (NUMCOLORS+RAMPSTART-1));
}

static void renderSphere(GLfloat x, GLfloat y, GLfloat z)
{
   glPushMatrix();
   glTranslatef(x, y, z);
   glutWireSphere(0.4, 16, 16);
   glPopMatrix();
}

/* Ghjwtlehf display() рисует 5 сфер на различной глубине
 */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glIndexi(RAMPSTART);
   renderSphere(-2., -0.5, -1.0);
   renderSphere(-1., -0.5, -2.0);
   renderSphere(0., -0.5, -3.0);
   renderSphere(1., -0.5, -4.0);
   renderSphere(2., -0.5, -5.0);

   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, w, h);
```

продолжение ⇩

Листинг 6.8 (*продолжение*)

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.5, 2.5, -2.5*(GLfloat)h/(GLfloat)w,
            2.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho(-2.5*(GLfloat)w/(GLfloat)h, 2.5*(GLfloat)w/(GLfloat)h, -2.5,
            2.5, -10.0, 10.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27:
            exit(0);
    }
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_INDEX | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow(argv[ 0 ]);
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Координаты тумана

Вы уже видели, что в уравнения тумана входит координата z:

$f = e^{-(density \times z)}$ (GL_EXP);

$f = e^{-(density \times z)^2}$ (GL_EXP2);

$f = \frac{end - z}{end - start}$ (GL_LINEAR).

По умолчанию z вычисляется автоматически как расстояние от глаза до фрагмента, но иногда требуется контролировать ее значения вручную. Благодаря этому можно получить уравнения тумана, отличные от предлагаемых OpenGL. Например, чтобы создать авиасимулятор с приземным туманом, важно смоделировать сгущение тумана ближе к уровню моря.

В OpenGL версии 1.4 доступно явное задание значений z на основе вершин с помощью команды `glFog(GL_FOG_COORDINATE_SOURCE, GL_FOG_CORDINATE)`. В ре-

жиме явного указания координат тумана можно определить координату тумана в каждой вершине посредством команды `glFogCoord*()`.

> void `glFogCoord{fd}`(*TYPE z*);
> void `glFogCoord{fd}v`(*TYPE *z*);
> Устанавливает текущую координату тумана в *z*. Если `GL_FOG_COORDINATE` — исходная текущая координата тумана, текущая координата тумана используется текущим уравнением тумана (`GL_LINEAR`, `GL_EXP` или `GL_EXP2`) для вычисления тумана.
> Значения *z* должны быть положительными числами, представляющими собой расстояние в координатах глаза. Необходимо избегать отрицательных координат, так как в этом случае результатом вычислений могут стать неправильные цвета.

В случае геометрических примитивов координаты тумана могут быть интерполированы для каждого фрагмента.

Программа из листинга 6.9 рисует треугольник и позволяет задавать координаты тумана для каждой вершины цифровыми клавишами. В режиме явного указания координат тумана перемещение точки обзора вперед и назад (клавиши `f` и `b`) не изменяет координат тумана и поэтому не сказывается на цвете вершин. При выключении этого режима (клавиша `c`) перемещение точки обзора будет влиять на цвета, вычисленные для тумана.

Листинг 6.9. Координаты тумана: fogcoord.c

```
static GLfloat f1,  f 2,  f 3;

/* Инициализируем туман
 */
static void init(void)
{
   GLfloat fogColor[4] = {0.0, 0.25, 0.25, 1.0};
   f1 = 1.0f;
   f2 = 5.0f;
   f3 = 10.0f;

   glEnable(GL_FOG);
   glFogi (GL_FOG_MODE, GL_EXP);
   glFogfv (GL_FOG_COLOR, fogColor);
   glFogf (GL_FOG_DENSITY, 0.25);
   glHint (GL_FOG_HINT, GL_DONT_CARE);
   glFogi(GL_FOG_COORDINATE_SOURCE, GL_FOG_COORDINATE);
   glClearColor(0.0, 0.25, 0.25, 1.0);  /* цвет тумана */
}

/* Процедура display() рисует треугольник в углу */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   glColor3f (1.0f, 0.75f, 0.0f);
```

продолжение ⇙

Листинг 6.9 (*продолжение*)

```
    glBegin (GL_TRIANGLES);
    glFogCoordf (f1);
    glVertex3f (2.0f, -2.0f, 0.0f);
    glFogCoordf (f 2);
    glVertex3f (-2.0f, 0.0f, -5.0f);
    glFogCoordf (f 3 );
    glVertex3f (0.0f, 2.0f, -10.0f);
    glEnd();

    glutSwapBuffers();
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'C:
            glFogi(GL_FOG_COORDINATE_SOURCE, GL_FRAGMENT_DEPTH);
            glutPostRedisplay();
            break;
        case 'C :
            glFogi(GL_FOG_COORDINATE_SOURCE, GL_FOG_COORDINATE);
            glutPostRedisplay();
            break;
        case '1':
            f1 = f1 + 0.25;
            glutPostRedisplay();
            break;
        case '2':
            f2 = f2 + 0.25;
            glutPostRedisplay();
            break;
        case '3':
            f3 = f3 + 0.25;
            glutPostRedisplay();
            break;
        case '8':
            if (f1 > 0.25) {
                f1 = f1 - 0.25;
                glutPostRedisplay();
            }
            break;
        case '9':
            if (f2 > 0.25) {
                f2 = f2 - 0.25;
                glutPostRedisplay();
            }
            break;
        case '0':
            if (f3 > 0.25) {
                f3 = f3 - 0.25;
                glutPostRedisplay();
            }
```

```
            break;
        case 'b':
            glMatrixMode (GL_MODELVIEW);
            glTranslatef (0.0, 0.0, -0.25);
            glutPostRedisplay();
            break;
        case 'f':
            glMatrixMode (GL_MODELVIEW);
            glTranslatef (0.0, 0.0, 0.25);
            glutPostRedisplay();
            break;
        case 27:
            exit(0);
            break;
        default:
            break;
    }
}
```

Параметры точки

В некоторых случаях требуется нарисовать объекты, напоминающие маленькие круги или сферы, не обращаясь к помощи неэффективных полигональных моделей. Например, в авиасимуляторе рисуются посадочные огни — при приближении самолета к взлетно-посадочной полосе посадочные огни становятся больше и ярче. Или в системах частиц — при рисовании капель жидкости (таких как дождь или в видеоиграх — брызги крови) необходимо моделировать соответствующие им явления.

Посадочные огни или капли жидкости можно нарисовать с помощью точечных примитивов, если изменять видимый размер и яркость точек. Сглаженные точки дают функции `glPointSize()` и `glEnable(GL_POINT_SMOOTH)`; это, конечно, шаг относительно решения, обыгрывающего расстояние туманом. Вместе с тем функция `glPointSize()` неприменима внутри операторных скобок `glBegin()`/`glEnd()`, поэтому приходится перерассчитать размер точек «на лету», а затем, ради улучшения производительности, перегруппировать точки исходя из размера.

Элегантное решение автоматического изменения размера точечных примитивов и, при необходимости, их яркости на основе информации о расстоянии до точки обзора предлагает управление параметрами точек. Для задания коэффициентов уравнения изменения размеров точек и их альфа-составляющих (управляющих яркостью) применяется функция `glPointParameterf*()`.

void `glPointParameterf`(GLenum *pname*, GLfloat *param*);
void `glPointParameterfv`(GLenum *pname*, GLfloat **param*);
Устанавливает значения, связанные с рисованием точечных примитивов. Если *pname* равно `GL_POINT_DISTANCE_ATTENUATION`, то параметр *param* представляет собой массив из трех значений вещественного типа (*a*, *b*, *c*), содержа-

щих коэффициенты константного, линейного и квадратичного изменения размера и яркости точки на основе расстояния, заданного в координатах глаза, *d*:

$$derivedSize = clamp\left(size \times \sqrt{\left(\frac{1}{a + b \times d + c \times d^2}\right)}\right).$$

Если значение *pname* равно GL_POINT_SIZE_MIN или GL_POINT_SIZE_MAX, значение *param* устанавливает пороговое значение (нижнюю или верхнюю границу соответственно), используемое предыдущим выражением для ограничения вычисленного размера точки.

Если включена мультивыборка и значение *pname* равно GL_FADE_THRESHOLD_SIZE, то *param* задает различные нижние границы (пороговые значения) размера точек. Если *derivedSize* < *threshold*, вычисляется коэффициент *fade* для корректировки альфа-значения точки с целью уменьшения ее яркости:

$$fade = \left(\frac{derivedSize}{threshold}\right)^2.$$

В случае GL_POINT_DISTANCE_ATTENUATION расстояние вычисляется аналогично коэффициентам ослабления локальных источников света. В листинге 6.10 приведена программа, в которой клавиша c, l или q переключает уравнение ослабления между постоянным, линейным и квадратичным коэффициентами. Нажатие клавиши f или b перемещает наблюдателя вперед или назад, что увеличивает или уменьшает точки в линейном или квадратичном режиме ослабления.

Листинг 6.10. Параметры точки: pointp.c

```
static GLfloat constant[3]  = {1.0, 0.0, 0.0};
static GLfloat linear[3]  = {0.0, 0.12, 0.0};
static GLfloat quadratic[3]  = {0.0, 0.0, 0.01};
...
void keyboard(unsigned char key, int x, int y) {
 switch (key) {
 case 'c':
  glPointParameterfv (GL_POINT_DISTANCE_ATTENUATION, constant);
  glutPostRedisplay();
  break;
 case 'l':
  glPointParameterfv(GL_POINT_DISTANCE_ATTENUATION, linear);
  glutPostRedisplay();
  break;
 case 'q':
  glPointParameterfv(GL_POINT_DISTANCE_ATTENUATION, quadratic);
  glutPostRedisplay();
  break;
 case 'b':
  glMatrixMode(GL_MODELVIEW);
  glTranslatef(0.0, 0.0, -0.5);
  glutPostRedisplay();
  break;
 case 'f':
  glMatrixMode(GL_MODELVIEW);
  glTranslatef(0.0, 0.0, 0.5);
```

```
    glutPostRedisplay();
    break;
...
```

Когда ослабление линейное или квадратичное, перемещение глаза слишком близко к точке может дополнительно увеличить выработанный диаметр точки — делением оригинального размера на правильную дробь. Для ослабления или предотвращения этого эффекта пользуйтесь постоянным коэффициентом ослабления или добавьте ограничивающие значения с помощью GL_POINT_SIZE_MAX.

Вы наверняка захотите получить круг, а не квадрат, поэтому надо включить сглаживание, как описано в разделе «Сглаживание точек или линий». Следующие строки кода делают это:

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Отклонение многоугольников по глубине

Если требуется выделить цветом границы сплошного объекта, можно нарисовать его с помощью многоугольника в режиме GL_FILL, а затем повторить рисование, но уже с другим цветом и в режиме GL_LINE. Однако поскольку линии и закрашенные многоугольники не растеризуются одинаковым способом, значения глубины, вычисленные для пикселов на линии, обычно не совпадают со значениями глубины для границ многоугольника даже между двумя одинаковыми вершинами. Выделенные цветом линии могут совпадать с границами соответствующих многоугольников и могут несколько отклоняться от них, напоминая стежки на швах одежды. Этот нежелательный эффект иногда и называют «стежками» (stitching), он визуально неприятен.

Он может быть устранен посредством отклонения многоугольников по глубине, которое добавляет некоторое смещение, чтобы разъединить совпадающие значения z и тем самым отделить границы многоугольника от оконтуривающей линии. (Буфер трафарета, описанный в разделе «Тест трафарета» главы 10, также годится для устранения стежков. Однако отклонение всегда работает быстрее.) Отклонение также применяется при маркировке поверхностей, рисовании изображений с удалением невидимых линий. Эта технология применима не только к линиям и закрашенным многоугольникам, но и к точкам.

Существует три различных способа отклонения, по одному для каждого типа растеризации: GL_FILL, GL_LINE и GL_POINT. Режим отклонения включается с помощью функции glEnable() со значениями аргумента GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE или GL_POLYGON_OFFSET_POINT. Кроме того, необходимо вызвать функцию glPolygonMode() для выбора текущего способа растеризации многоугольника.

void `glPolygonOffset`(GLfloat *factor*, GLfloat *units*);

Дополняет значение глубины каждого фрагмента вычисленным значением смещения. Смещение добавляется до теста глубины и перед записью значений глубины в буфер глубины. Значение смещения o вычисляется по следующей формуле:

$o = m \times \text{factor} + r \times \text{units}$,

где m — максимальное отклонение глубины многоугольника, а r — минимальное значение, гарантирующее получение видимого отличия в координатах окна. Значение r зависит от реализации.

Для сохранения хорошего качества сплошного объекта с выделенными цветом границами без появления каких-либо артефактов можно добавить положительное смещение к закрашенному объекту (отодвинуть его) или отрицательное смещение для каркаса (придвинуть его). Неминуем вопрос — какого смещения достаточно? К сожалению, ответ зависит от множества факторов, включая отклонение глубины каждого многоугольника и толщину линий каркаса.

OpenGL самостоятельно вычисляет отклонение глубины (рис. 6.5), но для правильного выбора значения параметра *factor* нужно представлять, что вкладывается в слова «угловой коэффициент глубины» (depth slope). Это — девиация z (глубины), деленная на девиацию координаты x или y вдоль многоугольника. Значение глубины вычисляется в оконных координатах и ограничивается диапазоном [0, 1]. Оценка максимального углового коэффициента (m в уравнении смещения) выполняется по формуле

$$m = \max\left\{\left|\frac{\partial V}{\partial s}\right|, \left|\frac{\partial V}{\partial t}\right|\right\}.$$

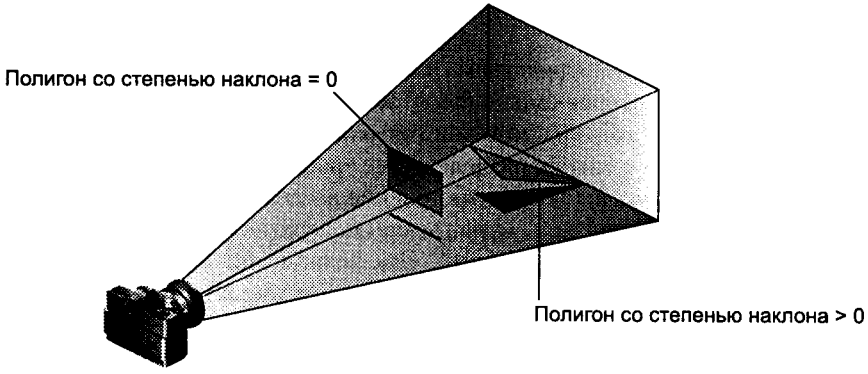


Рис. 6.5. Многоугольники и их угловые коэффициенты глубины

Для многоугольников, параллельных ближней и дальней плоскостям отсечения, угловой коэффициент равен нулю. Для многоугольников с наклоном, близким

к нулю, требуется небольшое константное значение смещения. Чтобы его получить, вызовите функцию `glPolygonOffset()` со значениями *factor* = 0.0 и *units* = 1.0.

Для многоугольников, образующих больший угол с плоскостями отсечения, угловой коэффициент значительно больше нуля и, соответственно, требуется большее смещение. Небольшого, ненулевого, значения параметра *factor*, такого как 0,75 или 1,0, достаточно для получения значений глубины и устранения нежелательных артефактов.

В листинге 6.11 представлена часть кода, который с помощью списка отображения (предположим, рисующего закрашенный объект) рисует объект с освещением, в режиме `GL_FILL`. Для смещения многоугольника параметры *factor* и *units* задаются равными 1.0 и 1.0. Эти значения гарантируют, что смещение будет достаточным для всех многоугольников в сцене независимо от наклона. Затем, с целью выделения цветом границ первого объекта, объект рисуется как каркас с отключенным смещением.

Листинг 6.11. Отклонение многоугольника для устранения артефактов: polyoff.c

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 1.0);
glCallList(list);
glDisable(GL_POLYGON_OFFSET_FILL);

glDisable(GL_LIGHTING);
glDisable(GL_LIGHT0);
glColor3f(1.0, 1.0, 1.0);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glCallList(list);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

В некоторых случаях «легкие в расчете» значения параметров *factor* и *units* (1.0 и 1.0) — не выход. Например, если толщина линий, выделяющих границы, больше 1, может потребоваться увеличить значение *factor*. Также, если значения глубины неравномерно преобразованы в оконные координаты в процессе перспективного проецирования (см. раздел «Измененная координата глубины» главы 3), меньшее смещение нужно для многоугольников, близких к ближней плоскости отсечения, а большее — для удаленных многоугольников. Опять же, эксперименты со значением параметра могут быть оправданы.

7

Списки отображения

Из этой главы вы узнаете:

- Что такое списки отображения и как использовать их в сочетании с командами непосредственного режима с целью наиболее эффективной организации и данных и улучшенной производительности
- Как извлечь максимальную эффективность из списков отображения, применяя их в нужное время и в надлежащих местах

Список отображения (display list) — иначе список команд или список вывода — это группа команд OpenGL, сохраненных для последующего выполнения. Когда список отображения вызывается на выполнение, команды обрабатываются в порядке их появления в списке. Большинство команд OpenGL может сохраняться в списке или обрабатываться в режиме непосредственного выполнения, для краткости далее именуемом *непосредственным* (immediate) режимом, то есть незамедлительно. Вы можете свободно смешивать непосредственный режим и списки отображения в рамках одной программы. В примерах, приводимых ранее, до сих пор использовался непосредственный режим. В этой главе обсуждается, что такое списки отображения и каким образом их лучше использовать. Соответственно, ее разделы посвящены следующим темам:

- **Для чего нужны списки отображения.** Объясняет, где используются списки отображения.
- **Пример списка отображения.** Содержит небольшой пример, демонстрирующий использование основных команд применительно к спискам отображения.
- **Философия проектирования списков отображения.** Говорит о возможностях (таких как присвоение им статуса нередактируемых), позволяющих добиться максимальной эффективности списков отображения.
- **Создание и выполнение списка отображения.** Содержит подробные описания команд для создания, обработки и удаления списков отображения.
- **Выполнение составных списков отображения.** Показывает, как обработать последовательность из нескольких списков, здесь используется небольшой набор символов в качестве примера.
- **Управление параметрами состояния с помощью списков отображения.** Иллюстрирует использование списков отображения для сохранения состояния и его восстановления с помощью команд OpenGL.

Для чего нужны списки отображения

Списки отображения позволяют улучшить производительность за счет использования запомненных для последующего выполнения команд OpenGL. Максимум отдачи дает кэширование команд в списке, если вы планируете перерисовывать один и тот же объект несколько раз или вы имеете набор изменяемых состояний, которые нужно воспроизводить многократно. С помощью списков отображения вы можете определить геометрию и/или изменяемое состояние однажды и повторить их столько раз, сколько нужно.

Случай, когда вы используете список отображения для единичного запоминания, проиллюстрируем примером рисования трехколесного велосипеда. Его два задних колеса имеют одинаковый размер, но требуют отдельной прорисовки, так как смещены друг относительно друга. Переднее колесо больше по диаметру и также находится в индивидуальной позиции. Эффективный способ передать вращение колес заключается в запоминании геометрии одного колеса в списке отображе-

ния и последующем трехкратном вызове этого списка. Вам нужно построить соответствующую модельно-видовую матрицу перед каждым вызовом списка для вычисления корректных размеров и положения каждого колеса.

Когда программы OpenGL запускаются удаленно с другого компьютера или по сети, особенно важно кэшировать команды в списке отображения.

В этом случае сервер является компьютером, отличным от хост-машины. (Информацию о клиент-серверной модели OpenGL см. в разделе «Что такое OpenGL» главы 1.)

Поскольку списки отображения являются частью состояния сервера и, следовательно, постоянно размещены на сервере, вы можете сократить расходы на передачу по сети повторно используемых данных, если сохраните повторно используемые команды в списке отображения.

В варианте локального запуска вы также можете добиться повышения производительности за счет помещения наиболее часто используемых команд в список отображения. Некоторые аппаратные графические средства способны сохранять списки отображения в специально выделенной области памяти или запоминать данные в форме, обеспечивающей максимум совместимости с программным обеспечением. (Более подробно вопросы оптимизации обсуждаются в разделе «Философия проектирования списков отображения».)

Пример списка отображения

Список отображения — это удобный и эффективный способ именования и организации набора команд OpenGL. Например, допустим, что вы хотите нарисовать тор и просмотреть его с различных углов зрения. Лучший способ сделать это — сохранить тор в списке отображения. Тогда всякий раз при изменении представления вам нужно будет изменить модельно-видовую матрицу и выполнить список отображения, рисующий тор. Листинг 7.1 иллюстрирует сказанное.

Листинг 7.1. Создание списка отображения: torus.c

```
GLuint theTorus;
/* Рисование тора */
static void torus(int numc, int numt)
{
   int i, j, k;
   double s, t, x, y, z, twopi;
   twopi = 2 * (double)M_PI;
   for (i = 0; i < numc; i++) {
      glBegin(GL_QUAD_STRIP);
      for (j = 0; j <= numt; j++) {
         for (k = 1; k >= 0; k--) {
            s = (i + k) % numc + 0.5;
            t = j % numt;
            x = (1+.1*cos(s*twopi/numc))*cos(t*twopi/numt);
            y = (1+.1*cos(s*twopi/numc))*sin(t*twopi/numt);
            z = .1 * sin(s*twopi/numc);
```

```
            glVertex3f(x, y, z);
        }
      }
      glEnd();
    }
}

/* Создание списка отображения с тором и инициализация */
static void init(void)
{
  theTorus = glGenLists(1);
  glNewList(theTorus, GL_COMPILE);
  torus(8, 25);
  glEndList();
  glShadeModel(GL_FLAT);
  glClearColor(0.0, 0.0, 0.0, 0.0);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 1.0, 1.0);
  glCallList(theTorus);
  glFlush();
}

void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(30, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0, 0, 10, 0, 0, 0, 0, 1, 0);
}

/* Поворот относительно оси x при нажатии клавиши «x»; вращение вокруг оси y
при нажатии клавиши «y»; при нажатии клавиши «i» тор возвращается
в оригинальное представление */
void keyboard(unsigned char key, int x, int y)
{
  switch (key) {
     case 'x':
     case 'X':
        glRotatef(30., 1.0, 0.0, 0.0);
        glutPostRedisplay();
        break;
     case 'y':
     case 'Y':
        glRotatef(30., 0.0, 1.0, 0.0);
        glutPostRedisplay();
        break;
```

продолжение ⇩

Листинг 7.1 (*продолжение*)

```
      case 'i':
      case 'I':
         glLoadIdentity();
         gluLookAt(0, 0, 10, 0, 0, 0, 0, 1, 0);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
   }
}

int main(int argc, char **argv)
{
   glutInitWindowSize(200, 200);
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

В первую очередь посмотрим на процедуру `init()`. Она создает список отображения для тора и устанавливает начальные состояния переменных рендеринга. Заметьте, что подпрограмма для рисования тора (`torus()`) обрамлена командами `glNewList()` и `glEndList()`, которые определяют начало и конец списка отображения соответственно. Аргумент *listName* метода `glNewList()` — это целочисленный индекс, сгенерированный командой `glGenLists()`, уникальным образом идентифицирующий список отображения.

Пользователь может вращать тор относительно оси x или y, нажимая клавиши `X` или `Y`, когда окно получает фокус клавиатуры. Всякий раз при этом происходит вызов функции `keyboard()`, которая накладывает матрицу поворота на 30° (относительно осей x и y) на текущую модельно-видовую матрицу. Команда `glutPostRedisplay()` своим вызовом влечет перерисовку тора функцией `display()` в цикле `glutMainLoop()` после того, как будут обработаны другие события.

По нажатии клавиши `I` функция `keyboard()` восстанавливает начальную модельно-видовую матрицу и возвращает тор в оригинальное положение. Процедура `display()` очень проста. Она очищает окно, а затем вызывает команду `glCallList()` для выполнения команд списка отображения. Если бы мы не использовали списки отображения, процедура `display()` выдавала бы распоряжение на перерисовку тора при каждом своем вызове.

Список отображения содержит исключительно OpenGL-команды. В листинге 7.1 в таком списке сохранены только вызовы `glBegin()`, `glVertex()` и `glEnd()`. Их

параметры являются *вычисляемыми*, а результирующие значения копируются в список отображения перед его созданием. Все тригонометрические расчеты производятся только единожды, с целью увеличить производительность рендеринга. Значения в списке отображения не могут быть изменены позднее, а команда, помещенная в список, — из него удалена. Также вы не можете добавить новые команды в список после того, как он был описан. То есть вы вправе удалить имеющийся список и создать вместо него новый, но не редактировать уже определенный.

ПРИМЕЧАНИЕ

Списки отображения хорошо сочетаются с командами библиотеки GLU, поскольку эти операции в конечном счете разбиваются на команды OpenGL нижнего уровня, которые проще сохранить в списке отображений. Использование списков отображения совместно с GLU особенно важно для оптимизации производительности «мозаичных» команд (tessellators) библиотек GLU (см. главу 11) и NURBS (глава 12).

Философия проектирования списков отображения

С целью поднятия производительности команды списков отображения OpenGL кэшируются, причем так, что однажды созданный список отображения не может быть модифицирован. В противном случае выигрыш в производительности был бы сведен к минимуму за счет поиска по списку и лишних операций управления памятью. Если бы части списка отображения были изменяемыми, распределение и высвобождение памяти привело бы к дефрагментации памяти. Любые модификации, внесенные реализацией OpenGL в порядок команд списка отображения с целью повышения эффективности рендеринга, все равно потребовали бы отката. Также вероятны были бы трудности с доступом к списку, кэшированному где-нибудь в сети или на системной шине.

Способ, которым достигается оптимизация команд в списке отображения, может варьироваться от реализации к реализации. Например, такая простая команда, как `glRotate*()`, может выполняться по-разному быстро, если она помещена в список отображения, поскольку вычисления для получения матрицы вращения далеко не тривиальны (они могут включать в себя извлечение квадратного корня и тригонометрические функции). В списке отображения, однако, требуется сохранение только конечной матрицы вращения, так что время выполнения любой команды вращения из состава списка отображения обусловлено тем, насколько быстро обрабатывает аппаратное обеспечение команду `glMultMatrix*()`. Изощренные реализации OpenGL могут даже объединять смежные команды преобразования в одном произведении матриц.

Хотя не гарантируется, что ваша реализация OpenGL оптимизирует списки отображения для любых специфических случаев, выполнение списка отображения никогда не будет медленнее, чем выполнение тех же команд в индивидуальном порядке. Вместе с тем имеются некоторые издержки, обусловленные процессом перехода к списку отображения. Если такой список невелик, эти издержки способны свести на нет все преимущества списочной организации. Наиболее пред-

почтительные места для оптимизации перечислены ниже, со ссылками на главы, где эти темы рассматриваются:

- Матричные операции (глава 3). Большинство матричных операций требуют вычисления обратных матриц. В зависимости от реализации OpenGL, обе матрицы — расчетная и обратная — сохраняются или не сохраняются в списке отображения.
- Растровые двоичные карты и изображения (глава 8). Формат, в котором вы определяете растровые данные, скорее всего, не будет идеальным в плане совместимости с аппаратными средствами. При компиляции списка отображения OpenGL может преобразовать данные в представление, предпочтительное для оборудования. Это оказывает существенный эффект на скорость рисования растровых символов, поскольку символьные строки обычно состоят из совокупности небольших образов (двоичных карт).
- Источники света, свойства материала и модели распространения света (глава 5). Когда вы отрисовываете сцену со сложными условиями освещения, вы можете изменять материалы для каждого элемента сцены. Задание материалов само по себе медленная операция, поскольку оно предполагает множество вычислений. Если вы поместите описание свойств материалов в список отображения, эти вычисления не будут выполняться всякий раз при переключении между материалами, так как сохранению подлежат лишь результаты вычислений; в итоге рендеринг освещенной сцены может ускориться. (См. раздел «Инкапсуляция изменений режима» для получения более подробной информации об использовании списков отображения для изменения таких характеристик, как условия освещения.)
- Шаблоны штриховки многоугольников (глава 2).

ПРИМЕЧАНИЕ

Для оптимизации текстурных изображений вам будет лучше сохранить данные текстур в объектах текстур, а не в списках отображений.

Некоторые из перечисленных здесь команд, используемых для указания свойств, являются контекстно-зависимыми, и вы должны принимать это во внимание, если хотите добиться максимальной производительности. Например, если активен режим `GL_COLOR_MATERIAL`, для отдельных свойств материала будет отслеживаться *текущий* цвет (см. главу 5). Любые вызовы `glMaterial*()`, устанавливающие свойства материала, будут игнорироваться. Увеличить производительность можно за счет сохранения переменных состояния вместе с параметрами геометрии. Допустим, вы хотите применить преобразования к части геометрических объектов, а затем вывести общий результат. Тогда ваш код может выглядеть следующим образом:

```
glNewList(1, GL_COMPILE);
draw_some_geometric_objects();
glEndList();

glLoadMatrix(M);
glCallList(1);
```

Но если геометрические объекты должны быть трансформированы таким образом несколько раз, лучше сохранять матрицу в списке отображения. Например, если вы напишете следующий код, в отдельных реализациях вы получите дополнительное увеличение производительности за счет трансформации объектов каждый раз при их отрисовке:

```
glNewList(1, GL_COMPILE);
glLoadMatrix(M);
draw_some_geometric_objects();
glEndList();
glCallList(1);
```

Более приятная ситуация имеет место при рендеринге изображений. Как вы увидите в главе 8, вы можете модифицировать переменные состояния для передачи пикселов (pixel-transfer), и управлять характером попиксельной развертки изображений и двоичных карт. Если команды, устанавливающие эти переменные состояния, в списке отображения предшествуют определению изображений или двоичных карт, реализация может выполнить эти операции упредительно и поместить результаты в кэш.

Помните, что списки отображений не являются панацеей. Очень маленькие списки не всегда имеют смысл в связи с издержками на выполнение самого списка. Другой недостаток заключается в неизменности содержимого списка отображения. Чтобы производительность была максимальной, списки изображений OpenGL делаются постоянными и недоступными для чтения содержимого. Если приложение нуждается в поддержании используемых в списке данных вне этого списка (например, для последующей их обработки), это увеличивает требования к объему необходимой памяти.

Создание и выполнение списка отображения

Как вы уже видели, команды `glNewList()` и `glEndList()` используются для указания места начала и окончания определения списка отображения, а сам список вызывается на выполнение с помощью идентифицирующего его индекса, генерируемого `glCallList()`. В листинге 7.2 список отображения создается в подпрограмме `init()`. Этот список отображения содержит команды OpenGL для рисования красного треугольника. В процедуре `display()` список отображения выполняется 10 раз. Дополнительно в непосредственном режиме рисуется линия. Заметьте, что список отображения выделяет память для размещения команд и значений необходимых переменных.

Листинг 7.2. Использование списка отображения: list.c

```
GLuint listName;
static void init(void)
{
  listName = glGenLists(1);
  glNewList(listName, GL_COMPILE);
     glColor3f(1.0, 0.0, 0.0); /* текущий цвет – красный */
     glBegin(GL_TRIANGLES);
```

продолжение ⇩

Листинг 7.2 (*продолжение*)

```
            glVertex2f(0.0, 0.0);
            glVertex2f(1.0, 0.0);
            glVertex2f(0.0, 1.0);
         glEnd();
         glTranslatef(1.5, 0.0, 0.0); /* перемещение позиции */
      glEndList();
      glShadeModel(GL_FLAT);
}
static void drawLine(void)
{
      glBegin(GL_LINES);
      glVertex2f(0.0, 0.5);
      glVertex2f(15.0, 0.5);
      glEnd();
}
void display(void)
{
      GLuint i;
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(0.0, 1.0, 0.0); /* текущий цвет - зеленый */
      for (i = 0; i < 10; i++) /* рисуются 10 треугольников */
         glCallList(listName);
      drawLine(); /* эта линия зеленая? НЕТ! */

/* Где нарисована линия? */
      glFlush();
}
```

Команда `glTranslatef()` перемещает позицию рисования в соответствии с положением следующего объекта. Без этого треугольник рисовался бы поверх себя самого. Эти десять вызовов `glTranslatef()` также воздействуют на выполнение процедуры `drawLine()`, вызываемой в непосредственном режиме. Поэтому, если вы обращаетесь к командам преобразования изнутри списка отображения, не забывайте о вносимом ими на всю программу эффекте.

Одновременно может выполняться только один список отображения. Другими словами, вы должны, когда придет время, следом за `glNewList()` вызвать `glEndList()`, чтобы указать конец одного списка перед началом выполнения другого. Конечно, команда `glEndList()` без сопутствующей ей открывающей команды влечет ошибку `GL_INVALID_OPERATION`. (См. раздел «Обработка ошибок» главы 14 для получения более подробной информации об обработке ошибок.)

Присвоение имени списку отображения и создание списка

Каждый список отображения идентифицируется целочисленным индексом. При создании списка отображения вы должны быть внимательны, чтобы ненароком не указать уже используемый номер и, таким образом, не перезаписать существующий список отображения новым. Соответственно, во избежание путаницы

используйте команду `glGenLists()` для генерирования одного или более ранее не использованных индексов.

GLuint `glGenLists`(GLsizei *range*);

Выделяет диапазон предварительно высвобожденных смежных индексов (*range*). Возвращаемое целое число — это индекс, соответствующий началу блока смежных индексов. Все индексы в блоке при этом помечаются как используемые, и последующие вызовы `glGenLists()` не вернут эти значения, пока вы не удалите их собственноручно. Нуль возвращается, если требуемое количество индексов не может быть выделено или если аргумент *range* сам равен нулю.

В следующем примере запрашивается один новый индекс, проверяется результат запроса и, если все в порядке, создается список отображения:

```
listIndex = glGenLists(1);
if (listIndex != 0) {
  glNewList(listIndex,GL_COMPILE);
     ...
  glEndList();
}
```

ПРИМЕЧАНИЕ

Ноль не является корректным индексом.

void `glNewList`(GLuint *list*, GLenum *mode*);

Указывает начало списка отображения. Команды OpenGL, вызываемые в дальнейшем (пока не встретится `glEndList()`), помещаются в список, за исключением тех команд, которые по определению не могут быть в нем сохранены. (Эти команды обрабатываются немедленно, во время создания списка.) Аргумент *list* — это ненулевое положительное целое число, уникальным образом идентифицирующее список. Возможные значения для аргумента *mode*: `GL_COMPILE` и `GL_COMPILE_AND_EXECUTE`. Используйте `GL_COMPILE`, если хотите, чтобы команды OpenGL обрабатывались во время их размещения в списке; чтобы команды до помещения их в список выполнялись в непосредственном режиме, укажите `GL_COMPILE_AND_EXECUTE`.

void `glEndList`(void);

Помечает конец списка отображения.

При создании списка отображения запоминается текущий контекст OpenGL. Таким образом, когда контекст разрушается, список отображения также уничтожается. Некоторые оконные системы поддерживают множество контекстов, разделяемых между собой различными списками отображения. В этом случае список отображения разрушается, когда уничтожается последний контекст в *общей группе*.

Что сохраняется в списке отображения?

Когда вы формируете список отображения, в нем сохраняются только значения выражений. Если значения в массиве впоследствии изменяются, значения в спи-

ске изменяются также. В следующем фрагменте кода список отображения содержит команду, устанавливающую в качестве текущего цвета RGBA черный — (0.0, 0.0, 0.0).

Последующее изменение значения массива *color_vector* на красный цвет (1.0, 0.0, 0.0) не оказывает эффекта на список отображений, поскольку последний содержит значения, которые были актуальны на момент его создания:

```
GLfloat color_vector[3] = {0.0, 0.0, 0.0};
glNewList(1, GL_COMPILE);
   glColor3fv(color_vector);
glEndList();
color_vector[0] = 1.0;
```

Не все команды OpenGL могут быть помещены в список отображения и выполнены таким образом. Например, ими не могут быть команды, которые устанавливают состояние клиента, и команды, возвращающие значения параметров состояния. (Их в основном легко опознать — так как они возвращают значения параметров, передаваемых по ссылке, или возвращают значения напрямую.) Такие команды при включении их в список отображения выполняются немедленно.

В табл. 7.1 перечислены команды OpenGL, которые можно включить в список отображения. (Заметьте, что `glNewList()` генерирует ошибку, если она вызывается изнутри еще не завершенного списка.) Некоторые из этих команд нами еще не рассматривались, но вы можете найти их описание в тексте книги с помощью предметного указателя.

Таблица 7.1. Функции OpenGL, допустимые к помещению в списке отображения

glAreTexturesResident()	glFlush()	glNormalPointer()
glClientActiveTexture()	glFogCoordPointer()	glPixelStore()
glColorPointer()	glGenLists()	glPopClientAttrib()
glDeleteLists()	glGenTextures()	glPushClientAttrib()
glDeleteTextures()	glGet*()	glReadPixels()
glDisableClientState()	glIndexPointer()	glRenderMode()
glEdgeFlagPointer()	glInterleavedArrays()	glSecondaryColorPointer()
glEnableClientState()	glIsEnabled()	glSelectBuffer()
glFeedbackBuffer()	glIsList()	glTexCoordPointer()
glFinish()	glIsTexture()	glVertexPointer()

Для четкого понимания причины этих ограничений вспомните, что в сетевом окружении клиент OpenGL может находиться на одном компьютере, а сервер — на другом. Созданный список отображения постоянно хранится на сервере, таким образом, сервер не может рассчитывать на клиента в плане получения информации, связанной со списком отображения. Если запрашивающие команды, такие как `glGet*()` или `glIs*()`, разрешить к использованию в списках отображения, вызывающая программа будет зависеть от непредсказуемости времени возврата данных через сеть. Пока у нее не будет списка отображения в оригинальном виде,

в каком он послан, вызывающая программа не будет знать, где ей разместить данные. Таким образом, любые команды, возвращающие значения, не могут быть помещены в список отображения. Команды, изменяющие состояние клиента, такие как `glPixelStore()`, `glSelectBuffer()`, и команды, описывающие массивы вершин, также не могут быть сохранены в списке.

Например, подпрограммы спецификации вершин массива (такие как `glVertexPointer()`, `glColorPointer()` и `glInterleavedArrays()`), устанавливают указатели состояния клиента и тоже не могут быть сохранены в списке отображения. Команды `glArrayElement()`, `glDrawArrays()` и `glDrawElements()` посылают данные для построения примитивов из элементов, принадлежащих доступным массивам, соответственно, эти операции могут быть сохранены в списке отображения. (См. раздел «Массивы вершин» главы 2.) Данные массива вершин, сохраненные в списке отображения, извлекаются через разыменование указателей, но не непосредственно. Таким образом, дальнейшие изменения значений в вершинах не будут влиять на определение примитивов в списке отображения.

К влияющим командам относятся и любые команды, использующие режимы сохранения пикселов (pixel-storage modes). (См. раздел «Режимы хранения пикселов» главы 8.) Другие подпрограммы, полагающиеся на состояние клиента, такие как `glFlush()` и `glFinish()`, не могут быть сохранены в списке отображения, поскольку они зависят от состояния клиента в момент обработки списка.

Выполнение списка отображения

После того как вы создали список отображения, вы можете запустить его на выполнение вызовом `glCallList()`. Разумеется, один и тот же список отображения можно выполнять многократно, а также можно смешивать его обработку с командами непосредственного режима, как мы это уже видели.

> void `glCallList`(GLuint *list*);
>
> Выполняет список отображения, указанный аргументом *list*. Команды списка отображения выполняются в порядке своего сохранения, как если бы списка отбражения и не было. Если аргумент *list* не определен, ничего не происходит.

Вы можете вызывать `glCallList()` из любой точки программы, пока активен соответствующий ему контекст OpenGL (тот же, что был активен на момент создания списка, или контекст из общей группы). Список отображения может быть создан в одной подпрограмме, а выполнен — в другой, благодаря его уникальной идентификации по индексу. А вот для сохранения списка отображения в файле или, наоборот, для загрузки его из файла средств не имеется. Это лишнее подтверждение временного характера списков отображения.

Иерархические списки отбражения

Вы можете создать иерархический список отображения, в котором изнутри одного списка отображения — между `glNewList()` и `glEndList()` — выполняется вызов других списков той же командой `glCallList()`. Иерархически упорядочен-

ные списки полезны для объектов, составленных из готовых компонентов, особенно если последние используются больше одного раза. Например, следующий список отображения выводит велосипед, причем отдельные детали конструкции велосипеда (руль, рама и колеса) отображаются собственными списками:

```
glNewList(listIndex,GL_COMPILE);
   glCallList(handlebars);
   glCallList(frame);
   glTranslatef(1.0, 0.0, 0.0);
   glCallList(wheel);
   glTranslatef(3.0, 0.0, 0.0);
   glCallList(wheel);
glEndList();
```

Это не позволит вам реализовать полноценную глубокую рекурсию, так как вложенность списков отображения ограничена значением 64, но это число может быть и больше, в зависимости от реализации. Для получения информации о максимальной вложенности списков для конкретной реализации OpenGL воспользуйтесь вызовом

```
glGetIntegerv(GL_MAX_LIST_NESTING, GLint *data);
```

OpenGL позволяет создавать списки отображения, которые вызывают еще не существующие списки. При этом ничего не происходит.

Наконец, иерархическая структура — просто удобное средство компоновки нескольких списков более низкого уровня в смысле последующего редактирования, как в листинге 7.3. Здесь внешний список отображения объединяет описания отдельных вершин многоугольника.

Листинг 7.3. Иерархический список отображения

```
glNewList(1,GL_COMPILE);
   glVertex3fv(v1);
glEndList();
glNewList(2,GL_COMPILE);
   glVertex3fv(v2);
glEndList();
glNewList(3,GL_COMPILE);
   glVertex3fv(v3);
glEndList();
glNewList(4,GL_COMPILE);
   glBegin(GL_POLYGON);
      glCallList(1);
      glCallList(2);
      glCallList(3);
   glEnd();
glEndList();
```

Чтобы воспроизвести многоугольник, вызывается список с номером 4. Чтобы отредактировать отдельную вершину, нужно всего лишь пересоздать соответствующий этой вершине вложенный список. Удобно то, что, когда вы создадите новый список под старым индексом, существующий список автоматически будет уда-

лен. Это не значит, что распределение памяти будет оптимальным для достижения максимальной производительности, но в иных случаях это лучше чем ничего.

Управление индексацией списков отображений

Пока что мы советовали не пренебрегать командой `glGenLists()`, чтобы всегда быть уверенными в уникальности индексов списков отображения. Если это по каким-либо причинам вас не устраивает, воспользуйтесь командой `glIsList()`, позволяющей узнать, свободен ли указанный индекс.

> GLboolean `glIsList`(GLuint *list*);
> Возвращает `GL_TRUE`, если аргумент *list* представляет уже используемый индекс, и `GL_FALSE` в противном случае.

Вы можете явно удалить указанный список отображения или набор списков с последовательными индексами при помощи вызова `glDeleteLists()`. Соответствующие индексы становятся снова доступными.

> void `glDeleteLists`(GLuint *list*, GLsizei *range*);
> Удаляет заданное количество (*range*) списков отображения, начиная от заданного индексом *list*. При попытке удаления несуществующего списка ничего не происходит.

Выполнение составных списков отображения

OpenGL предоставляет эффективный механизм последовательного выполнения нескольких списков отображения. Этот механизм требует, чтобы вы поместили индексы списков в массив и воспользовались вызовом `glCallLists()`. Очевидно, что применение данного механизма имеет смысл в случае, когда индексы соответствуют каким-либо выразительным значениям. Так, если вы создаете шрифт, каждый индекс будет ассоциирован с ASCII-кодом символа шрифта. Чтобы иметь несколько таких шрифтов, вам нужно задать для них различные начальные индексы, а сделать это легко с помощью вызова `glListBase()` перед выполнением `glCallLists()`.

> void `glListBase`(GLuint *base*);
> Задает смещение, добавляемое к индексам списка отображения. По умолчанию принято значение 0. Это смещение не имеет эффекта ни на команду `glCallList()`, примененную к одиночному списку отображения, ни на команду `glNewList()`.
> void `glCallLists`(GLsizei *n*, GLenum *type*, const GLvoid **lists*);
> Выполняет *n* списков отбражения. Целевые индексы вычисляются путем сложения базового смещения (заданного `glListBase()`) с целыми числами массива, указанного аргументом *lists*.

Аргумент *type* обозначает тип данных элемента массива: `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT` или `GL_FLOAT`.

Соответственно, понимается, что с аргументом *lists* нужно обращаться как с массивом байтов, беззнаковых байтов, коротких целых чисел, коротких целых без знака, целых чисел, беззнаковых целых чисел или чисел с плавающей запятой. Аргумент *type* может также принимать значение `GL_2_BYTES`, `GL_3_BYTES` или `GL_4_BYTES`. В этом случае цепочки длиной в 2, 3 или 4 байта читаются из массива *lists*, а результирующее смещение вычисляется по следующему алгоритму (здесь *byte*[*0*] — стартовый байт последовательности):

```
/* b = 2, 3 или 4; байты массива нумеруются в последовательности 0, 1, 2, 3 */
offset = 0;
for (i = 0; i < b; i++) {
   offset = offset << 8;
   offset += byte[i];
}
index = offset + listbase;
```

Для многобайтовых полей первыми обрабатываются старшие разряды, а в остальном порядок идентичен обработке массива.

Листинг 7.4 приведен нами в качестве примера использования комбинированных списков отображения, это фрагмент программы, полностью представленной в листинге 7.5. Эта программа рисует символы векторного шрифта (набора символов, образованных отрезками линий). Процедура `initStrokedFont()` устанавливает индексы списка отображения для каждого символа так, чтобы они соответствовали своим ASCII-кодам.

Листинг 7.4. Определение составных списков отображений

```
void initStrokedFont(void)
{
   GLuint base;
   base = glGenLists(128);
   glListBase(base);
   glNewList(base+'A', GL_COMPILE);
      drawLetter(Adata); glEndList();
   glNewList(base+'E', GL_COMPILE);
      drawLetter(Edata); glEndList();
   glNewList(base+'P', GL_COMPILE);
      drawLetter(Pdata); glEndList();
   glNewList(base+'R', GL_COMPILE);
      drawLetter(Rdata); glEndList();
   glNewList(base+'S', GL_COMPILE);
      drawLetter(Sdata); glEndList();
   glNewList(base+' ', GL_COMPILE); /* пробел */
      glTranslatef(8.0, 0.0, 0.0);
   glEndList();
}
```

Команда `glGenLists()` заказывает 128 последовательных индексов. Первый из них становится базовым значением. В списке отображения каждый символ представляется суммой базового значения и ASCII-кода символа. В этом примере мы описываем только пять букв и пробел.

После создания списков отображения можно вызвать команду `glCallLists()` для их обработки. Так, можно передать процедуре `printStrokedString()` строку символов:

```
void printStrokedString(GLbyte *s)
{
  GLint len = strlen(s);
  glCallLists(len, GL_BYTE, s);
}
```

Здесь значение ASCII-кода служит как смещение в наборе индексов, а целевой индекс образуется его сложением с текущим базовым смещением. Результат работы программы, представленной в листинге 7.5, показан на рис. 7.1.

Рис. 7.1. Фраза, составленная из символов векторного шрифта: A, E, P, R, S

Листинг 7.5. Составные списки отображения, определяющие векторный шрифт: stroke.c

```
#define PT 1
#define STROKE 2
#define END 3

typedef struct charpoint {
  GLfloat x, y;
  int type;
} CP;

CP Adata[] = {
  { 0, 0, PT}, {0, 9, PT}, {1, 10, PT}, {4, 10, PT},
  {5, 9, PT}, {5, 0, STROKE}, {0, 5, PT}, {5, 5, END}
};

CP Edata[] = {
  {5, 0, PT}, {0, 0, PT}, {0, 10, PT}, {5, 10, STROKE},
  {0, 5, PT}, {4, 5, END}
};

CP Pdata[] = {
  {0, 0, PT}, {0, 10, PT}, {4, 10, PT}, {5, 9, PT}, {5, 6, PT},
  {4, 5, PT}, {0, 5, END}
};

CP Rdata[] = {
  {0, 0, PT}, {0, 10, PT}, {4, 10, PT}, {5, 9, PT}, {5, 6, PT},
  {4, 5, PT}, {0, 5, STROKE}, {3, 5, PT}, {5, 0, END}
};

CP Sdata[] = {
  {0, 1, PT}, {1, 0, PT}, {4, 0, PT}, {5, 1, PT}, {5, 4, PT},
```

продолжение ⇩

Листинг 7.5 (*продолжение*)

```
   {4, 5, PT}, {1, 5, PT}, {0, 6, PT}, {0, 9, PT}, {1, 10, PT},
   {4, 10, PT}, {5, 9, END}
};

/* drawLetter() интерпретирует команды из массива
* для символа и воспроизводит символ отрезками прямых
*/
static void drawLetter(CP *l)
{
   glBegin(GL_LINE_STRIP);
   while (1) {
      switch (l->type) {
         case PT:
            glVertex2fv(&l->x);
            break;
         case STROKE:
            glVertex2fv(&l->x);
            glEnd();
            glBegin(GL_LINE_STRIP);
            break;
         case END:
            glVertex2fv(&l->x);
            glEnd();
            glTranslatef(8.0, 0.0, 0.0);
            return;
      }
      l++;
   }
}

/* Создание списка отображения для каждого из 6 символов */
static void init(void)
{
   GLuint base;
   glShadeModel(GL_FLAT);
   base = glGenLists(128);
   glListBase(base);
   glNewList(base+'A', GL_COMPILE); drawLetter(Adata);
   glEndList();
   glNewList(base+'E', GL_COMPILE); drawLetter(Edata);
   glEndList();
   glNewList(base+'P', GL_COMPILE); drawLetter(Pdata);
   glEndList();
   glNewList(base+'R', GL_COMPILE); drawLetter(Rdata);
   glEndList();
   glNewList(base+'S', GL_COMPILE); drawLetter(Sdata);
   glEndList();
   glNewList(base+' ', GL_COMPILE);
   glTranslatef(8.0, 0.0, 0.0); glEndList();
}
char *test1 = "A SPARE SERAPE APPEARS AS";
char *test2 = "APES PREPARE RARE PEPPERS";
```

```
static void printStrokedString(char *s)
{
  GLsizei len = strlen(s);
  glCallLists(len, GL_BYTE, (GLbyte *)s);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 1.0, 1.0);
  glPushMatrix();
  glScalef(2.0, 2.0, 2.0);
  glTranslatef(10.0, 30.0, 0.0);
  printStrokedString(test1);
  glPopMatrix();
  glPushMatrix();
  glScalef(2.0, 2.0, 2.0);
  glTranslatef(10.0, 13.0, 0.0);
  printStrokedString(test2);
  glPopMatrix();
  glFlush();
}

void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
}

void keyboard(unsigned char key, int x, int y)
{
  switch (key) {
     case ' ':
        glutPostRedisplay();
        break;
     case 27:
        exit(0);
        break;
  }
}

int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(440, 120);
  glutCreateWindow(argv[0]);
  init();
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutDisplayFunc(display);
```

продолжение ⇙

Листинг 7.5 (*продолжение*)

```
    glutMainLoop();
    return 0;
}
```

Управление параметрами состояния с помощью списков отображения

Список отображения может содержать вызовы команд, которые изменяют переменные состояния OpenGL. Эти изменения производятся во время выполнения списка отображения, как если бы команды выполнялись в непосредственном режиме, и их результаты остаются зафиксированными по окончании выполнения списка отображения. Вы уже видели это в листинге 7.2, и мы снова демонстрируем в листинге 7.6, что изменения текущих цвета и матрицы остаются в силе и после выполнения списка отображения.

Листинг 7.6. Сохранение состояния после выполнения списка отображения

```
glNewList(listIndex,GL_COMPILE);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_POLYGON);
        glVertex2f(0.0, 0.0);
        glVertex2f(1.0, 0.0);
        glVertex2f(0.0, 1.0);
    glEnd();
    glTranslatef(1.5, 0.0, 0.0);
glEndList();
```

Если теперь вы вызовете следующую последовательность команд, линия будет нарисована красным (текущим) цветом и дополнительно сместится на значение (1.5, 0.0, 0.0):

```
glCallList(listIndex);
glBegin(GL_LINES);
    glVertex2f(2.0,-1.0);
    glVertex2f(1.0, 0.0);
glEnd();
```

С переменными состояния обращаются по-разному. Можно либо оставить изменения в силе, либо, наоборот, предварительно перед обработкой списка зафиксировать состояние и восстановить его по окончании выполнения списка. Это ваше право, но помните, что команда `glGet*()` в списке отображения недоступна и, таким образом внутри него нет возможности запросить текущие переменные состояния.

В то же время в вашем распоряжении имеется команда `glPushAttrib()`, позволяющая сохранить разом несколько переменных состояния, и команда `glPopAttrib()` — для восстановления значений. Чтобы сохранить и восстановить текущую матрицу, используйте команды `glPushMatrix()` и `glPopMatrix()` согласно рекомендациям раздела «Операции со стеком матрицы» главы 3. Эти

команды могут легально кэшироваться в списке отображения. Чтобы восстановить значения переменных состояния из листинга 7.6, мы могли бы воспользоваться кодом из листинга 7.7.

Листинг 7.7. Восстановление значений переменных состояния изнутри списка отображения

```
glNewList(listIndex,GL_COMPILE);
   glPushMatrix();
   glPushAttrib(GL_CURRENT_BIT);
   glColor3f(1.0, 0.0, 0.0);
   glBegin(GL_POLYGON);
      glVertex2f(0.0, 0.0);
      glVertex2f(1.0, 0.0);
      glVertex2f(0.0, 1.0);
   glEnd();
   glTranslatef(1.5, 0.0, 0.0);
   glPopAttrib();
   glPopMatrix();
glEndList();
```

Если вы прибегнете к варианту листинга 7.7, с помощью кода листинга 7.8 вы сможете нарисовать линию зеленого цвета, в прежней позиции. Если вы воспользуетесь списком отображения из листинга 7.6, то есть не сохраните (и не восстановите) состояние, линия будет красной, а ее позиция каждый из десяти раз будет смещаться на значение (1.5, 0.0, 0.0).

Листинг 7.8. Пример того, как список отображения может или не может воздействовать на drawLine()

```
void display(void)
{
   GLint i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 1.0, 0.0); /* установить текущим цветом зеленый */
   for (i = 0; i < 10; i++)
      glCallList(listIndex); /* список отображения выполняется 10 раз */
   drawLine(); /* где появляется линия и каким цветом она окрашена? */
   glFlush();
}
```

Инкапсуляция изменений режима

Вы можете использовать списки отображения для организации и сохранения групп команд, изменяющих атрибуты режимов, или для варьирования параметров. При переключении с одной группы настроек на другую использование списков отображения бывает более эффективно, чем непосредственные вызовы, благодаря кэшированию настроек в формате, наиболее подходящем вашей графической системе.

Это случаи варьирования освещения, модели распространения света и свойств материала. Вы можете также использовать списки отображения в качестве шаблонов штриховки, параметров тумана и для обсчета секущих плоскостей. В этих

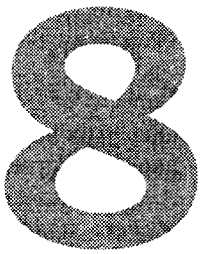
случаях вы увидите, что списки отображения близки по скорости выполнения к эквивалентным непосредственным вызовам, но помните, что спискам отображений присущи дополнительные издержки на само их поддержание при выполнении.

Листинг 7.9 показывает, как использовать списки отбражения для переключения между тремя различными шаблонами штриховки. Сначала вызывается `glGenLists()` для выделения индексов для каждого образца, и затем для всех трех создаются списки отображения. В результате этой подготовки можно использовать команду `glCallList()` для переключения с одного шаблона на другой.

Листинг 7.9. Списки отображения для изменения режима

```
GLuint offset;
offset = glGenLists(3);

glNewList(offset, GL_COMPILE);
   glDisable(GL_LINE_STIPPLE);
glEndList();

glNewList(offset+1, GL_COMPILE);
   glEnable(GL_LINE_STIPPLE);
   glLineStipple(1, 0x0F0F);
glEndList();

glNewList(offset+2, GL_COMPILE);
   glEnable(GL_LINE_STIPPLE);
   glLineStipple(1, 0x1111);
glEndList();

#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
   glVertex2f((x1),(y1)); glVertex2f((x2),(y2)); glEnd();

glCallList(offset);
drawOneLine(50.0, 125.0, 350.0, 125.0);

glCallList(offset+1);
drawOneLine(50.0, 100.0, 350.0, 100.0);

glCallList(offset+2);
drawOneLine(50.0, 75.0, 350.0, 75.0);
```

Вывод пикселов, битовых образов, шрифтов и изображений



После прочтения этой главы вы сможете:

- Позиционировать и отображать данные в виде битовых образов
- Считывать пикселы (битовых и иных изображений) из буфера кадров в оперативную память и из памяти обратно в буфер
- Копировать пикселы из одного буфера цвета в другой или в другую позицию того же буфера
- Увеличивать или уменьшать изображения при их записи в буфер кадра
- Управлять форматированием пикселов и выполнять другие преобразования данных во время их передачи в буфер и обратно
- Выполнять обработку пикселов с помощью расширения Imaging Subset

До этого времени в руководстве основное внимание уделялось воспроизведению векторных данных — точек, линий и многоугольников. Теперь мы рассмотрим визуализацию данных двух других важных классов OpenGL:

- битовых (двоичных) образов, которыми обычно представляются символы шрифта;
- изображений, как просто отсканированных, так и сгенерированных программно.

Оба этих типа являются прямоугольниками — двумерными массивами пикселов. Единственная разница между ними состоит в том, что для *битового образа* (bitmap) каждый пиксел представлен одиночным битом, а *изображения* (image) обычно подразумевают соответствие каждому пикселу нескольких элементов данных (например, информацию о компонентах: красном, зеленом и синем). То есть битовые образы подобны маскам, накладываемым на другие изображения, а изображения перезаписываются на месте других данных в буфере кадров или смешиваются с ними.

В этой главе рассказывается, как передать на отрисовку пикселы из оперативной памяти в буфер кадров и как считать их в обратном направлении. Здесь также описывается, как копировать пикселы из одной позиции в другую, между буферами или в пределах одного буфера.

ПРИМЕЧАНИЕ

OpenGL не поддерживает чтение пикселов в файл и чтение их из файла.

Эта глава состоит из следующих основных разделов:

- **Битовые образы и шрифты.** Здесь описываются команды для позиционирования и рисования данных, представленных битовыми образами. Битовые образы обычно используются для представления шрифтов.
- **Изображения.** Предоставляет базовую информацию о рисовании, чтении и копировании пикселов.
- **Конвейерная обработка изображений.** Описывает операции, производимые над изображениями и битовыми образами во время чтения их из буфера кадров и записи их в буфер.
- **Чтение и рисование прямоугольных областей.** Здесь вы найдете подробные сведения о том, как пикселы хранятся в оперативной памяти и что с ними происходит при извлечении их из памяти и сохранении в ней.
- **Советы по увеличению быстродействия вывода пикселов.** Список советов, направленных на лучшую производительность операций рисования пикселов.
- **Расширение Imaging Subset.** Информация о расширении OpenGL для дополнительной обработки пикселов.

В большинстве случаев действия над пикселами сводятся к простейшим операциям, поэтому для понимания того, как создать соответствующее приложение, вам достаточно ознакомиться с первыми тремя разделами этой главы. Но манипулирование пикселами может быть и более сложным делом — пикселы можно

по-разному хранить в памяти и выполнять их передачу в буфер и обратно также разными способами. Этим тонкостям посвящен четвертый раздел главы. Скорее всего, информация этого раздела вам понадобится не ранее того момента, когда вы перейдете к практическим действиям. Здесь вам окажутся полезными рекомендации раздела «Советы по увеличению быстродействия вывода пикселов».

В реализации OpenGL версии 1.2 появилась упаковка данных (таких как `GL_UNSIGNED_BYTE_3_3_2` и `GL_UNSIGNED_INT_10_10_10_2`) в форматы BGR и BGRA. В этой же версии появился набор операций над изображениями, включающий в себя преобразование цветовых матриц, подстановку цвета по таблице, гистограммы и дополнительные операции смешивания (`glBlendEquation()`, `glBlendColor()` и еще несколько констант режимов). Это расширение получило название *Imaging Subset*. В версии 1.4 эти новые операции смешивания были переведены на уровень ядра OpenGL.

Версия 1.4 также позволила использовать функции смешивания источника `GL_SRC_COLOR` и `GL_ONE_MINUS_SRC_COLOR` так же, как `GL_DST_COLOR` и `GL_ONE_MINUS_DST_COLOR` — целевые функции смешивания. Еще в ней появилась функция `glWindowPos*()` для указания растровой позиции в системе оконных координат.

Битовые образы и шрифты

Битовый образ (bitmap) — это прямоугольный массив, состоящий из нулей и единиц, который служит маской для прямоугольной области окна. Допустим, что вы рисуете битовый образ и при этом текущий растровый цвет — красный. Тогда во всех позициях массива, в которых находятся единицы, соответствующие пикселы будут замещены на красный (или скомбинированы с красными пикселами, что зависит от характера операции). (См. раздел «Тесты и операции над фрагментами» главы 10.) Там же, где окажутся нули, пикселы остаются без каких либо изменений. Чаще всего битовые образы находят применение для отображения символов на экране.

OpenGL обеспечивает только самый низкий уровень поддержки отображения строк символов и управления шрифтами. Команды `glRasterPos*()` (или альтернативная ей `glWindowPos*()`) и `glBitmap()` позиционируют и отображают на экране только один битовый образ. В дополнение к этому имеется механизм списков отображения, позволяющий обработать последовательность из кодов символов, связанных по индексу с соответствующими битовыми образами. (Подробную информацию о списках отображения см. в главе 7.) При необходимости производить более сложные действия над битовыми образами, шрифтами и строками символов вам потребуется подготовить собственные процедуры.

В листинге 8.1 приведен пример программы, три раза отображающей на экране символ F. Рисунок 8.1 демонстрирует этот символ как битовый образ, вместе с его двоичными (нулями и единицами) данными.

Обратите внимание, что на рис. 8.1 максимальная ширина видимой части символа F составляет 10 бит. Хотя горизонтальный размер массива битов всегда

кратен 8, фактическая ширина битового образа не обязательно должна быть кратной 8. Отсчет и отображение битов ведутся от левого нижнего угла: первым отображается нижний ряд, затем — расположенный над ним и т. д. Как можно увидеть из кода, битовый образ хранится в памяти в этом порядке — сначала следуют числа 0xC0, 0x00, 0xC0, 0x00, соответствующие двум нижним строкам, и заканчивается он значениями 0xFF, 0xC0, 0xFF, 0xC0, то есть верхними двумя рядами.

Листинг 8.1. Рисование битового образа символа: drawf.c

```
GLubyte rasters[24] = {
   0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
   0xff, 0x00, 0xff, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
   0xff, 0xc0, 0xff, 0xc0};

void init(void)
{
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glClearColor(0.0, 0.0, 0.0, 0.0);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glRasterPos2i(20, 20);
   glBitmap(10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
   glBitmap(10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
   glBitmap(10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
   glFlush();
}
```

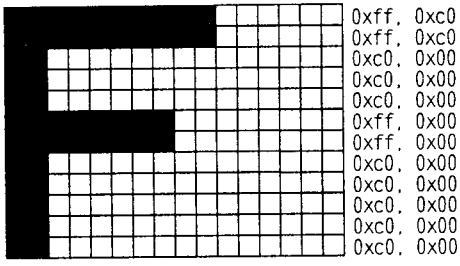


Рис. 8.1. Битовый образ буквы F и кодировка его данных

Здесь для нас также интересны команды `glRasterPos2i()` и `glBitmap()`; они обсуждаются в подробностях в следующем разделе. Что касается функции `glPixelStorei()` — пока не обращайте на нее внимания; она относится к способу хранения данных битового образа в памяти компьютера (см. раздел «Режимы хранения пикселов» для получения подробной информации).

Текущая позиция растра

Текущая позиция растра — это координаты того места экрана, куда будет выведен следующий битовый образ (или изображение). В примере с буквой F растровая

позиция устанавливается вызовом `glRasterPos*()`, присвоением значений (20, 20), с этого места начинается вывод нижнего ряда битового образа:

```
glRasterPos2i(20, 20);
```

void `glRasterPos`{234}{sifd}(*TYPE x, TYPE y, TYPE z, TYPE w*);
void `glRasterPos`{234}{sifd}v(*TYPE *coords*);
Устанавливает текущую позицию растра. Аргументы *x*, *y*, *z* и *w* указывают координаты растрового объекта. Если используется векторная форма функции, аргумент *coords* определяет все необходимые координаты. В варианте `glRasterPos2*()` аргумент *z* неявно приравнивается нулю, и значение аргумента *w* также неявно задается равным 1; аналогично, для команды `glRasterPos3*()` значение $w - 1$.

Координаты позиции растра преобразуются в координаты экрана так же, как координаты, принимаемые функцией `glVertex*()` (то есть с использованием модельно-видовых матриц (modelview) и перспективы). После преобразования эти координаты проверяются на принадлежность окну просмотра (viewport) и отсекаются, если выходят за границы видимой области. В последнем случае текущая позиция растра считается некорректной (invalid).

До версии 1.4, если вы хотели указать растровую позицию в оконных координатах, это требовало указания макетной (modelview) и проекционной матриц для обычной отрисовки в стиле 2D, примерно так, как в следующей последовательности команд, где *width* и *height* — размеры области просмотра в пикселах:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, (GLfloat) width, 0.0, (GLfloat) height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

В версии 1.4 в ваше распоряжение предоставлена команда `glWindowPos*()` как альтернатива для `glRasterPos*()`. Команда `glWindowPos*()` принимает оконные координаты позиции растра, не требуя их матричного преобразования, но не отсекая их по размерам области просмотра. Функция `glWindowPos*()` облегчает смешивание 2D- и 3D-графики, выполняя за вас скучную задачу переключения между состояниями.

void `glWindowPos`{23}{sifd}(*TYPE x, TYPE y, TYPE z*);
void `glWindowPos`{23}{sifd}v(*TYPE *coords*);
Устанавливает текущую позицию растра, принимая аргументы *x* и *y*, указанные в оконных координатах, без выполнения матричных преобразований, отсечения, подсветки или генерирования координат текстуры. Значение *z* преобразуется согласно текущим значениям (ими ограничивается — clamped), задаваемым функцией `glDepthRange()`. Если используется векторная форма команды, все координаты помещаются в массив *coords*. В варианте `glWindowPos2*()` аргумент *z* неявно равен 0.

Для получения текущих координат растра (ранее установленных `glRasterPos*()` или `glWindowPos*()`) вы можете использовать запрос `glGetFloatv()` с первым

аргументом, равным GL_CURRENT_RASTER_POSITION. Второй аргумент должен быть указателем на массив значений (x, y, z, w) в формате с плавающей запятой. Из результата вызова glGetBooleanv() с первым аргументом GL_CURRENT_RASTER_POSITION_VALID можно узнать, действительна (valid) ли текущая позиция растра.

Вывод битового образа

Установив необходимую позицию растра, вы можете использовать команду glBitmap() для вывода данных.

void glBitmap(GLsizei *width*, GLsizei *height*, GLfloat x_{bo}, GLfloat y_{bo}, GLfloat x_{bi}, GLfloat y_{bi}, const GLubyte **bitmap*);

Рисует битовый образ, указанный аргументом *bitmap*, значение которого представляет собой указатель на данные изображения. Битовый образ располагается в соответствии с текущей позицией растра. Если она недействительна, ничего не происходит. Аргументы *width* и *height* задают ширину и высоту битового образа в пикселах. Ширина не обязана быть кратной 8, хотя при сохранении данные будут выровнены по значению 8. (В примере с буквой F избыток «мусорных» битов правее левых 10 не принципиален: все равно glBitmap() отрисует столько битов, сколько нужно, в нашем случае — 10 у каждого ряда.) Используйте аргументы x_{bo} и y_{bo} для описания начальных координат битового образа относительно текущей позиции растра (положительные значения поднимают рисунок вверх и смещают вправо; отрицательные — сдвигают вниз и влево); аргументы x_{bi} и y_{bi} отвечают за приращения текущей позиции растра — ее координаты x и y изменятся на эти значения после завершения операции (рис. 8.2).

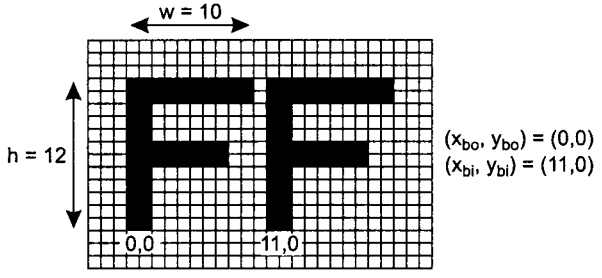


Рис. 8.2. Битовый образ и связанные с ним параметры

Такое разграничение «обязанностей» удобно тем, что позволяет размещать произвольным образом символы, опущенные ниже базовой линии (обычно с подстрочными элементами, такие как «g», «j» и «y»), буквы с орнаментом или перекошенные влево символы (в шрифтах с левым наклоном).

По окончании рисования образа текущая позиция растра смещается на значения x_{bi} и y_{bi}. (Если вашей целью является лишь перемещение текущей позиции, лучше вызвать glBitmap() с аргументом *bitmap*, установленным в NULL, и нулевыми значениями ширины и высоты (*width* и *height*).) Для стандартной латиницы y_{bi} обычно задается как 0.0, а x_{bi} положительно (в соответствии с направлением

письма слева направо). Для иврита (шрифт Hebrew), где направление письма противоположное, значение x_{bi}, соответственно, отрицательное. Для шрифтов, предполагающих вывод знаков в столбец вертикально, x_{bi} равно 0, а y_{bi} — ненулевое. На рис. 8.2 для каждого экземпляра буквы F текущая позиция передвигается вправо на 11 пикселов, тем самым образуется межсимвольный интервал шириной в пиксел.

Поскольку x_{bo}, y_{bo}, x_{bi} и y_{bi} — числа вещественные, то и размер символов не ограничивается только целыми значениями. Хотя при отображении символов от расположения их по пикселам на растровых экранах никуда не уйти, все равно текущая позиция вычисляется в формате с плавающей запятой, с целью избежать накопления погрешности при выводе последующих символов. Например, если код примера с буквой F модифицировать таким образом, чтобы x_{bi} стало равным 11.5 вместо 12, и продолжить рисование дальше, интервал между буквами будет либо 1, либо 2 пиксела, что максимально приближено к требуемой дистанции в 1.5 пиксела.

ПРИМЕЧАНИЕ

Предлагаемые средства не позволяют поворачивать символы растровых шрифтов, поскольку предполагают выравнивание битового образа относительно осей x и y буфера кадров. Также не поддерживается масштабирование битовых образов.

Выбор цвета для битового образа

Вы уже знакомы с командами `glColor*()` и `glIndex*()` для установки текущего цвета или его индекса на примере геометрических примитивов. Те же самые команды используются для задания значений переменных `GL_CURRENT_RASTER_COLOR` и `GL_CURRENT_RASTER_INDEX`, отображающих битовые образы. Параметры состояния цвета растра определяются значением текущего цвета с помощью вызова `glRasterPos*()`, и для нас это — подвох. Ответим на вопрос, каким окажется цвет битового образа в следующем фрагменте кода?

```
glColor3f(1.0, 1.0, 1.0); /* белый */
glRasterPos3fv(position);
glColor3f(1.0, 0.0, 0.0); /* красный */
glBitmap(....);
```

Это может быть неожиданным для вас, но он — белый. Это значение было присвоено параметру состояния `GL_CURRENT_RASTER_COLOR` при вызове `glRasterPos3fv()`. Второй вызов — `glColor3f()` — изменил значение `GL_CURRENT_COLOR` в расчете на обработку геометрических примитивов, но вовсе не цвет битового образа.

Узнать значение текущего цвета растра или его индекса можно, воспользовавшись командой `glGetFloatv()` или `glGetIntegerv()` с указанием `GL_CURRENT_RASTER_COLOR` или `GL_CURRENT_RASTER_INDEX` в качестве первого аргумента.

Шрифты и списки отображения

В общем и целом списки отображения были рассмотрены в главе 7. Тем не менее есть несколько команд управления списками отображения, более уместных при

обработке строк символов. При чтении этого раздела имейте в виду, что все представленные здесь идеи в той же мере применимы не только к битовым образам, но и к геометрическим примитивам — точкам, линиям и многоугольникам. (См. раздел «Выполнение составных списков отображения» главы 7, как пример геометрически описываемого шрифта.)

Шрифт обычно состоит из набора символов, идентифицируемых номерами (как правило, ASCII-кодом). Для стандартной ASCII-таблицы заглавная буква A имеет номер 65, B — 66 и т. д. Строка «DAB» представляется тремя индексами — 68, 65, 66. В простейшем приближении список отображения с номером 65 рисует букву A, с номером 66 — букву B и т. д. Чтобы вывести строку с кодами 68, 65, 66, нужно просто выполнить соответствующие списки отображения.

Вы можете использовать команду `glCallLists()` несколькими способами:

```
void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);
```

Здесь первый аргумент, *n*, указывает количество символов для рисования, *type* — обычно `GL_BYTE`, а *lists* — массив кодов символов.

Поскольку многие приложения требуют отображения строк символов различными шрифтами и с разными размерами, этот самый простой подход неприемлем. Вы бы наверняка хотели выводить букву A по ее коду вне зависимости от активного на данный момент шрифта. Нельзя ассоциировать буквы A, B и C одного шрифта с кодами 1065, 1066, 1067, а другого шрифта — с кодами 2065, 2066, 2067, так как индекс ограничен числом 256 по причине 8-разрядности байта. Лучшее решение — добавить смещение к каждому элементу строки перед выбором списка отображения. В этом случае можно было бы воспользоваться предыдущими цифрами. Так, чтобы вывести символы шрифта 1, нужно задать смещение 1000 и обратиться к спискам с номерами 65, 66 и 67. Чтобы отобразить ту же строку шрифтом 2, можно задать смещение 2000 и воспользоваться теми же списками.

Чтобы установить смещение, используйте команду `glListBase()`. Для предшествующего примера она должна быть вызвана со значением 1000 или 2000 в качестве аргумента (единственного). Теперь вы нуждаетесь только в списке последовательных неиспользуемых индексов, который предоставляет вызов `glGenLists()`:

```
GLuint glGenLists(GLsizei range);
```

Эта функция возвращает блок из *range* идентификаторов. Они помечаются как «используемые», даже если списки пока пустые, с той целью, чтобы дальнейшие вызовы `glGenLists()` не возвращали больше эти номера (пока вы явно не отмените это действие). Поэтому, если вы используете 4 как аргумент и если `glGenLists()` вернула 81, вы можете присвоить своим спискам отображения номера 81, 82, 83 и 84. Если `glGenLists()` не находит свободного диапазона требуемой длины, она возвращает 0. (Заметьте, что команда `glDeleteLists()` позволяет быстро удалить все списки, связанные со шрифтом, одним действием.)

В большинстве американских и европейских шрифтов количество символов невелико (меньше 256), поэтому такие символы представимы однобайтовыми индексами. В отличие от них, азиатские шрифты требуют объемных наборов символов, и для них кодирование «байт-на-символ» не представляется возможным.

OpenGL поддерживает строки, составленные из 1-, 2-, 3- или 4-байтовых символов, посредством задания аргумента *type* функции `glCallLists()`. Этот параметр может принимать следующие значения:

`GL_BYTE`	`GL_UNSIGNED_BYTE`
`GL_SHORT`	`GL_UNSIGNED_SHORT`
`GL_INT`	`GL_UNSIGNED_INT`
`GL_FLOAT`	`GL_2_BYTES`
`GL_3_BYTES`	`GL_4_BYTES`

(См. раздел «Выполнение составных списков отображения» главы 7 для получения более подробной информации.)

Описание и использование полного шрифта

Команда `glBitmap()` в сочетании с механизмом списков отображения, о котором говорилось в предыдущем разделе, облегчает задачу описания растровых шрифтов. В листинге 8.2 представлены символы шрифта в кодировке ASCII и в верхнем регистре. В этом примере все символы имеют одинаковую ширину, но так бывает не всегда. После определения символов программа печатает сообщение «THE QUICK BROWN FOX JUMPS OVER A LAZY DOG.»

Код из листинга 8.2 напоминает пример с буквой F, за исключением того, что каждый битовый образ хранится в своем собственном списке. Когда вы добавляете смещение, возвращенное функцией `glGenLists()`, идентификатор списка отображения равен ASCII-коду символа.

Листинг 8.2. Отображение символов полного шрифта: font.c

```
GLubyte space[] =
   {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
GLubyte letters[][13] = {
   {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c, 0x18},
   {0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
   {0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
   {0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0xce, 0xfc},
   {0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
   {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xff},
   {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
   {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
   {0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x7e},
   {0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06},
   {0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6, 0xc3},
   {0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
   {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xdb, 0xff, 0xff, 0xe7, 0xc3},
   {0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3, 0xf3, 0xe3, 0xe3},
   {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xe7, 0x7e},
   {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
   {0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0x66, 0x3c},
   {0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
```

продолжение ⇩

Листинг 8.2 (*продолжение*)

```
   {0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0, 0xc0, 0xe7, 0x7e},
   {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0xff},
   {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
   {0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
   {0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
   {0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
   {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
   {0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06, 0x03, 0x03, 0xff}
};
GLuint fontOffset;
void makeRasterFont(void)
{
   GLuint i, j;
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   fontOffset = glGenLists(128);
   for (i = 0,j = 'A'; i < 26; i++,j++) {
      glNewList(fontOffset + j, GL_COMPILE);
      glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, letters[i]);
      glEndList();
   }
   glNewList(fontOffset + ' ', GL_COMPILE);
   glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, space);
   glEndList();
}
void init(void)
{
   glShadeModel(GL_FLAT);
   makeRasterFont();
}
void printString(char *s)
{
   glPushAttrib(GL_LIST_BIT);
   glListBase(fontOffset);
   glCallLists(strlen(s), GL_UNSIGNED_BYTE, (GLubyte *) s);
   glPopAttrib();
}
/* Все, что находится выше этой строки, могло бы размещаться в библиотеке,
 * описывающей шрифт. Чтобы это сделать, вы можете вызвать
 * команду makeRasterFont() перед тем, как обратиться к printString()
 */
void display(void)
{
   GLfloat white[3] = { 1.0, 1.0, 1.0 };

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3fv(white);

   glRasterPos2i(20, 60);
   printString("THE QUICK BROWN FOX JUMPS");
   glRasterPos2i(20, 40);
   printString("OVER A LAZY DOG");
   glFlush();
}
```

Изображения

Изображение — это почти что битовый образ, с той разницей, что вместо одного бита на пиксел прямоугольной области экрана изображение может содержать больше информации. Например, цвет пиксела (компоненты R, G, B, A). Изображения обычно получают из разных источников, в том числе:

- из фотографии, оцифрованной сканером;
- изображения, первоначально сгенерированного на экране графической программой с использованием специального оборудования и затем считанного обратно попиксельно;
- программы, попиксельно непосредственно сгенерировавшей изображение в памяти.

Обычно считают, что источником картинок являются буферы цветов. Тем не менее вы можете читать и записывать прямоугольные области и из буфера глубины (depth buffer), и из буфера трафарета (stencil buffer). (Информацию о других типах буферов см. в главе 10.)

В дополнение к простому отображению на экране изображения могут быть использованы для растрового отображения текстур, где они, по сути, вставляются в многоугольники, которые воспроизводятся на экране обычным образом. (См. главу 9 для получения подробной информации об этой технике.)

Чтение, запись и копирование пикселов

OpenGL предоставляет три основные команды для управления изображениями:

- `glReadPixels()` читает прямоугольный массив пикселов из буфера кадров и сохраняет данные в оперативной памяти;
- `glDrawPixels()` записывает прямоугольный массив пикселов из данных, извлекаемых из оперативной памяти, в буфер кадров в текущую растровую позицию, заданную командой `glRasterPos*()`;
- `glCopyPixels()` копирует прямоугольный массив пикселов из одной части буфера кадров в другую. Эта команда ведет себя подобно последовательности из вызовов `glReadPixels()` и `glDrawPixels()`, но оперативная память не принимает участия в передаче данных.

Рисунок 8.3 иллюстрирует порядок операций обработки элемента изображения для всех вышеупомянутых команд.

Функция `glRasterPos*()` задает текущую позицию растра, используемую `glDrawPixels()` и `glCopyPixels()`, эти координаты преобразуются конвейером геометрической обработки. Как `glDrawPixels()`, так и `glCopyPixels()` не обходятся без растеризации и пофрагментных операций. (Но при отображении и копировании прямоугольных областей пикселов туман и текстуры теряют смысл.)

Тем не менее эта простая схема существенно усложняется по причине разносортности буфера кадров, наличия множества способов хранения информации о пикселах в памяти компьютера и различных преобразований данных, выполняемых

во время чтения, записи и копирования. Отсюда возникает множество режимов выполнения команд. Если все ваши программы только копируют изображения на экран или читают их в память на какое-то время, чтобы позднее выполнить копирование, тонкости и детали этих режимов вас не должны интересовать. Однако если ваша программа модифицирует данные в памяти — например, выполняет преобразование из формата в формат, или если вам нужно сохранить изображение в файле для последующего его просмотра в другом сеансе или на компьютере другого типа с иными характеристиками графической подсистемы, вы должны понимать разницу между режимами.

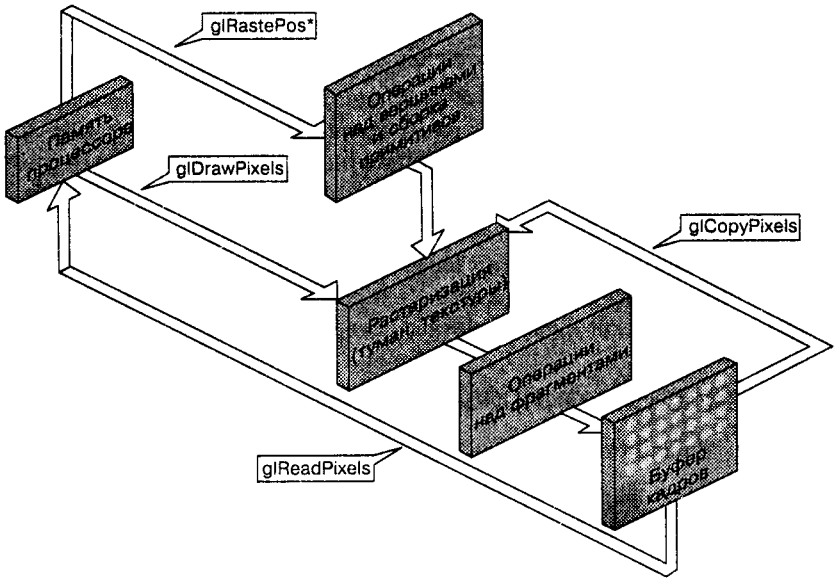


Рис. 8.3. Упрощенная схема следования пикселов

Далее до конца раздела мы описываем основные команды в подробностях. В следующем разделе подробно обсуждается конвейерная обработка изображений: режимы хранения пикселов (pixel-storage modes), операции при передаче пикселов (pixel-transfer operations) и операции табличного преобразования пикселов (pixel-mapping operations).

Чтение пиксельных данных из видеопамяти в оперативную память

void `glReadPixels`(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, GLvoid **pixels*);

Читает из буфера кадров пиксельный прямоугольник, левый нижний угол которого находится в позиции (*x*, *y*) в оконных координатах, с шириной *width* и высотой *height*, и сохраняет данные в массиве *pixels*. Аргумент *format* задает формат читаемого элемента (индекс цвета или компоненты R, G, B, A, в соответствии с табл. 8.1). Аргумент *type* уточняет тип данных каждого элемента (табл. 8.2).

Таблица 8.1. Форматы пикселов для команд glReadPixels() и glDrawPixels()

Формат	Описание
GL_COLOR_INDEX	Индекс цвета
GL_RGB	Красный компонент цвета с сопутствующими зеленым и синим компонентами
GL_RGBA	Красный компонент цвета с сопутствующими зеленым, синим и альфа-компонентами
GL_BGR	Синий компонент цвета с сопутствующими зеленым и красным компонентами
GL_BGRA	Синий компонент цвета с сопутствующими зеленым, красным и альфа-компонентами
GL_RED	Красный компонент цвета
GL_GREEN	Зеленый компонент цвета
GL_BLUE	Синий компонент цвета
GL_ALPHA	Альфа-компонент цвета
GL_LUMINANCE	Компонент яркости
GL_LUMINANCE_ALPHA	Компонент яркости с сопутствующим альфа-компонентом
GL_STENCIL_INDEX	Индекс трафарета
GL_DEPTH_COMPONENT	Компонент глубины

Таблица 8.2. Типы данных для команд glReadPixels() и glDrawPixels()

type Constant	Data Type
GL_UNSIGNED_BYTE	8-разрядное целое число без знака
GL_BYTE	8-разрядное целое число со знаком
GL_BITMAP	Отдельные биты в беззнаковых 8-разрядных целых числах того же формата, что и для glBitmap()
GL_UNSIGNED_SHORT	16-разрядное целое число без знака
GL_SHORT	16-разрядное целое число со знаком
GL_UNSIGNED_INT	32-разрядное целое число без знака
GL_INT	32-разрядное целое число со знаком
GL_FLOAT	Число с плавающей запятой одинарной точности
GL_UNSIGNED_BYTE_3_3_2	Упакованное в 8-разрядное целое число без знака
GL_UNSIGNED_BYTE_2_3_3_REV	Упакованное в 8-разрядное целое число без знака
GL_UNSIGNED_SHORT_5_6_5	Упакованное в 16-разрядное целое число без знака
GL_UNSIGNED_SHORT_5_6_5_REV	Упакованное в 16-разрядное целое число без знака
GL_UNSIGNED_SHORT_4_4_4_4	Упакованное в 16-разрядное целое число без знака
GL_UNSIGNED_SHORT_4_4_4_4_REV	Упакованное в 16-разрядное целое число без знака
GL_UNSIGNED_SHORT_5_5_5_1	Упакованное в 16-разрядное целое число без знака
GL_UNSIGNED_SHORT_1_5_5_5_REV	Упакованное в 16-разрядное целое число без знака
GL_UNSIGNED_INT_8_8_8_8	Упакованное в 32-разрядное целое число без знака
GL_UNSIGNED_INT_8_8_8_8_REV	Упакованное в 32-разрядное целое число без знака
GL_UNSIGNED_INT_10_10_10_2	Упакованное в 32-разрядное целое число без знака
GL_UNSIGNED_INT_2_10_10_10_REV	Упакованное в 32-разрядное целое число без знака

Если вы используете команду `glReadPixels()` для получения RGBA-данных или определенных индексом данных, требуется уточнить буфер для доступа. Например, если ваше окно имеет двойную буферизацию, нужно указать, следует читать данные из буфера переднего плана или заднего плана. Для этого используется команда `glReadBuffer()`. (См. раздел «Выбор буферов цвета для записи и чтения» главы 10.)

ПРИМЕЧАНИЕ

Форматы GL_*_REV особенно полезны в операционных системах Microsoft Windows.

Помните, что в зависимости от формата читается (или записывается) от одного до четырех элементов. Например, если выбран формат `GL_RGBA` и вы читаете в 32-битовые целые числа (то есть если *type* равен `GL_UNSIGNED_INT` или `GL_INT`), каждая операция чтения потребует 16 байт памяти (четыре компонента по четыре байта на компонент).

Таблица 8.2 показывает, как каждый элемент хранится в памяти. Если значение представляет собой не дискретную величину, например компоненты цвета или яркость, оно масштабируется с целью вписаться в отведенное количество битов. Так, допустим, что красный компонент изначально указывается числом с плавающей запятой в интервале между 0 и 1. При необходимости упаковать его в байт без знака точность будет ограничена восемью разрядами, невзирая на то, сколько реально число занимает в буфере кадров. Величины `GL_UNSIGNED_SHORT` и `GL_UNSIGNED_INT` дают 16 и 32 бита точности соответственно. Знаковые версии: `GL_BYTE`, `GL_SHORT` и `GL_INT` — имеют 7, 15 и 31 бит точности, поскольку отрицательными значениями цвет обычно не представляется.

Если элемент является индексом (цвета, трафарета и др.) и тип не `GL_FLOAT`, значение просто маскируется по количеству доступных для типа битов. Маски для знаковых версий — `GL_BYTE`, `GL_SHORT` и `GL_INT` — на один бит меньше. Например, если индекс цвета задается 8-битовым целым со знаком, значение сначала маскируется как 0x7F. Если тип — `GL_FLOAT`, индекс просто конвертируется в число с плавающей запятой одинарной точности (например, индекс 17 преобразуется в вещественное число 17,0).

Для «упакованных» типов данных (константы которых начинаются с `GL_UNSIGNED_BYTE_*`, `GL_UNSIGNED_SHORT_*` или `GL_UNSIGNED_INT_*`) все компоненты цвета каждого пиксела уплотнены в отдельный беззнаковый тип данных: один байт, короткое целое или обычное целое число. Допустимые форматы для каждого типа представлены в табл. 8.3. Если используется недействительный формат, генерируется ошибка `GL_INVALID_OPERATION`.

Порядок значений цвета в битовых полях для упакованных данных определяется форматом пиксела и наличием в константе суффикса `_REV`. Без последнего компоненты цвета располагаются обычным образом, когда первый компонент находится в самых старших разрядах. Если задан суффикс `_REV`, компоненты пакуются в обратной последовательности, начиная с младших разрядов.

Таблица 8.3. Допустимые форматы пикселов для упакованных типов данных

Тип упакованных данных	Действующие форматы
GL_UNSIGNED_BYTE_3_3_2	GL_RGB
GL_UNSIGNED_BYTE_2_3_3_REV	GL_RGB
GL_UNSIGNED_SHORT_5_6_5	GL_RGB
GL_UNSIGNED_SHORT_5_6_5_REV	GL_RGB
GL_UNSIGNED_SHORT_4_4_4_4	GL_RGBA, GL_BGRA
GL_UNSIGNED_SHORT_4_4_4_4_REV	GL_RGBA, GL_BGRA
GL_UNSIGNED_SHORT_5_5_5_1	GL_RGBA, GL_BGRA
GL_UNSIGNED_SHORT_1_5_5_5_REV	GL_RGBA, GL_BGRA
GL_UNSIGNED_INT_8_8_8_8	GL_RGBA, GL_BGRA
GL_UNSIGNED_INT_8_8_8_8_REV	GL_RGBA, GL_BGRA
GL_UNSIGNED_INT_10_10_10_2	GL_RGBA, GL_BGRA
GL_UNSIGNED_INT_2_10_10_10_REV	GL_RGBA, GL_BGRA

Сказанное иллюстрирует рис. 8.4, на примерах битовых полей GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, и четырех допустимых сочетаний типа GL_UNSIGNED_SHORT_4_4_4_4 (и _REV) с форматами RGBA/BGRA. Битовые поля для всех прочих 14 допустимых комбинаций выглядят аналогично.

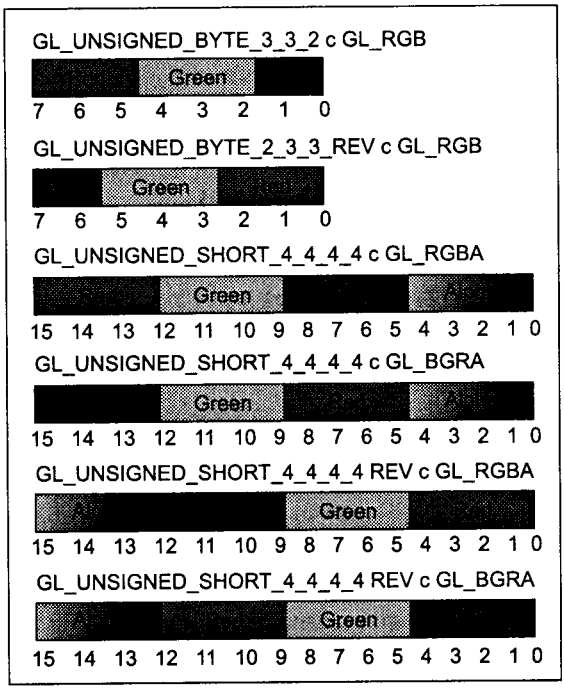


Рис. 8.4. Порядок размещения компонентов для некоторых типов данных и форматов пиксела

Самый старший бит каждого компонента цвета всегда пакуется в позицию самого старшего бита отведенного поля. Размещение отдельного компонента не зависит от режима хранения пикселов, но подчиняется принятому порядку следования байтов в слове. (См. раздел «Режимы хранения пикселов».)

Запись пикселов из оперативной памяти в буфер кадров

void `glDrawPixels`(GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **pixels*);

Отображает прямоугольник из пикселов, с шириной *width* и высотой *height*. Левый нижний угол прямоугольника соответствует текущей позиции растра. Аргументы *format* и *type* несут ту же нагрузку, что и для функции `glReadPixels()` (см. табл. 8.1 и 8.2.). Массив *pixels* содержит данные для отрисовки. Если текущая растровая позиция некорректна, никакого вывода не производится.

Листинг 8.3 представляет собой часть программы, использующей `glDrawPixels()` для отображения прямоугольника пикселов в левом нижнем углу окна. Процедура `makeCheckImage()` создает RGB-массив 64 × 64 черно-белого изображения шахматного поля. Команда `glRasterPos2i(0, 0)` позиционирует изображение в левый нижний угол. И опять-таки пока не обращайте внимания на команду `glPixelStorei()`.

Листинг 8.3. Использование команды glDrawPixels(): image.c

```
#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage[checkImageHeight][checkImageWidth][3];

void makeCheckImage(void)
{
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {
      for (j = 0; j < checkImageWidth; j++) {
         c = ((((i&0x8)==0)^((j&0x8))==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
   }
   }
}

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(0, 0);
   glDrawPixels(checkImageWidth, checkImageHeight, GL_RGB,
```

```
    GL_UNSIGNED_BYTE, checkImage);
  glFlush();
}
```

Когда вы используете `glDrawPixels()` для записи RGBA-данных или индексов цвета, вам может потребоваться выбрать текущий буфер для рисования командой `glDrawBuffer()`, которая, как и `glReadBuffer()`, описана в разделе «Выбор буферов цвета для записи и чтения» главы 10.

Копирование пикселов в пределах буфера кадров

void `glCopyPixels`(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*, GLenum *buffer*);

Копирует прямоугольную область внутри буфера кадров с позицией левого нижнего угла (*x*, *y*), шириной *width* и высотой *height*. Левый нижний угол целевого прямоугольника совпадает с текущей позицией растра. Аргумент *buffer* это либо `GL_COLOR`, либо `GL_STENCIL`, либо `GL_DEPTH`, он указывает тип используемого буфера. Функция `glCopyPixels()` ведет себя аналогично последовательности из `glReadPixels()` и `glDrawPixels()`, с промежуточным «преобразованием» константы *buffer* в аргумент *format*:

- если *buffer* — `GL_DEPTH` или `GL_STENCIL`, используется `GL_DEPTH_COMPONENT` или `GL_STENCIL_INDEX` соответственно;
- если указано значение `GL_COLOR`, используется `GL_RGBA` или `GL_COLOR_INDEX`, в зависимости от режима — RGBA или индекса цвета.

Заметьте, что в аргументах *format* или *data* при использовании `glCopyPixels()` нет необходимости, так как данные никогда не поступают через оперативную память. Исходный и целевой буферы для `glCopyPixels()` указываются с помощью `glReadBuffer()` и `glDrawBuffer()` соответственно. Как `glDrawPixels()`, так и `glCopyPixels()` используется в листинге 8.4.

Для всех трех функций то, насколько верно преобразуются поступающие из буфера кадров или передаваемые в него данные, зависит от текущего режима. Об этом мы поговорим в следующем разделе.

Конвейерная обработка изображений

В этом разделе рассказывается о конвейеризации обработки изображений (imaging pipeline): режимах сохранения пикселов и операциях при передаче пикселов, включающих в себя произвольные преобразования данных. Вы можете растянуть или сократить прямоугольную область перед ее отображением с помощью команды `glPixelZoom()`. Последовательность этих операций показана на рис. 8.5.

При вызове `glDrawPixels()` данные сначала распаковываются из оперативной памяти согласно режимам хранения пикселов, а затем к ним применяются преобразования при передаче. Результат растеризуется. Во время растеризации прямоугольник пикселов может быть масштабирован в обе стороны, в зависимости от его текущего состояния. В заключение выполняются операции над фрагментами

и пикселы записываются в буфер кадров. (См. раздел «Тесты и операции над фрагментами» главы 10).

При вызове `glReadPixels()` данные считываются из буфера кадров, аналогичным образом преобразуются и упаковываются в оперативную память.

Функция `glCopyPixels()` выполняет промежуточные преобразования при передаче, подобно `glReadPixels()`. Результирующие данные записываются подобно `glDrawPixels()`, но во второй раз преобразования не применяются. Рисунок 8.6 демонстрирует, как `glCopyPixels()` передает пикселы от самого буфера кадров.

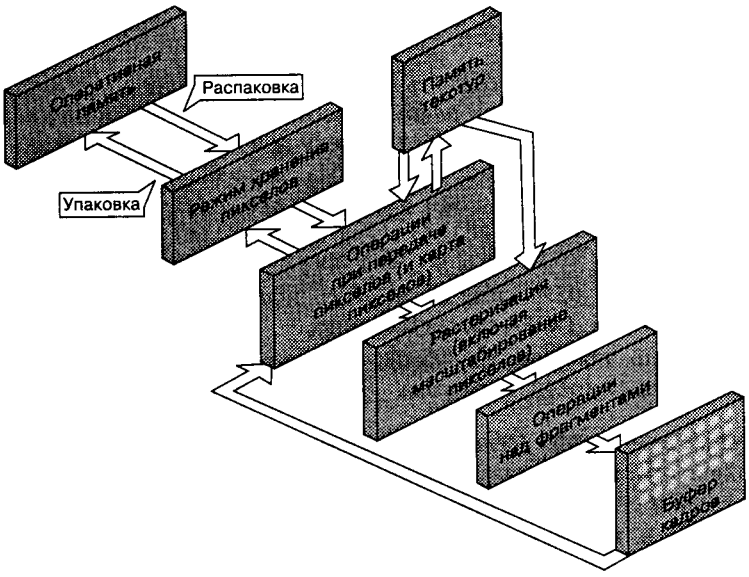


Рис. 8.5. Конвейеризация обработки изображений

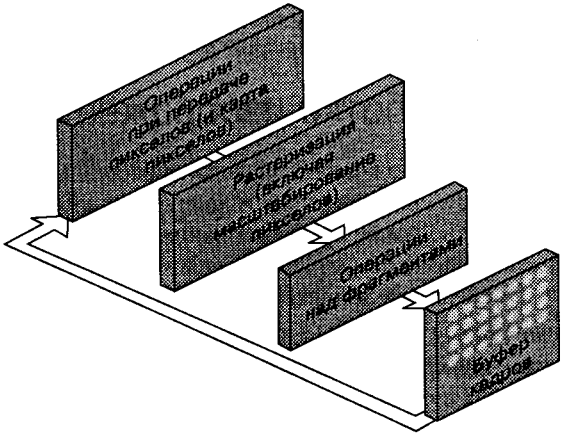


Рис. 8.6. Маршрут пикселов при выполнении glCopyPixels()

Из раздела «Вывод битового образа» и рис. 8.7 вы можете составить представление, что воспроизведение битового образа проще воспроизведения изображений за счет отсутствия перечисленных выше промежуточных преобразований.

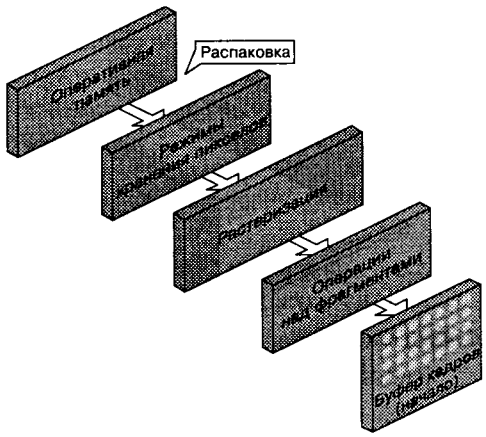


Рис. 8.7. Путь пикселов при выполнении glBitmap()

Обратите внимание, что режимы сохранения пикселов и операции при передаче применяются к текстурам, когда те читаются из памяти текстур или записываются в нее. Рисунок 8.8 демонстрирует эффект от команд `glTexImage*()`, `glTexSubImage*()` и `glGetTexImage()`.

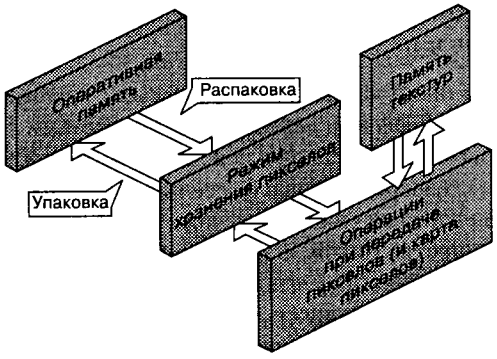


Рис. 8.8. Пути пикселов при выполнении glTexImage*(), glTexSubImage*() и glGetTexImage()

Как видно из рис. 8.9, когда пикселы копируются из буфера кадров в память текстур (`glCopyTexImage*()` или `glCopyTexSubImage*()`), применяются только операции при передаче. (См. главу 9 для получения большей информации о текстурах.)

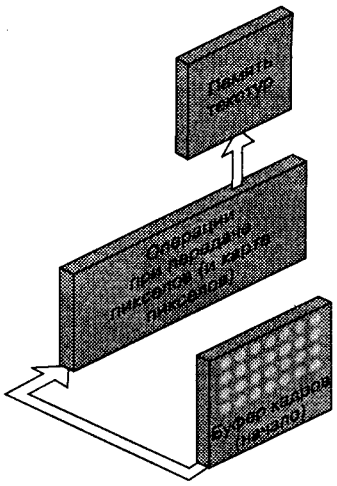


Рис. 8.9. Пути пикселов при выполнении glCopyTexImage*() и glCopyTexSubImage*()

Упаковка и распаковка пикселов

Упаковываются или распаковываются пикселы, определяется тем, в каком направлении следуют данные — на запись или чтение из оперативной памяти.

Изображение сохраняется в памяти в нескольких *элементах*. Данные могут представляться просто индексом цвета или значением яркости (то есть средневзвешенным значением красного, зеленого и синего компонентов) или же быть выражены красным, зеленым, синим и альфа-компонентом (RGBA) для каждого пиксела. Возможное размещение данных пикселов, то есть *формат*, определяется количеством элементов и их порядком.

Некоторые элементы (такие как индексы цвета или трафарета) являются целочисленными, а другие (RGBA или глубины) — значениями с плавающей запятой (вещественными), обычно в диапазоне между 0 и 1. Вещественные компоненты обычно сохраняются в буфере кадров с более низким разрешением, чем позволяет число с плавающей запятой (например, компонентам цвета может отводиться по 8 бит). Точное количество битов, используемых для представления компонентов, зависит от особенностей аппаратной части. Таким образом, сохранение каждого компонента в виде 32-разрядного вещественного числа часто теряет смысл, тем более что изображения могут содержать миллионы пикселов.

Элементы могут сохраняться в памяти с различными типами данных, от 8-разрядного байта до 32-разрядного целого или числа с плавающей запятой. OpenGL явно определяет преобразование каждого компонента в каждом формате для каждого допустимого типа данных. Имейте в виду, что вы можете потерять данные, если попытаетесь сохранить компонент высокого разрешения с типом, представимым малым числом битов.

Режимы хранения пикселов

Данные изображения обычно сохраняются в оперативной памяти в виде прямоугольных (двух- или трехмерных) массивов. Часто нужно отобразить или сохранить только часть изображения, то есть вложенный прямоугольник. Кроме того, вы должны принять во внимание, что различные компьютеры имеют различный порядок следования байтов. Наконец, аппаратное обеспечение некоторых машин куда более эффективно работает с данными, выровненными по 2-, 4- или 8-байтовой границе оперативной памяти. Для таких компьютеров вы, вероятно, пожелаете управлять выравниванием. За все, что было упомянуто в этом абзаце, отвечают режимы хранения пикселов, обсуждаемые ниже. Вы указываете эти режимы с помощью команды `glPixelStore*()`, которую мы пока просили проигнорировать в паре наших примеров.

Все поддерживаемые OpenGL режимы сохранения пикселов управляются командой `glPixelStore*()`. Как правило, выполняется несколько последовательных вызовов этой команды с целью установить значения нескольких параметров.

> void `glPixelStore`{if}(GLenum *pname*, *TYPE param*);
>
> Устанавливает режимы сохранения пикселов применительно к операциям `glDrawPixels()`, `glReadPixels()`, `glBitmap()`, `glPolygonStipple()`, `glTexImage1D()`, `glTexImage2D()`, `glTexImage3D()`, `glTexSubImage1D()`, `glTexSubImage2D()`, `glTexSubImage3D()`, `glGetTexImage()` и, если поддержана конвейеризация обработки изображений, также к `glGetColorTable()`, `glGetConvolutionFilter()`, `glGetSeparableFilter()`, `glGetHistogram()` и `glGetMinmax()`.
>
> Возможные имена параметров (*pname*) представлены в табл. 8.4 вместе с типами данных, начальными значениями и допустимыми диапазонами значений. Параметры `GL_UNPACK*` управляют тем, как выполняют распаковку данных из памяти функции `glDrawPixels()`, `glBitmap()`, `glPolygonStipple()`, `glTexImage1D()`, `glTexImage2D()`, `glTexImage3D()`, `glTexSubImage1D()`, `glTexSubImage2D()` и `glTexSubImage3D()`. Параметры `GL_PACK*` аналогично отвечают за упаковку в память при выполнении `glReadPixels()` и `glGetTexImage()` и, если доступна конвейеризация, также для `glGetColorTable()`, `glGetConvolutionFilter()`, `glGetSeparableFilter()`, `glGetHistogram()` и `glGetMinmax()`.
>
> Параметры `GL_UNPACK_IMAGE_HEIGHT`, `GL_PACK_IMAGE_HEIGHT`, `GL_UNPACK_SKIP_IMAGES` и `GL_PACK_SKIP_IMAGES` затрагивают только 3D-текстурирование (`glTexImage3D()`, `glTexSubImage3D()` и `glGetTexImage(GL_TEXTURE_3D,..)`).

Поскольку соответствующие упаковке и распаковке параметры имеют одинаковые значения, они обсуждаются далее вместе и упоминаются без префикса `GL_PACK` или `GL_UNPACK`. Например, `*SWAP_BYTES` подразумевает как `GL_PACK_SWAP_BYTES`, так и `GL_UNPACK_SWAP_BYTES`.

Если параметр `*SWAP_BYTES` равен `FALSE` (по умолчанию), действует «родной» для клиента OpenGL порядок следования байтов в памяти, и в противном случае – обратный. Изменение порядка следования применимо к элементам любого размера, но имеет смысл только для многобайтовых элементов.

Таблица 8.4. Параметры команды glPixelStore()

Название параметра	Тип	Значение инициализации	Допустимый диапазон
GL_UNPACK_SWAP_BYTES, GL_PACK_SWAP_BYTES	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_LSB_FIRST, GL_PACK_LSB_FIRST	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH	GLint	0	Любое неотрицательное целое число
GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS	GLint	0	Любое неотрицательное целое число
GL_UNPACK_SKIP_PIXELS, GL_PACK_SKIP_PIXELS	GLint	0	Любое неотрицательное целое число
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT	GLint	4	1, 2, 4, 8
GL_UNPACK_IMAGE_HEIGHT, GL_PACK_IMAGE_HEIGHT	GLint	0	Любое неотрицательное целое число
GL_UNPACK_SKIP_IMAGES, GL_PACK_SKIP_IMAGES	GLint	0	Любое неотрицательное целое число

Эффект от перестановки байтов может различаться от реализации OpenGL к реализации. Так, рис. 8.10 иллюстрирует перестановку байтов для различных типов данных в предположении, что в некоторой системе тип GLubyte имеет размер 8 бит, GLushort — 16 бит и GLuint — 32 бита. Заметьте, что перестановка не влияет на однобайтовые данные.

ПРИМЕЧАНИЕ

Пока ваше приложение OpenGL работает с изображениями само по себе, не разделяя их с другими машинами, вы можете игнорировать аспект порядка следования байтов. Но если оно должно воспроизвести изображение OpenGL, созданное на другой машине, с иным расположением байтов в слове, параметр *SWAP_BYTES в вашем распоряжении. Тем не менее параметр *SWAP_BYTES не позволяет переупорядочивать элементы (например, переставлять местами зеленый и красный).

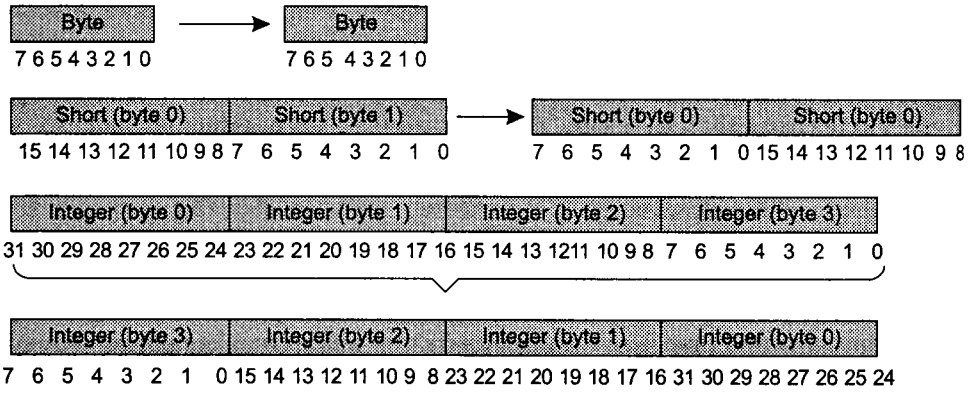


Рис. 8.10. Эффект для перестановки байтов для типов Byte, Short и Integer

Параметр *LSB_FIRST применяется только при отображении или чтении одноразрядных изображений или битовых образов (где одному пикселу соответствует один бит). Если значение *LSB_FIRST равно FALSE (по умолчанию), биты каждого байта отсчитываются от позиции самого старшего бита; иначе подразумевается противоположный порядок. Например, если *LSB_FIRST равно FALSE, а интересующий байт — 0x31, биты следуют в последовательности {0, 0, 1, 1, 0, 0, 0, 1}. Если *LSB_FIRST равно TRUE, эта последовательность выглядит так: {1, 0, 0, 0, 1, 1, 0, 0}.

Иногда нужно отрисовать или прочитать только прямоугольник, являющийся частью изображения, сохраненного в памяти. В таком случае требуется указать действительную длину (в пикселах) большего прямоугольника с помощью параметра *ROW_LENGTH. Если значение *ROW_LENGTH нулевое (по умолчанию), длина ряда приравнивается ширине, указанной в glReadPixels(), glDrawPixels() или glCopyPixels(). Также требуется задать количество рядов и пикселов в ряду, пропускаемых перед копированием вложенного прямоугольника. Эти числа задаются параметрами *SKIP_ROWS и *SKIP_PIXELS, как показано на рис. 8.11. По умолчанию оба значения равны 0, то есть обработка осуществляется от левого нижнего угла.

Часто аппаратные средства компьютеров оптимизированы под выровненные в памяти данные. Например, если длина слова в компьютере 32 бита, оборудование с большей эффективностью будет обращаться к данным, выровненным по 32-разрядной границе, адреса которых обычно кратны 4. Аналогично, 64-битовые архитектуры могут работать лучше, когда данные выровнены по 8-байтовым границам. На некоторых компьютерах, тем не менее, выравнивание не играет роли.

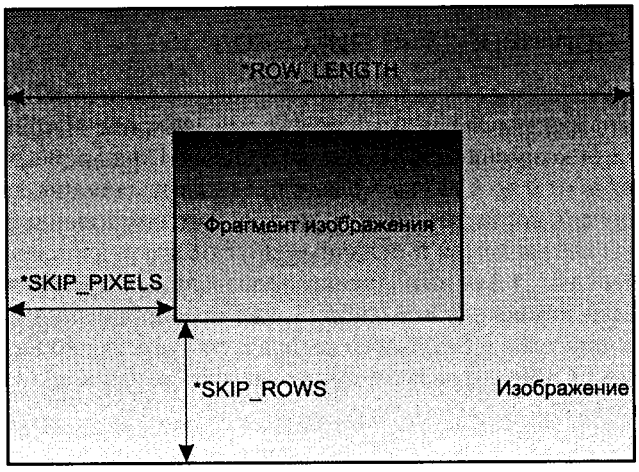


Рис. 8.11. Параметры *SKIP_ROWS, *SKIP_PIXELS и *ROW_LENGTH

Предположим, что ваша машина лучше справляется с пикселами, выровненными по 4-байтовой границе. Изображения хранятся с учетом этого фактора, и каждый ряд начинается с адреса, кратного 4 байтам. Если изображение имеет ширину

5 пикселов и для каждого пиксела отводится по байту на компонент (красный, зеленый и синий), то вся строка займет 5 × 3 = 15 байт. Для достижения максимальной эффективности требуется дополнительное выделение еще одного байта под каждую строку. Если данные выровнены подобным образом, следует задать параметр `*ALIGNMENT` (4 в нашем случае).

Если значение `*ALIGNMENT` установлено в 1, задействуется каждый байт. Если оно равно 2, при необходимости пропускается один байт в конце строки для того, чтобы следующая строка начиналась с четного адреса. В случае битовых образов (или одноразрядных изображений) выравнивание производится аналогично, с той разницей, что обрабатываются не байты, а биты. Так, если за пиксел отвечает всего один бит, длина строки равна 75 и выравнивание производится по значению 4, каждая строка требует для себя 75/8, то есть 9 3/8 байта. Поскольку 12 является наименьшим числом, одновременно кратным 4 и большим числа 9 3/8, каждый ряд будет занимать в памяти 12 байт. Если параметр выравнивания равен 1, будут использоваться 10 байт каждой строки, то есть округленное вверх значение 9 3/8. (Пример применения `glPixelStorei()` см. в листинге 8.4.)

ПРИМЕЧАНИЕ

Значение по умолчанию для параметра *ALIGNMENT равно 4. Допущение, что данные изображения плотно упакованы и выровнены по байтовой границе (*ALIGNMENT равно 1) — весьма часто встречаемая ошибка.

Параметры `*IMAGE_HEIGHT` и `*SKIP_IMAGES` имеют смысл только в отношении трехмерных текстур. (См. раздел «Режимы хранения пикселов для трехмерных текстур» главы 9.)

Операции при передаче пикселов

Во время передачи данных из памяти в буфер кадров (или наоборот) OpenGL может производить над этими данными некоторые действия. Например, изменяются диапазоны для компонентов — обычно значения красного компонента находятся в интервале от 0 до 1, но вы можете предпочесть для них другой диапазон; или возьмем случай, когда данные того же красного компонента в разных графических системах сохраняются в отличных друг от друга диапазонах. Вы можете создавать карты (maps) для выполнения произвольных преобразований над индексом цвета или компонентами цвета. Такие преобразования выполняются во время передачи пикселов в (из) буфер кадров и соответственно называются (pixel-transfer operations). Отвечают за такие преобразования команды `glPixelTransfer*()` и `glPixelMap*()`.

Хотя буферы цвета, глубины и трафарета имеют много общего, они не ведут себя идентично и в некоторых режимах имеют свою специфику. Мы подробно рассмотрим все режимы, включая их специальные случаи, в этом и следующих разделах.

Часть характеристик устанавливается с помощью `glPixelTransfer*()`, а часть – указывается посредством `glPixelMap*()`, обсуждаемой в следующем разделе.

void `glPixelTransfer{if}`(GLenum *pname*, *TYPE param*);

Устанавливает режимы передачи пикселов, воздействующие на выполнение `glDrawPixels()`, `glReadPixels()`, `glCopyPixels()`, `glTexImage1D()`, `glTexImage2D()`, `glTexImage3D()`, `glCopyTexImage1D()`, `glCopyTexImage2D()`, `glTexSubImage1D()`, `glTexSubImage2D()`, `glTexSubImage3D()`, `glCopyTexSubImage1D()`, `glCopyTexSubImage2D()`, `glCopyTexSubImage3D()` и `glGetTexImage()`. Аргумент *pname* принимает одно из значений, перечисленных в первом столбце табл. 8.5, а значения аргумента *param* должны быть в допустимом диапазоне, как показано в той же таблице.

Таблица 8.5. Параметры команды glPixelTransfer*()

Название параметра	Тип	Значение инициализации	Допустимый диапазон
GL_MAP_COLOR	GLboolean	FALSE	TRUE/FALSE
GL_MAP_STENCIL	GLboolean	FALSE	TRUE/FALSE
GL_INDEX_SHIFT	GLint	0	$(-\infty, \infty)$
GL_INDEX_OFFSET	GLint	0	$(-\infty, \infty)$
GL_RED_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_GREEN_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_BLUE_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_ALPHA_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_DEPTH_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_RED_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_GREEN_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_BLUE_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_ALPHA_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_DEPTH_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_POST_CONVOLUTION_RED_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_POST_CONVOLUTION_GREEN_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_POST_CONVOLUTION_BLUE_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_POST_CONVOLUTION_ALPHA_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_POST_CONVOLUTION_RED_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_POST_CONVOLUTION_GREEN_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_POST_CONVOLUTION_BLUE_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_POST_CONVOLUTION_ALPHA_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_POST_COLOR_MATRIX_RED_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_POST_COLOR_MATRIX_GREEN_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_POST_COLOR_MATRIX_BLUE_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_POST_COLOR_MATRIX_ALPHA_SCALE	GLfloat	1.0	$(-\infty, \infty)$
GL_POST_COLOR_MATRIX_RED_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_POST_COLOR_MATRIX_GREEN_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_POST_COLOR_MATRIX_BLUE_BIAS	GLfloat	0.0	$(-\infty, \infty)$
GL_POST_COLOR_MATRIX_ALPHA_BIAS	GLfloat	0.0	$(-\infty, \infty)$

ВНИМАНИЕ

Параметры GL_POST_CONVOLUTION_* и GL_POST_COLOR_MATRIX_* имеют смысл, если ваша реализация OpenGL поддерживает конвейеризацию обработки изображений. (См. раздел «Расширение Imaging Subset».)

Если параметр GL_MAP_COLOR или GL_MAP_STENCIL установлен в TRUE, доступно табличное преобразование (mapping). Табличное преобразование рассматривается в следующем подразделе. Все остальные параметры непосредственно относятся к компонентам пикселов.

Масштабирование (scale) и отклонение (bias) применимы к красному, зеленому, синему, альфа-компонентам и к глубине. Например, можно масштабировать компоненты RGB в процессе сведения их к яркости. Яркость вычисляется как сумма красного, зеленого и синего цветов, поэтому, если вы используете для них значения по умолчанию (GL_RED_SCALE, GL_GREEN_SCALE и GL_BLUE_SCALE), каждый из компонентов внесет свой вклад в значение результирующей интенсивности (яркости). Если преобразование из модели RGB к яркости выполняется согласно стандарту NTSC, нужно устанавливать компоненты в конкретные значения: GL_RED_SCALE — 0.3, GL_GREEN_SCALE — 0.59 и GL_BLUE_SCALE — 0.11.

Индексы (цвета и трафарета) также могут быть преобразованы. В этом случае применяются сдвиг (shift) и смещение (offset). Это полезно, если вам нужно управлять частью таблицы цветов, принимающей участие в построении изображения.

Табличные преобразования пикселов

Все компоненты цвета, индексы цвета и трафарета могут быть модифицированы с помощью *таблиц подстановки* (table lookup) перед их помещением в видеопамять. Этот процесс называется *табличным преобразованием* (mapping) и управляется командой `glPixelMap*()`.

void `glPixelMap`{ui us f}v(GLenum *map*, GLint *mapsize*, const *TYPE* **values*);

Загружает пиксельную карту (таблицу) *map*, состоящую из *mapsize* элементов, соответствующих значениям (массива) *values*. Таблица 8.6 содержит список названий карт и значений; по умолчанию все размеры считаются равными 1, а значения — 0. Размер каждой карты должен быть степенью 2.

Таблица 8.6. Названия и значения аргументов команды glPixelMap*()

Название карты	Адрес	Значение
GL_PIXEL_MAP_I_TO_I	Индекс цвета	Индекс цвета
GL_PIXEL_MAP_S_TO_S	Индекс трафарета	Индекс трафарета
GL_PIXEL_MAP_I_TO_R	Индекс цвета	R
GL_PIXEL_MAP_I_TO_G	Индекс цвета	G
GL_PIXEL_MAP_I_TO_B	Индекс цвета	B
GL_PIXEL_MAP_I_TO_A	Индекс цвета	A
GL_PIXEL_MAP_R_TO_R	R	R

Название карты	Адрес	Значение
GL_PIXEL_MAP_G_TO_G	G	G
GL_PIXEL_MAP_B_TO_B	B	B
GL_PIXEL_MAP_A_TO_A	A	A

Максимальный размер карт машинно-зависим. Вы можете узнать, какие размеры пиксельных карт поддерживаются на вашей машине, с помощью команды `glGetIntegerv()`. Если нужно узнать размер по всем картам, используйте для запроса параметр `GL_MAX_PIXEL_MAP_TABLE`, и используйте параметр `GL_PIXEL_MAP_*_TO_*_SIZE`, чтобы узнать текущий размер конкретной карты. Для шести карт, адресуемых индексом цвета или трафарета, подразумевается размер, кратный степени 2. Четыре RGBA-карты могут иметь любой размер от 1 до значения `GL_MAX_PIXEL_MAP_TABLE`.

Чтобы понять, как работают таблицы, рассмотрим простой пример. Предположим, что вы хотите создать 256-элементную таблицу, отображающую индекс цвета в индекс цвета с помощью параметра `GL_PIXEL_MAP_I_TO_I`. Вы создаете таблицу с элементами для каждого из значений в диапазоне 0–255 и инициируете ее посредством `glPixelMap*()`. Допустим, что вам нужна таблица для пороговой обработки (thresholding), индексы, меньшие 101 (0–100), требуется установить в 0, а все индексы от 101 — заменить числом 255. В этом случае ваша таблица будет состоять из 101 нуля и из 155 чисел 255. Чтобы сделать пиксельную карту доступной, используйте `glPixelTransfer*()`, установив параметр `GL_MAP_COLOR` в `TRUE`. После того как пиксельная карта загружена и доступна, все исходные цветовые индексы могут быть преобразованы: те, которые меньше 101, станут нулями, а принадлежащие диапазону 101–255 отобразятся в 255. Если исходный индекс больше 255, он сначала маскируется числом 255, то есть сбрасываются все биты старше восьмого, а результирующее значение замещается числом из таблицы. Если исходный индекс представлен числом с плавающей запятой (скажем, 88,14585), он будет округлен до ближайшего целого значения (давая в итоге 88), и тогда уже этот номер ищется в таблице (получаем 0).

Используя пиксельные карты, вы можете также отображать индексы трафарета или преобразовывать индексы цвета в модель RGB. (См. раздел «Чтение и рисование прямоугольных областей» для получения информации о преобразовании индексов.)

Растяжение, сжатие и зеркальное отражение изображений

После применения режимов сохранения пикселов и операций при передаче пикселов изображения и битовые образы растеризуются. В простом варианте каждому пикселу изображения соответствует один пиксел экрана. Вместе с тем вы вправе произвольно увеличивать и уменьшать изображение и даже зеркально отражать (flip, reflect) его с помощью команды `glPixelZoom()`.

void `glPixelZoom`(GLfloat *zoomx*, GLfloat *zoomy*);

Задает коэффициенты растяжения (magnify) и сжатия (reduce) для операций записи пикселов (`glDrawPixels()` и `glCopyPixels()`) по горизонтали и вертикали. По умолчанию значения $zoom_x$ и $zoom_y$ устанавливаются в 1. Так, если оба значения равны 2,0, каждый пиксел изображения рисуется четырьмя пикселами экрана. Заметьте, что допускаются дробные значения факторов масштабирования, а также отрицательные. Последние служат для зеркального отражения изображения относительно текущей позиции растра.

Во время растеризации каждый пиксел изображения интерпретируется как прямоугольник с размерами $zoom_x \times zoom_y$, и фрагменты генерируются для всех пикселов, центры которых находятся в пределах этого прямоугольника. Если говорить более точно, (x_r, y_r) является текущей позицией растра. Если определенная группа элементов (индексов или компонентов) строки n принадлежит столбцу m, рассматриваемая нами область в оконных координатах ограничена прямоугольником с углами

$$(x_r + zoom_x \times n, y_r + zoom_y \times m) \text{ и } (x_r + zoom_x(n + 1), y_r + zoom_y(m + 1)).$$

Все фрагменты с центрами внутри прямоугольника (или на нижней или левой границе) ставятся в соответствие означенной группе элементов.

Отрицательные значения *zoom* используются при зеркальном отражении изображений. OpenGL описывает изображения в направлении от нижнего ряда пикселов к верхнему (и слева направо). Если вы имеете изображение вида «сверху вниз», такое как видеокадр, вы можете захотеть воспользоваться операцией `glPixelZoom` (1.0, 1.0), чтобы получить из изображения правильное с точки зрения OpenGL. Убедитесь, что вы задали верную текущую позицию растра, если это требуется.

Листинг 8.4 демонстрирует использование `glPixelZoom()`. Изображение шахматной доски изначально находится в левом нижнем углу окна. При нажатии кнопки мыши или при передвижении мыши вы можете использовать `glCopyPixels()` для копирования изображения из исходного положения в окне в точку, соответствующую положению указателя мыши. (Разумеется, копировать изображение само в себя не имеет смысла!) Копируемое изображение масштабируется; по умолчанию с коэффициентом 1.0, но вы вправе задать другое значение. Клавиши z и Z связаны с увеличением и уменьшением коэффициента масштабирования на значение 0.5. Любое «повреждение» содержимого окна влечет за собой перерисовку последнего. Нажатие клавиши r сбрасывает коэффициенты в первоначальное состояние.

Листинг 8.4. Рисование, копирование и масштабирование пикселов: image.c

```
#define checkImageWidth 64
#define checkImageHeight 64

GLubyte checkImage[checkImageHeight][checkImageWidth][3];
static GLdouble zoomFactor = 1.0;
static GLint height;
```

```
void makeCheckImage(void)
{
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {
      for (j = 0; j < checkImageWidth; j++) {
         c = ((((i&0x8)==0)^((j&0x8))==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
      }
   }
}

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(0, 0);
   glDrawPixels(checkImageWidth, checkImageHeight, GL_RGB,
      GL_UNSIGNED_BYTE, checkImage);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   height = (GLint) h;
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void motion(int x, int y)
{
   static GLint screeny;

   screeny = height - (GLint) y;
   glRasterPos2i(x, screeny);
   glPixelZoom(zoomFactor, zoomFactor);
   glCopyPixels(0, 0, checkImageWidth, checkImageHeight,
      GL_COLOR);
   glPixelZoom(1.0, 1.0);
   glFlush();
}
```

продолжение ⇩

Листинг 8.4 (*продолжение*)

```
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 'r':
      case 'R':
         zoomFactor = 1.0;
         glutPostRedisplay();
         printf("масштабный коэффициент сброшен в 1.0\n");
         break;
      case 'z':
         zoomFactor += 0.5;
         if (zoomFactor >= 3.0)
            zoomFactor = 3.0;
         printf("текущее значение масштабного коэффициента %4.1f\n",
zoomFactor);
         break;
      case 'Z':
         zoomFactor -= 0.5;
         if (zoomFactor <= 0.5)
            zoomFactor = 0.5;
         printf("текущее значение масштабного коэффициента %4.1f\n",
zoomFactor);
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
```

Чтение и рисование прямоугольных областей

Этот раздел подробно описывает процесс чтения и отображения прямоугольных областей пикселов. Преобразование при чтении из буфера кадров в память похоже на преобразование в обратном направлении — рисование, но не идентично ему. Материал этого раздела не обязателен при первом чтении, особенно если вы не планируете сразу начать выполнять подобные преобразования.

Процесс вывода пиксельного прямоугольника

Следующий список и рис. 8.12 описывают операции отображения пикселов в буфер кадров.

1. Если пикселы представлены не индексами (то есть их формат отличен от `GL_COLOR_INDEX` или `GL_STENCIL_INDEX`), на первом шаге компоненты преобразуются в числа с плавающей запятой. (Подробности преобразования представлены в табл. 4.1.)
2. Если задан формат `GL_LUMINANCE` или `GL_LUMINANCE_ALPHA`, элемент яркости преобразуется в компоненты R, G и B с назначением равной доли каждому

компоненту. Если формат — GL_LUMINANCE_ALPHA, величина альфа-компонента равна значению A, которое, если указан параметр GL_LUMINANCE, устанавливается в 1.0.

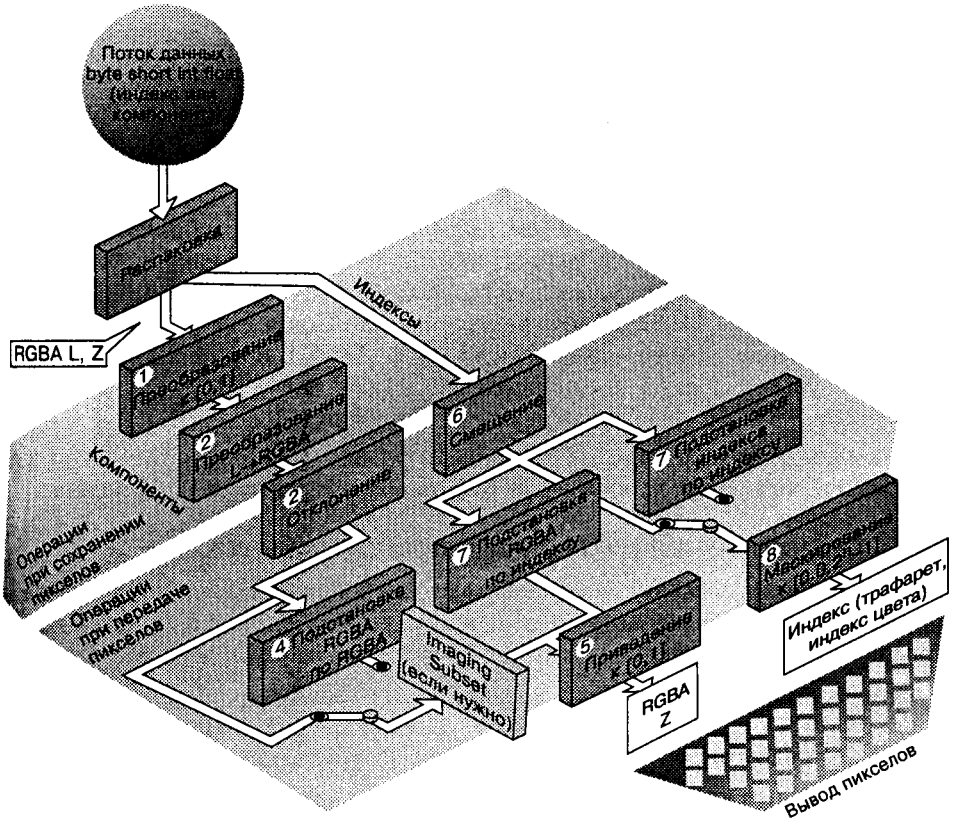


Рис. 8.12. Рисование пикселов с помощью glDrawPixels()

3. Каждый компонент (R, G, B, A или глубина) умножается на соответствующий размерный коэффициент, с добавлением смещения (bias). Например, компонент R умножается на величину GL_RED_SCALE и после к нему добавляется значение GL_RED_BIAS.
4. Если параметр GL_MAP_COLOR установлен в TRUE, каждый из R-, G-, B- и A-компонентов, ограниченный диапазоном [0.0, 1.0], умножается на целое число, меньшее на единицу размера таблицы, обрезается и ищется в таблице подстановки. (См. раздел «Советы по увеличению быстродействия вывода пикселов».)
5. Затем R-, G-, B- и A-компоненты ограничиваются диапазоном [0.0, 1.0], если это еще не сделано, и преобразуются в число с десятичной точкой с таким количеством битов левее точки, сколько нужно для помещения в буфер кадров.
6. Если вы работаете с индексами (цвета или трафарета), сначала значения преобразуются в числа с десятичной точкой (если они исходно представлены

в формате с плавающей запятой), с некоторым количеством неопределенных битов справа от точки. Для индексов, изначально представленных десятичными числами, любые биты справа от десятичной точки сбрасываются в нуль.

Результирующее значение индекса сдвигается вправо или влево в соответствии с абсолютным значением параметра GL_INDEX_SHIFT; значение сдвигается влево, когда GL_INDEX_SHIFT > 0, и вправо — в противном случае. В заключение к индексу добавляется смещение GL_INDEX_OFFSET.

7. Следующий шаг зависит от используемой модели цвета — RGBA или индексной. В первом случае индекс цвета преобразуется в RGBA с использованием параметров GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B и GL_PIXEL_MAP_I_TO_A (см. раздел «Табличные преобразования пикселов»). Во втором варианте, если GL_MAP_COLOR определен как GL_TRUE, индекс цвета замещается по таблице подстановки GL_PIXEL_MAP_I_TO_I. (Если GL_MAP_COLOR — GL_FALSE, индекс не изменяется.) Когда изображение представлено индексами трафарета и если GL_MAP_STENCIL определен как GL_TRUE, используется таблица подстановки GL_PIXEL_MAP_S_TO_S. Если GL_MAP_STENCIL — FALSE, индекс трафарета не изменяется.
8. Наконец, если индексы еще не конвертированы в RGBA, они маскируются к количеству битов, нужному для буфера индекса цвета или трафарета.

Процесс считывания пиксельного прямоугольника

Многие из преобразований, имеющих место во время отображения пиксельного прямоугольника, применяются и в процессе чтения данных из буфера кадров. Порядок чтения пикселов представлен следующим списком и на рис. 8.13.

1. Если читаемые пикселы не индексированы (то есть их формат отличен от GL_COLOR_INDEX или GL_STENCIL_INDEX), они попадают в диапазон [0.0, 1.0] — то есть имеет место противоположная процессу рисования ситуация.
2. Затем каждый компонент масштабируется со смещением. Если параметр GL_MAP_COLOR равен GL_TRUE, эти компоненты таблично преобразуются и снова сводятся к диапазону [0.0; 1.0]. При преобразовании от RGB к яркости компоненты R, G и B суммируются (L = R + G + B).
3. Если пикселы представлены индексами (цвета или трафарета), они сдвигаются (shifted), смещаются (offset) и, если параметр GL_MAP_COLOR установлен в GL_TRUE, таблично преобразуются.
4. Если формат хранения пикселов — GL_COLOR_INDEX или GL_STENCIL_INDEX, индексы маскируются к количеству битов, поддерживаемому устройством хранения (1, 8, 16 или 32), и упаковываются в память, как описано ранее.
5. Если формат хранения — один из компонентных типов (такой как яркость или RGB), пикселы всегда таблично преобразуются по карте «индекс-в-RGBA». Тогда они интерпретируются так же, как если бы были пикселами RGBA и ранее (включая возможное преобразование к яркости).
6. Наконец, результаты предыдущих шагов упаковываются в память согласно режимам GL_PACK*, устанавливаемым glPixelStore*().

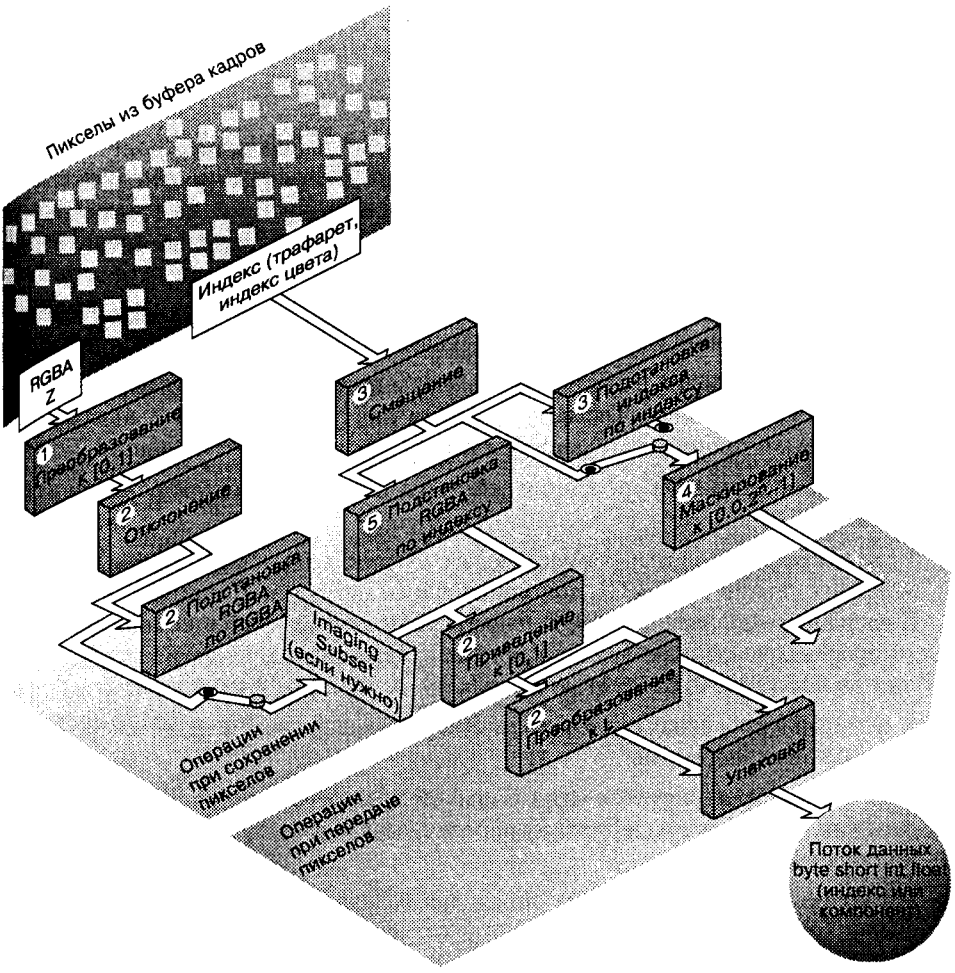


Рис. 8.13. Чтение пикселов с помощью glReadPixels()

Значения для масштаба (scale), отклонения (bias), сдвига (shift) и смещения (offset) — те же, что и для отображения пикселов. Поэтому если вы попеременно читаете и отображаете пикселы, не забывайте настраивать их под чтение и рисование. Аналогично, предварительно должны быть инициализированы различные карты.

ПРИМЕЧАНИЕ

Не исключено, что при выполнении чтения и отображения яркость будет вычислена некорректно. Так, вклад в яркость компонентов R, G и B в общем случае неодинаковый, что видно из рис. 8.12 и 8.13. При необходимости вычислить значение яркости согласно числам стандарта 30 (R), 59 (G) и 11 (B) в процентах вы можете, установив параметр GL_RED_SCALE в 0,3, GL_RED_BIAS — в 0 и т. д. Тогда величина L будет получена по формуле 0,30R + 0,59G + 0,11B.

Советы по увеличению быстродействия вывода пикселов

Как вы могли увидеть, в OpenGL имеется богатый набор возможностей для чтения, рисования пикселов и других манипуляций с ними. Хотя эти возможности часто очень полезны, они могут также отрицательно повлиять на производительность. Поэтому мы хотим дать несколько рекомендаций с целью предупредить такие явления или просто для улучшения эффективности:

- Чтобы добиться максимальной скорости, установите все параметры передачи пикселов в значения по умолчанию, а масштабные коэффициенты — в (1.0, 1.0).
- При записи пикселов в буфер кадров к ним применяются операции над фрагментами. (См. раздел «Тесты и операции над фрагментами» главы 10.) Для оптимальной производительности следует отключить эти операции.
- Отключите другие ресурсоемкие действия, такие как текстурирование и освещение.
- Если вы воспользуетесь совместимым с буфером кадров форматом изображения, вы снимете с OpenGL дополнительную нагрузку. Например, если вы записываете изображения в RGB-буфер с 8-битовыми компонентами, установите при вызове `glDrawPixels()` аргумент *format* в значение RGB и аргумент *type* — в `UNSIGNED_BYTE`.
- В некоторых реализациях OpenGL беззнаковые форматы изображений быстрее знаковых форматов.
- Обычно быстрее рисуется один большой прямоугольник, чем несколько маленьких, поскольку издержки на передачу пикселов сглаживаются на фоне их количества.
- По возможности уменьшайте объем данных для копирования за счет менее емких типов (например, используйте `GL_UNSIGNED_BYTE`) и меньшего числа компонентов (например, используйте формат `GL_LUMINANCE_ALPHA`).
- Операции при передаче пикселов, будь то табличное преобразование, масштабирование, отклонение (bias), смещение (offset) или сдвиг, со значениями не по умолчанию, могут снизить производительность.

Расширение Imaging Subset

Расширение Imaging Subset (буквально — подмножество обработки изображений) — это совокупность программ, обеспечивающих дополнительные возможности процесса обработки изображений. С его помощью вы можете:

- использовать таблицы подстановки цветов для замещения значений пикселов;
- использовать свертки для фильтрации изображения;
- использовать матрицу преобразований цвета для преобразования цветового пространства и других линейных преобразований;

- собирать статистику для гистограмм и получать информацию о минимальном и максимальном значениях компонента цвета.

Imaging Subset будет вам полезно, если вы хотите иметь больше возможностей, чем предоставляют операции `glPixelTransfer*()` и `glPixelMap*()`.

Поскольку Imaging Subset является расширением, то для того, чтобы убедиться в его присутствии в своей реализации, нужно посмотреть на значение параметра `GL_ARB_imaging`. Если эта константа определена, вам доступны все новые возможности. Больше информации вы найдете в подразделе «Расширения стандарта» главы 14.

Функции подмножества доступны всякий раз, когда OpenGL читает или передает пикселы. К числу операций, на которые воздействуют функции Imaging Subset, относятся:

- отображение и чтение пиксельных прямоугольников: `glReadPixels()`, `glDrawPixels()`, `glCopyPixels()`;
- задание текстур: `glTexImage1D()`, `glTexImage2D()`, `glCopyTexImage*D()`, `glSubTexImage1D()`, `glSubTexImage2d()` и `glCopySubTexImage*D()`.

Рисунок 8.14 иллюстрирует операции, выполняемые Imaging Subset над пикселами, обрабатываемыми OpenGL. Практически любые возможности расширения могут быть отключены или включены, кроме матрицы преобразования цветов, доступной всегда.

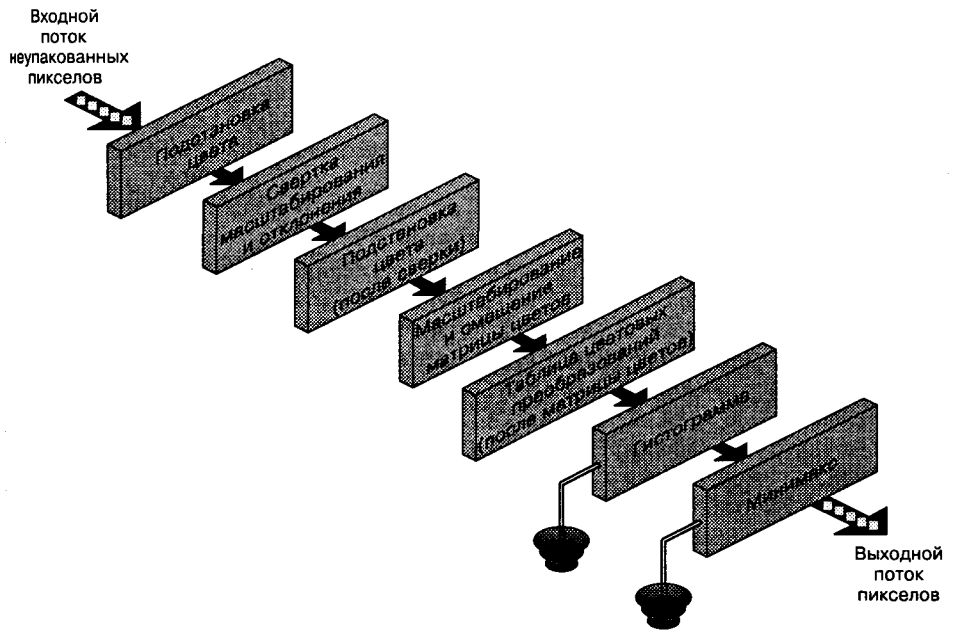


Рис. 8.14. Операции расширения Imaging Subset

Таблицы цветов

Таблицы цветов — это таблицы подстановки, предназначенные для замещения цвета пикселов. В приложениях таблицы цветов могут быть использованы для изменения контраста, фильтрации и исправления (equalization) изображений.

Доступны три различные таблицы подстановки, работающие на разных этапах конвейерной обработки. В табл. 8.7 показано, в каком месте конвейера может быть применена соответствующая таблица.

Таблица 8.7. Использование таблиц цветов в конвейерной обработке изображений

Параметр таблицы цветов	Операции над пикселами
GL_COLOR_TABLE	Место, где пикселы попадают в конвейер
GL_POST_CONVOLUTION_COLOR_TABLE	После свертки
GL_POST_COLOR_MATRIX_COLOR_TABLE	После преобразования матрицей цвета

Функция `glEnable()` делает доступной каждую из перечисленных в табл. 8.7 таблиц цветов, принимая значения из левого столбца в виде аргумента.

Задание таблиц цветов

Таблицы цветов задаются подобно одномерным изображениям. Как показано на рис. 8.14, есть три таблицы, доступные для модификации пикселов. Для указания на конкретную таблицу используется операция `glColorTable()`.

void `glColorTable`(GLenum *target*, GLenum *internalFormat*, GLsizei *width*, GLenum *format*, GLenum *type*, const GLvoid **data*);

Задает указанную таблицу цветов, в зависимости от того, чему равен аргумент *target* — GL_COLOR_TABLE, GL_POST_CONVOLUTION_COLOR_TABLE или GL_POST_COLOR_MATRIX_COLOR_TABLE. Если *target* установлен в GL_PROXY_COLOR_TABLE, GL_PROXY_POST_CONVOLUTION_COLOR_TABLE или GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE, команда `glColorTable()` проверяет, можно ли отнести указанную таблицу к доступным ресурсам или нет.

Аргумент *internalFormat* определяет представление OpenGL, в котором хранятся данные *(data)*. Это одно из следующих символических имен: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12 и GL_RGBA16.

Аргумент *width* должен быть степенью 2, указывает количество пикселов в таблице цветов. Аргументы *format* и *type* описывают формат и тип данных в таблице цветов. Их значения эквиваленты соответствующим параметрам `glDrawPixels()`.

Внутренний формат таблицы отвечает за то, какие компоненты будут замещены. Например, если выбран формат GL_RGB, это будут красный, зеленый и синий компоненты. Таблица 8.8 описывает соответствие между замещаемыми компонентами и внутренними форматами.

Таблица 8.8. Замещение пикселов по таблице цветов

Основной внутренний формат	Красный компонент	Зеленый компонент	Синий компонент	Альфа-компонент
GL_ALPHA	Не изменяется	Не изменяется	Не изменяется	A_t
GL_LUMINANCE	L_t	L_t	L_t	Не изменяется
GL_LUMINANCE_ALPHA	L_t	L_t	L_t	A_t
GL_INTENSITY	I_t	I_t	I_t	I_t
GL_RGB	R_t	G_t	B_t	Не изменяется
GL_RGBA	R_t	G_t	B_t	A_t

Здесь обозначение L_t представляет элементы яркости, оно имеет отношение только к красному, зеленому и синему компонентам. Обозначение I_t соответствует элементам интенсивности, то есть тем же RGB-компонентам плюс альфа-компоненту.

После применения таблицы цветов к изображению можно масштабировать или отклонить (bias) его пикселы, а далее их значения преобразуются к диапазону [0; 1]. Факторы GL_COLOR_TABLE_SCALE и GL_COLOR_TABLE_BIAS устанавливаются для каждой таблицы командой glColorTableParameter*().

void glColorTableParameter{if}v(GLenum *target*, GLenum *pname*, *TYPE* **param*);
Устанавливает параметры GL_COLOR_TABLE_SCALE и GL_COLOR_TABLE_BIAS для каждой таблицы цветов. Аргумент *target* — это одна из следующих констант: GL_COLOR_TABLE, GL_POST_CONVOLUTION_COLOR_TABLE или GL_POST_COLOR_MATRIX_COLOR_TABLE, он указывает саму таблицу цветов.
Возможные значения для *pname* — это GL_COLOR_TABLE_SCALE и GL_COLOR_TABLE_BIAS.
Значением аргумента *param* является массив из четырех значений, представляющих красный, зеленый, синий и альфа-модификаторы в этой последовательности.

Листинг 8.5 показывает, как с помощью таблицы цветов можно инвертировать изображение, то есть заменить каждый цветовой компонент инверсным ему.

Листинг 8.5. Замещение пикселов с помощью таблицы цветов: colortable.c

```
extern GLubyte* readImage(const char*, GLsizei*, GLsizei* );

GLubyte *pixels;
GLsizei width, height;

void init(void)
{
  int i;
  GLubyte colorTable[256][3];
```

продолжение

Листинг 8.5 (*продолжение*)

```
   pixels = readImage("Data/leeds.bin", &width, &height);
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glClearColor(0, 0, 0, 0);
/* Формирование таблицы инверсных цветов */
   for (i = 0; i < 256; ++i) {
      colorTable[i][0] = 255 - i;
      colorTable[i][1] = 255 - i;
      colorTable[i][2] = 255 - i;
   }
   glColorTable(GL_COLOR_TABLE, GL_RGB, 256, GL_RGB,
      GL_UNSIGNED_BYTE, colorTable);
   glEnable(GL_COLOR_TABLE);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1, 1);
   glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels);
   glFlush();
}
```

ПРИМЕЧАНИЕ

Листинг 8.5 включает в себя процедуру readImage(), введенную для упрощения структуры программы. В общем, вам требуется такого рода подпрограмма, умеющая читать изображение из файла. Формат файла, понимаемого readImage(), сводится к следующим принципам: ширина изображения (width) имеет тип GLsizei; высота (height) — тип GLsizei; при сохранении RGB-триад (количеством width × height) для каждого компонента используется тип GLubyte.

В дополнение к явному заданию таблицы цветов вы можете захотеть использовать изображения, созданные в буфере кадров, в качестве определения для таблицы цветов. Операция `glCopyColorTable()` позволяет указать отдельный ряд пикселов; с целью считывания его из буфера кадров для дальнейшего включения в таблицу цветов.

void `glCopyColorTable`(GLenum *target*, GLenum *internalFormat*, GLint *x*, GLint *y*, GLsizei *width*);

Создает таблицу цветов из данных буфера кадров, используя их как элементы таблицы цветов. Пикселы читаются из текущего буфера `GL_READ_BUFFER` и обрабатываются как при вызове `glCopyPixels()`, но без «паузы» перед конечным преобразованием. Здесь применяются настройки команды `glPixelTransfer*()`.

Аргумент *target* должен быть установлен на одну из таблиц командой `glColorTable()`. Для аргумента *internalFormat* используются те же символические константы, что для одноименного аргумента `glColorTable()`. Массив цветов задается строкой буфера кадров с шириной (*width*) и начальной позицией (*x*, *y*).

Замена всей таблицы цветов или ее части

Если нужно заменить часть таблицы цветов, используйте операции `glColorSubTable()` и `glCopyColorSubTable()`.

void glColorSubTable(GLenum *target*, GLsizei *start*, GLsizei *count*, GLenum *format*, GLenum *type*, const GLvoid **data*);

Заменяет элементы таблицы цветов, начиная с элемента *start* и заканчивая элементом *start* + *count* – 1, значениями из массива *data*.

Аргумент *target* — это GL_COLOR_TABLE, GL_POST_CONVOLUTION_COLOR_TABLE или GL_POST_COLOR_MATRIX_COLOR_TABLE. Аргументы *format* и *type* такие же, как для glColorTable(), но относятся к массиву *data*.

void glCopyColorSubTable(GLenum *target*, GLsizei *start*, GLint *x*, GLint *y*, GLsizei *count*);

Заменяет элементы таблицы цветов, начиная с элемента *start* и заканчивая элементом *start* + *count* – 1, где *count* — количество пикселов в строке буфера кадров, начинающейся с позиции (*x*, *y*). Пикселы преобразуются во внутренний формат (*internalFormat*) таблицы цветов.

Запрос значений из таблицы цветов

Пикселы, сохраненные в таблице цветов, могут быть извлечены с помощью команды glGetColorTable(). Подробно см. раздел «Команды запроса» приложения Б.

Запрос на допустимость таблицы цветов

Параметры-посредники (proxies) предоставляют способ, которым можно оценить, достаточно ли ресурсов в системе для сохранения таблицы цветов. Эти параметры заносятся в аргумент *target* команды glColorTable():

- GL_PROXY_COLOR_TABLE;
- GL_PROXY_POST_CONVOLUTION_COLOR_TABLE;
- GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE.

Тогда OpenGL определяет, хватит ли ресурсов системы для сохранения таблицы. Если она не помещается, значения ширины, формата и компонента разрешения устанавливаются в 0. Например:

```
glColorTable(GL_PROXY_COLOR_TABLE, GL_RGB, 1024, GL_RGB, GL_UNSIGNED_BYTE,
null);
glGetColorTableParameteriv(GL_PROXY_COLOR_TABLE, GL_COLOR_TABLE_WIDTH,
&width);
if (width == 0)
  /* нельзя поддержать такую таблицу */
```

Более подробно команда glGetColorTableParameter*() описана в разделе «Команды запроса» приложения Б.

Свертки

Свертки представляют собой фильтры, которые замещают каждый пиксел средневзвешенным значением его соседей и собственно самого этого пиксела. Размытие (blurring) и повышение резкости (sharpening) изображений, поиск границ и корректировка контраста — вот основные примеры применения сверток.

На рисунке 8.15 показано, как пиксел P_{00} и окружающие его пикселы обрабатываются фильтром 3 × 3, чтобы получить пиксел P'_{11}.

Свертки — это массивы весовых значений пикселов, и оперируют они только RGBA-пикселами. Фильтр, известный под названием kernel (ядро), является простым двухмерным массивом весовых значений. Каждый пиксел результирующего изображения создается умножением набора пикселов входного изображения на веса свертки, а результаты суммируются. Например, на рис. 8.15 в формировании пиксела P'_{11} принимают участие девять пикселов исходного изображения и девять элементов свертки.

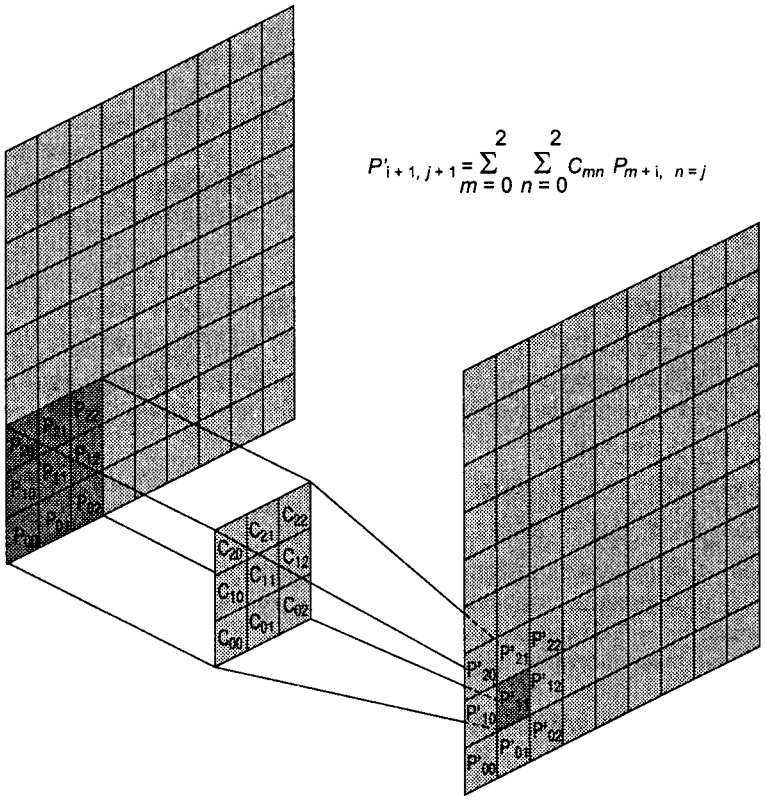


Рис. 8.15. Пример свертки

void `glConvolutionFilter2D`(GLenum *target*, GLenum *internalFormat*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **image*);

Определяет двухмерный фильтр свертки, где аргумент *target* установлен в значение `GL_CONVOLUTION_2D`.

Аргумент *internalFormat* определяет, над какими компонентами пикселов выполняется операция свертки, он может быть одной из 38 символических констант, перечисленных для одноименного аргумента функции `glColorTable()`.

Ширина (*width*) и высота (*height*) указывают размеры матрицы фильтра. Максимально допустимые ширина и высота могут быть оценены запросом `glGetConvolutionParameter*()`. (См. раздел «Команды запроса» приложения Б.) Как и для `glDrawPixels()`, формат (*format*) и тип (*type*) задают способ хранения изображения (*image*).

Подобно таблицам цветов, внутренний формат фильтра определяет, какие компоненты изображения подлежат обработке. Таблица 8.9 описывает связь форматов с компонентами пикселов. Обозначения R_s, G_s, B_s и A_s представляют компоненты цвета. Обозначение L_f соответствует значению яркости в фильтре `GL_LUMINANCE`, а I_f — значению интенсивности в фильтре `GL_INTENSITY`. Наконец, R_f, G_f, B_f и A_f представляют RGBA-компоненты фильтра.

Таблица 8.9. Воздействие свертки на RGBA-компоненты пикселов

Основной формат фильтра	Результат для красного компонента	Результат для зеленого компонента	Результат для синего компонента	Результат для альфа-компонента
GL_ALPHA	Не изменяется	Не изменяется	Не изменяется	$A_s \times A_f$
GL_LUMINANCE	$R_s \times L_f$	$G_s \times L_f$	$B_s \times L_f$	Не изменяется
GL_LUMINANCE_ALPHA	$R_s \times L_f$	$G_s \times L_f$	$B_s \times L_f$	$A_s \times A_f$
GL_INTENSITY	$R_s \times I_f$	$G_s \times I_f$	$B_s \times I_f$	$A_s \times I_f$
GL_RGB	$R_s \times R_f$	$G_s \times G_f$	$B_s \times B_f$	Не изменяется
GL_RGBA	$R_s \times R_f$	$G_s \times G_f$	$B_s \times B_f$	$A_s \times A_f$

Используйте команду `glEnable(GL_CONVOLUTION_2D)`, чтобы разрешить применение двухмерной свертки.

Листинг 8.6 демонстрирует использование нескольких 3 × 3 `GL_LUMINANCE` фильтров для обнаружения границ в RGB-изображении. Смена фильтра выполняется клавишами h, l и v.

Листинг 8.6. Использование двухмерного фильтра свертки: convolution.c

```
extern GLubyte* readImage(const char*, GLsizei*, GLsizei*);

GLubyte *pixels;
GLsizei width, height;

/* Описание фильтра свертки */
GLfloat horizontal[3][3] = {
  { 0, -1, 0 },
  { 0, 1, 0 },
  { 0, 0, 0 }
};

GLfloat vertical[3][3] = {
  { 0, 0, 0 },
  { -1, 1, 0 },
  { 0, 0, 0 }
};
```

продолжение

Листинг 8.6 (*продолжение*)

```
GLfloat laplacian[3][3] = {
   { -0.125, -0.125, -0.125 },
   { -0.125, 1.0, -0.125 },
   { -0.125, -0.125, -0.125 }
};

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1, 1);
   glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels);
   glFlush();
}

void init(void)
{
   pixels = readImage("Data/leeds.bin", &width, &height);
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glClearColor(0.0, 0.0, 0.0, 0.0);

   printf("Горизонтальный фильтр\n");
   glConvolutionFilter2D(GL_CONVOLUTION_2D, GL_LUMINANCE, 3, 3,
      GL_LUMINANCE, GL_FLOAT, horizontal);
   glEnable(GL_CONVOLUTION_2D);
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 'h':
         printf("Горизонтальный фильтр\n");
         glConvolutionFilter2D(GL_CONVOLUTION_2D, GL_LUMINANCE,
            3, 3, GL_LUMINANCE, GL_FLOAT, horizontal );
         break;
      case 'v':
         printf("Вертикальный фильтр\n" );
         glConvolutionFilter2D( GL_CONVOLUTION_2D, GL_LUMINANCE,
            3, 3, GL_LUMINANCE, GL_FLOAT, vertical );
         break;
      case 'l':
         printf("Фильтр Лапласа\n");
         glConvolutionFilter2D(GL_CONVOLUTION_2D, GL_LUMINANCE,
            3, 3, GL_LUMINANCE, GL_FLOAT, laplacian );
         break;
      case 27: /* Клавиша Esc */
         exit(0);
         break;
   }
   glutPostRedisplay();
}
```

Как и в таблицах цветов, вы можете задать фильтр свертки со значениями пикселов из буфера кадров. Команда `glCopyConvolutionFilter2D()` копирует прямо-

угольник пикселов из текущего буфера GL_READ_BUFFER для последующего их использования в фильтре свертки. Если GL_LUMINANCE или GL_INTENSITY заданы для внутреннего формата (*internalFormat*), для задания значения фильтра свертки используется красный компонент пиксела.

void glCopyConvolutionFilter2D(GLenum *target*, GLenum *internalFormat*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

Определяет двухмерный фильтр свертки, инициализируемый пикселами из буфера кадров цвета. Аргумент *target* должен быть установлен в GL_CONVOLUTION_2D, а в аргументе *internalFormat* должен быть указан один из внутренних форматов glConvolutionFilter2D().

Прямоугольник пикселов с левым нижним углом (*x*, *y*), шириной *width* и высотой *height* читается из буфера кадров и преобразуется в указанный внутренний формат.

Задание разъемных двухмерных фильтров свертки

Фильтр свертки является *разъемным* (separable), если он может быть представлен *векторным произведением* (outer product) двух одномерных фильтров.

Команда glSeparableFilter2D() используется для задания двух одномерных фильтров, представляющих разъемный двухмерный фильтр свертки. Как и для glConvolutionFilter2D(), внутренний формат фильтра свертки определяет то, как обрабатываются фильтры изображения.

void glSeparableFilter2D(GLenum *target*, GLenum *internalFormat*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **row*, const GLvoid **column*);

Описывает двухмерный разъемный фильтр свертки. Аргумент *target* должен быть установлен в GL_SEPARABLE_2D. Внутренний формат (*internalFormat*) принимает те же значения, что и для glConvolutionFilter2D().

Ширина (*width*) задает количество пикселов в массиве строки (*row*). Аналогично, высота (*height*) задает количество пикселов массива столбца (*column*). Аргументы типа (*type*) и формата (*format*) описывают способ хранения строки и столбца так же, как и для glConvolutionFilter2D().

Используйте glEnable(GL_SEPARABLE_2D), чтобы разрешить применение сверток, использующих разъемный двухмерный фильтр свертки. Если заданы одновременно GL_CONVOLUTION_2D и GL_SEPARABLE_2D, параметр GL_CONVOLUTION_2D имеет больший приоритет.

Например, вы можете построить 3 × 3 GL_LUMINANCE разъемный фильтр свертки, задав одномерные фильтры для строки и столбца. Тогда полный двухмерный фильтр был бы вычислен как векторное произведение:

$$\begin{bmatrix} -1/2 \\ 1 \\ -1/2 \end{bmatrix} \times \begin{bmatrix} -1/2 & 1 & -1/2 \end{bmatrix} = \begin{bmatrix} 1/4 & -1/2 & 1/4 \\ -1/2 & 1 & -1/2 \\ 1/4 & -1/2 & 1/4 \end{bmatrix}.$$

Разъемные двухмерные фильтры по отношению к затратам на вычисления более эффективны, чем обычные двухмерные фильтры свертки.

Одномерные фильтры свертки

Одномерные фильтры свертки идентичны двухмерной версии за исключением того, что параметр высоты фильтра предположительно равен единице. Однако они влияют только на спецификацию одномерных текстур (см. раздел «Одномерные текстуры» главы 9.)

void `glConvolutionFilter1D`(GLenum *target*, GLenum *internalFormat*, GLsizei *width*, GLenum *format*, GLenum *type*, const GLvoid **image*);

Задает одномерный фильтр свертки. Аргумент *target* должен быть установлен в `GL_CONVOLUTION_1D`. Ширина (*width*) задает количество пикселов фильтра. Аргументы *internalFormat*, *format* и *type* принимают те же значения, его и для `glConvolutionFilter2D()`. Массив *image* указывает на одномерное изображение, используемое в качестве фильтра свертки.

Используйте `glEnable(GL_CONVOLUTION_1D)`, чтобы разрешить применение одномерных сверток.

Вы можете захотеть использовать для фильтра свертки значения, сгенерированные из буфера кадров. Команда `glCopyConvolutionFilter1D()` копирует строку пикселов из текущего буфера `GL_READ_BUFFER` и преобразует ее во внутренний формат *(internalFormat)* для дальнейшего применения в фильтре свертки.

void `glCopyConvolutionFilter1D`(GLenum *target*, GLenum *internalFormat*, GLint *x*, GLint *y*, GLsizei *width*);

Описывает одномерный фильтр свертки со значениями пикселов, взятыми из буфера кадров. Команда `glCopyConvolutionFilter1D()` копирует *width* пикселов, начиная с позиции (*x*, *y*), и преобразует их в указанный внутренний формат.

К заданному фильтру свертки могут быть применены масштабирование и отклонение (bias). Значения масштаба и отклонения задаются командой `glConvolutionParameter*()`. Причем никакой фиксации (clamping) значений фильтра свертки после этого не делается.

void `glConvolutionParameter`{if}(GLenum *target*, GLenum *pname*, *TYPE param*);

void `glConvolutionParameter`{if}v(GLenum *target*, GLenum *pname*, const *TYPE* **params*);

Устанавливает параметры, управляющие выполнением операции свертки. Аргумент *target* должен быть задан одной из следующих констант: `GL_CONVOLUTION_1D`, `GL_CONVOLUTION_2D` или `GL_SEPARABLE_2D`.

Имя параметра (*pname*) должно быть указано как `GL_CONVOLUTION_BORDER_MODE`, `GL_CONVOLTION_FILTER_SCALE` или `GL_CONVOLUTION_FILTER_BIAS`. Когда для *pname* используется параметр `GL_CONVOLUTION_BORDER_MODE`, подразумевается режим обработки границ свертки. В таком случае *params* устанавливается в `GL_REDUCE`, `GL_CONSTANT_BORDER` или `GL_REPLICATE_BORDER`. Если *pname* задан как `GL_CONVOLUTION_FILTER_SCALE` или `GL_CONVOLUTION_FILTER_BIAS`, массив *params* должен содержать четыре значения цвета для красного, зеленого, синего и альфа-компонентов в этой последовательности.

Особенности применения свертки на границах

Операция свертки в отношении границ изображения отличается от обработки пикселов, лежащих во внутренней его области. Для управления применением свертки на границах предусмотрено три режима:

- GL_REDUCE приводит к сжатию (shrink) в каждом измерении согласно размерам фильтра свертки. Ширина результирующего изображения изменяется как ($width - W_f$), а высота — как ($height - H_f$), где W_f и H_f — это ширина и высота фильтра свертки. Если конечное изображение вырождается до нулевого или отрицательного размера, оно не вырабатывается, причем без сообщения об ошибке.
- GL_CONSTANT_BORDER — для обработки краев изображения, для выходящих за его границы пикселов, используются константные значения, задаваемые командой `glConvolutionParameter*()`. В этом случае размеры изображения не меняются.
- GL_REPLICATE_BORDER — свертка выполняется аналогично режиму GL_CONSTANT_BORDER, с той разницей, что к вычислениям привлекаются значения крайних пикселов ряда или столбца. Здесь размеры изображения также не меняются.

Операции, дополняющие свертку

После завершения процесса свертки к пикселам результирующего изображения могут быть применены масштабирование и смещение (bias), а также преобразование к диапазону [0, 1]. Соответствующие значения указываются вызовом `glPixelTransfer*()`: GL_POST_CONVOLUTION_*_SCALE для масштабирования GL_POST_CONVOLUTION_*_BIAS для отклонения. Также можно изменить цвета пикселов по таблице подстановки, задав режим GL_POST_CONVOLUTION_COLOR_TABLE командой `glColorTable()`.

Матрица цвета

С целью поддержки преобразований цветового пространства (перехода к другой цветовой модели) или линейных преобразований значений пикселов, в расширении Imaging Subset предусмотрен стек матриц с размерами 4 × 4, который начинает работать при вызове `glMatrixMode(GL_COLOR)`. Например, чтобы перейти от модели RGB к CMY (голубой, фиолетовый, желтый), вы можете сделать следующее:

```
GLfloat rgb2cmy[16] = {
  -1, 0, 0, 0,
  0, -1, 0, 0,
  0, 0, -1, 0,
  1, 1, 1, 1
};
glMatrixMode(GL_COLOR); /* активация режима матрицы цвета */
glLoadMatrixf(rgb2cmy);
glMatrixMode(GL_MODELVIEW); /* возврат в режим макетной матрицы */
```

ПРИМЕЧАНИЕ

OpenGL хранит матрицы по столбцам. См. раздел «Команды преобразования общего назначения» главы 3 для получения более подробной информации об использовании матриц в OpenGL.

В стек матриц цвета заносятся как минимум две матрицы. (См. раздел «Команды запроса» приожения Б.) В отличие от других операций Imaging Subset, преобразование по матрице цвета выполняется всегда и не может быть блокировано.

Листинг 8.7 иллюстрирует использование матрицы цвета для обмена позициями красного и зеленого компонентов цвета.

Листинг 8.7. Обмен компонентов цвета с использованием матрицы цвета: colormatrix.c

```
extern GLubyte* readImage(const char*, GLsizei*, GLsizei*);

GLubyte *pixels;
GLsizei width, height;

void init(void)
{
   /* Задание матрицы цвета для переупорядочивания компонентов
    * из модели RGB в GBR */
   GLfloat m[16] = {
      0.0, 1.0, 0.0, 0.0,
      0.0, 0.0, 1.0, 0.0,
      1.0, 0.0, 0.0, 0.0,
      0.0, 0.0, 0.0, 1.0
   };
   pixels = readImage("Data/leeds.bin", &width, &height);
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glMatrixMode(GL_COLOR);
   glLoadMatrixf(m);
   glMatrixMode(GL_MODELVIEW);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1, 1);
   glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels);
   glFlush();
}
```

Операции, дополняющие преобразование по матрице цвета

Подобно пост-операциям свертки, к пикселам можно применить масштабирование и отклонение уже после срабатывания матрицы цвета. Вызов `glPixelTransfer*()` с параметром `GL_POST_COLOR_MATRIX_*_SCALE` или `GL_POST_`

COLOR_MATRIX_*_BIAS задает соответствующие коэффициенты масштабирования и отклонения. По выполнении пост-операций значения пикселов сводятся к диапазону [0, 1].

Гистограмма

С помощью расширения Imaging Subset вы можете собирать статистическую информацию об изображении. Гистограмма демонстрирует распределение значений цвета в изображении, знания о котором вам пригодятся, например, при корректировке контраста.

Команда `glHistogram()` задает, какие компоненты изображения вы хотите видеть на гистограмме и достаточно ли вам просмотреть статистику или же требуется продолжить обработку изображения. Для того чтобы разрешить сбор статистики, вы должны вызвать команду `glEnable(GL_HISTOGRAM)`.

Подобно цветовым палитрам (см. раздел «Таблицы цветов»), в вашем распоряжении посреднический механизм для определения наличия необходимых ресурсов системы. Если ресурсов недостаточно, ширина, формат и разрешение гистограммы устанавливаются в нуль. Воспользоваться прокси-механизмом можно с помощью команды `glGetHistogramParameter()`, описанной в разделе «Команды запросов» приложения Б.

void `glHistogram`(GLenum *target*, GLsizei *width*, GLenum *internalFormat*, GLboolean *sink*);

Определяет, в каком виде будут сохранены данные гистограммы. Аргумент *target* должен быть задан как `GL_HISTOGRAM` или `GL_PROXY_HISTOGRAM`. Ширина (*width*) указывает количество элементов в таблице гистограммы. Это значение должно быть степенью 2.

Внутренний формат (*internalFormat*) отвечает за то, как будут сохранены данные гистограммы. Допустимые значения: `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_RGB`, `GL_RGB2`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12` и `GL_RGBA16`. Этот список не включает в себя значения `GL_INTENSITY*`. Этим он отличается от списка команды `glColorTable()`.

Аргумент *sink* задает, следует ли обрабатывать пикселы на минимаксном этапе конвейерной обработки.

После прохождения пикселов по конвейеру через операцию `glDrawPixels()` вы можете извлечь результаты формирования гистограммы с помощью вызова `glGetHistogram()`. В дополнение к возврату самих значений, команда `glGetHistogram()` может быть использована для внутренней памяти гистограммы. Она же может быть сброшена напрямую, через вызов `glResetHistogram()`, как описывается парой страниц ниже.

void glGetHistogram(GLenum *target*, GLboolean *reset*, GLenum *format*, GLenum *type*, GLvoid **values*);

Возвращает собранную для гистограммы статистику. Аргумент *target* задается как GL_HISTOGRAM. Аргумент *reset* отвечает за очистку таблиц (внутренней памяти) гистограммы.

Формат (*format*) и тип (*type*) задают способ сохранения массива значений (*values*) и то, как данные гистограммы будут переданы приложению. Они принимают те же значения, что и их аналоги для команды glDrawPixels().

Программа из листинга 8.8 подготавливает гистограмму изображения и рисует результирующее распределение в окне. В этом примере клавиша s служит для управления параметром *sink*, отвечающим за допуск пикселов к последующим этапам конвейерной обработки.

Листинг 8.8. Подготовка и рисование гистограммы: histogram.c

```
#define HISTOGRAM_SIZE 256 /* Должно быть степенью 2 */
extern GLubyte* readImage(const char*, GLsizei*, GLsizei*);

GLubyte *pixels;
GLsizei width, height;

void init(void)
{
   pixels = readImage("Data/leeds.bin", &width, &height);
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glClearColor(0.0, 0.0, 0.0, 0.0);

   glHistogram(GL_HISTOGRAM, HISTOGRAM_SIZE, GL_RGB, GL_FALSE);
   glEnable(GL_HISTOGRAM);
}

void display(void)
{
   int i;
   GLushort values[HISTOGRAM_SIZE][3];
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1, 1);
   glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels);
   glGetHistogram(GL_HISTOGRAM, GL_TRUE, GL_RGB, GL_UNSIGNED_SHORT, values);

/* Рисование гистограммы */

   glBegin(GL_LINE_STRIP);
   glColor3f(1.0, 0.0, 0.0);
   for (i = 0; i < HISTOGRAM_SIZE; i++)
      glVertex2s(i, values[i][0]);
   glEnd();

   glBegin(GL_LINE_STRIP);
   glColor3f(0.0, 1.0, 0.0);
   for (i = 0; i < HISTOGRAM_SIZE; i++)
      glVertex2s(i, values[i][1]);
```

```
   glEnd();

   glBegin(GL_LINE_STRIP);
   glColor3f(0.0, 0.0, 1.0);
   for (i = 0; i < HISTOGRAM_SIZE; i++)
      glVertex2s(i, values[i][2]);
   glEnd();

   glFlush();
}

void keyboard(unsigned char key, int x, int y)
{
   static GLboolean sink = GL_FALSE;

   switch (key) {
      case 's':
         sink = !sink;
         glHistogram(GL_HISTOGRAM, HISTOGRAM_SIZE, GL_RGB, sink);
         break;
      case 27: /* Клавиша Esc */
         exit(0);
         break;
   }
   glutPostRedisplay();
}
```

Команда `glResetHistogram()` сбрасывает гистограмму без извлечения ее данных.

> void `glResetHistogram`(GLenum *target*);
> Сбрасывает счетчики гистограммы в нуль. Аргумент *target* задается как `GL_HISTOGRAM`.

Метод минимакса

Функция `glMinmax()` находит экстремальные (минимальное и максимальное) значения компонентов пиксела. Как и в случае `glHistogram()`, после ее выполнения вы можете продолжить обработку пикселов или отказаться от дальнейших действий.

> void `glMinmax`(GLenum *target*, GLenum *internalFormat*, GLboolean *sink*);
> Определяет минимальное и максимальное значения компонентов пикселов в изображении. Аргумент *target* должен быть установлен в `GL_MINMAX`.
> Внутренний формат (*internalFormat*) задает, какие компоненты цвета привлекаются к обработке. Значения этого аргумента те же, что и для `glHistogram()`.
> Если аргумент *sink* задан как `GL_TRUE`, пикселы не записываются в буфер кадров, а если как `GL_FALSE` — выполняется построение изображения.

Команда `glGetMinmax()` используется для получения найденных экстремальных значений. Подобно `glHistogram()`, обращение к внутренним данным может быть объединено с их сбросом.

void glGetMinmax(GLenum *target*, GLboolean *reset*, GLenum *format*, GLenum *type*, GLvoid **values*);

Возвращает результаты операции метода минимакса. Аргумент *target* должен быть установлен в GL_MINMAX. Если аргумент *reset* задан как GL_TRUE, выполняется также сброс минимального и максимального значений. Формат (*format*) и тип (*type*) определяют способ помещения выходных данных в массив *values*, эти аргументы принимают те же значения, что и для glDrawPixels().

Листинг 8.9 демонстрирует использование glMinmax() для случая формата RGB. Чтобы разрешить применение операции минимакса, вы должны вызвать команду glEnable(GL_MINMAX).

Минимальное и максимальное значения помещаются в массив *values* и группируются по компонентам. Например, при задании константы GL_RGB для формата, первые три значения массива представляют минимальные величины красного, зеленого и синего компонентов, а следующие три — их максимальные значения.

Листинг 8.9. Нахождение минимального и максимального значений пикселов: minmax.c

```
extern GLubyte* readImage(const char*, GLsizei*, GLsizei*);
GLubyte *pixels;
GLsizei width, height;
void init(void)
{
   pixels = readImage("Data/leeds.bin", &width, &height);
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glClearColor(0.0, 0.0, 0.0, 0.0);

   glMinmax(GL_MINMAX, GL_RGB, GL_FALSE);
   glEnable(GL_MINMAX);
}
void display(void)
{
   GLubyte values[6];
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1, 1);
   glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels);
   glGetMinmax(GL_MINMAX, GL_TRUE, GL_RGB, GL_UNSIGNED_BYTE, values);
   glFlush();
   printf("Красный: мин = %d макс = %d\n", values[0], values[3]);
   printf("Зеленый: мин = %d макс = %d\n", values[1], values[4]);
   printf("Синий: мин = %d макс = %d\n", values[2], values[5]);
}
```

Команда glGetMinmax() после извлечения результатов может сбрасывать буфер значений, но вы можете сделать это и явно вызовом glResetMinmax().

void glResetMinmax(GLenum *target*);

Сбрасывает минимальное и максимальное значения к их значениям инициализации. Аргумент *target* задается равным GL_MINMAX.

9

Наложение текстур

После прочтения этой главы вы научитесь:

- Понимать, что наложение текстур в состоянии привнести в вашу сцену
- Задавать изображения текстур со сжатием и без сжатия
- Управлять фильтрацией изображения текстуры при его применении к фрагменту
- Создавать изображения текстур в объектах текстур и управлять ими, а также, если это возможно, управлять высокопроизводительным набором этих объектов текстур
- Устанавливать, как значения цвета в изображении будут комбинироваться с фрагментами, к которым изображение применяется
- Дополнять изображение координатами текстуры, показывающими, как изображение текстуры должно быть выровнено относительно объектов сцены
- Автоматически генерировать координаты текстуры для получения таких эффектов, как контуры предметов и структура природных объектов
- Выполнять сложные операции над текстурами за один проход посредством мультитекстурирования (последовательности модулей текстур)
- Использовать комбинированные функции текстур для математического оперирования текстурой, фрагментом и значениями цвета
- Обрабатывать фрагменты после текстурирования с помощью вторичных цветов
- Выполнять матричные преобразования координат текстур
- Отображать тонированные объекты, основываясь на глубине

До настоящего момента мы либо однотонно закрашивали каждый геометрический примитив, либо плавно изменяли цвет от вершины к вершине — то есть все примитивы были нарисованы без применения текстур. Но если вам нужно нарисовать, например, большую кирпичную стену, не прибегая к текстурам, каждый кирпич потребует собственный многоугольник. Без текстурирования большая плоская стена — сама являющаяся прямоугольником — может потребовать тысячи отдельных кирпичей, которые даже тогда могут производить впечатление чересчур гладких и регулярных вместо того, чтобы давать реалистичную картинку.

Наложение текстур позволяет вам вписать изображение кирпича кладки (полученное, скажем, сканированием фотографии реальной стены) в многоугольник и нарисовать полную сцену как один многоугольник. Наложение текстур гарантирует, что при трансформировании и воспроизведении многоугольников не возникнут какие-либо искажения. Например, когда стена уходит в перспективу, кирпичи должны выглядеть меньше своего действительного размера при удалении от точки наблюдения. Другие применения наложения текстур — отображение растительного мира в больших многоугольниках для представления земной поверхности в имитаторах полета; шаблоны рисунков для рабочего стола; текстуры, придающие многоугольникам вид природных материалов, таких как мрамор, дерево и тканевое полотно. Перечислить все невозможно. Хотя естественнее думать о применении текстур в отношении многоугольников, текстуры могут быть применены к любым примитивам — точкам, линиям, битовым образам и изображениям. Цветные рисунки 6, 8, 18–21 и 24–32 демонстрируют использование текстур.

По причине такого большого числа областей применения, наложение текстур — довольно большая, сложная тема, и вы должны выбрать для себя из этой темы что-то более конкретное. Большинство людей интуитивно понимают двухмерные текстуры, но ведь текстуры могут быть одномерными и даже трехмерными. Вы можете накладывать текстуры на поверхности, собранные из набора многоугольников, или на искривленные поверхности, и вы можете тиражировать текстуру в одном, двух или более направлениях (в зависимости от того, сколько измерений для нее подразумевается), распространяя ее по поверхности. Кроме того, вы можете автоматически отображать текстуру в объект так, чтобы выделить контуры или другие свойства просматриваемого элемента. Зеркальные объекты можно текстурировать так, чтобы они, будучи в центре помещения или другой планировки, отражали окружение на своих поверхностях. Наконец, текстура может быть применена к поверхности различными способами. Это может быть непосредственная окраска (подобная переводной картинке), модуляция цвета поверхности в другом случае или смешивание цвета текстуры с цветом поверхности. Если вы раньше не сталкивались с наложением текстур, вам может показаться, что мы углубляемся в предмет по ходу этой главы быстрее, чем следовало бы. Тогда в качестве дополнительного источника информации советуем просмотреть главу о наложении текстур в книге *3D Computer Graphics* Алана Уатта (Alan Watt) (Addison–Wesley, 1999).

Текстуры — это просто прямоугольные массивы данных — цвета, яркости или цвета и альфа-компонент. Отдельные значения в массиве текстуры часто называются *текселами* (texels). Становятся сложными текстуры тогда, когда применяются к непрямоугольным областям, и во избежание излишней сложности это должно быть сделано разумным способом.

Рисунок 9.1 иллюстрирует процесс наложения текстуры. Фигура слева представляет собой полную текстуру, где углы черного четырехугольного контура обозначают место, куда накладывается текстура. Когда четырехугольник отображается на экране, он может быть деформирован посредством различных преобразований — вращения, смещения, масштабирования и проецирования. Справа показана фигура в том виде, как она может выглядеть на экране после выполнения таких преобразований. (Заметьте, что этот четырехугольник вогнутый и может быть воспроизведен OpenGL некорректно, если не сделать предварительное разбиение, или тесселяцию. См. главу 11 для получения большей информации о разбиении на многоугольники.)

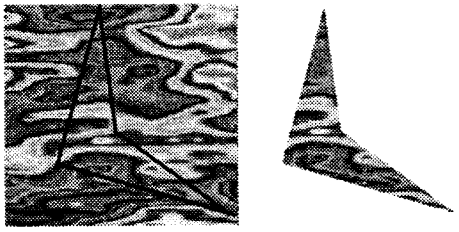


Рис. 9.1. Процесс наложения текстур

Обратите внимание на то, как деформируется текстура, чтобы соответствовать четырехугольнику. В этом случае она растянута в x-направлении и сжата по y; чуть-чуть повернута и деформирована сдвигом. В зависимости от размера текстуры, деформации четырехугольника и размеров изображения на экране некоторые текселы могут быть отображены в более чем один фрагмент, а некоторые фрагменты могут покрывать несколько текселов. Так как текстура составлена из дискретных элементов (в нашем случае таких 256 × 256), при отображении текселов на фрагменты должна быть применена фильтрация. Например, если одному фрагменту соответствует множество текселов, их следует усреднить до нужного размера; если границы тексела пересекают границы фрагмента, для надлежащих текселов определяется средневзвешенное значение. Из-за этих вычислений текстурирование требует вычислительного ресурса, поэтому во многих специализированных графических системах имеются средства для его аппаратной поддержки.

Приложение может создавать объекты тестур, каждый из которых представляет отдельную текстуру (и, возможно, связанные множественные отображения, или MIP-текстуры). Некоторые реализации OpenGL поддерживают специальный *рабочий набор* (working set) объектов текстур, обеспечивающий лучшую производительность по сравнению с внешними по отношению к набору текстурами. Эти высокопроизводительные объекты текстур являются резидентными, а специаль-

ное аппаратное или программное обеспечение предназначено для ускорения их обработки. С помощью OpenGL вы можете создавать или удалять объекты текстур и составлять из интересующих вас рабочий набор.

Эта глава описывает средства OpenGL для наложения текстур и разбита на следующие основные разделы:

- **Общий обзор и пример.** Дает краткое, общего уровня обозрение действий, выполняемых при наложении текстур, включая относительно простой пример.
- **Описание текстуры.** Поясняет, как специфицируются одномерные, двухмерные или трехмерные текстуры. Здесь также рассказано, как использовать границы текстур, как сделать серию связанных текстур различных размеров и как управлять методами фильтрации, отвечающими за проецирование текстуры в координаты экрана.
- **Фильтрация.** Рассматривает увеличение и уменьшение текстур при их применении к пикселам многоугольников. Уменьшение (minification) связано со специальными MIP-текстурами, о которых здесь также говорится.
- **Объекты текстур.** Описывает помещение изображений текстур в объекты так, что вы можете управлять несколькими текстурами одним действием. С помощью объектов текстур вы можете создать высокопроизводительный набор резидентных объектов текстур. Вы можете также расставить приоритеты для объектов текстур, регулируя тем самым способность к нахождению текстур в резидентном состоянии.
- **Функции текстур.** Повествует о методах, используемых для окрашивания поверхностей текстурами. Вы можете заменить другим оригинальный цвет объекта или получить результирующий цвет смешиванием цвета объекта и текстуры.
- **Присвоение координат текстур.** Описывает то, как вычислить и присвоить соответствующие координаты текстуры вершинам объекта. Здесь также говорится об управлении поведением координат, лежащих вне диапазона по умолчанию — то есть о том, как распределить текстуры по поверхности.
- **Автоматическая генерация координат текстур.** Показывает, как OpenGL автоматически генерирует координаты текстур, которые позволяют добиться таких эффектов, как отображение контуров предметов и структуры природных объектов.
- **Мультитекстурирование.** Подробно описывает процесс применения текстур в нескольких конвейерах, последовательно осуществляющих текстурирование.
- **Комбинирование функций текстур.** Объясняет, как вы можете управлять математическими операциями (умножение, сложение, вычитание, интерполяция и даже скалярное произведение (dot product)) над RGB- и альфа-значениями текстур, постоянными цветами и входящими фрагментами. Комбинирование функций предоставляет возможность гибкой, программно управляемой обработки фрагментов.
- **Применение вторичного цвета после текстурирования.** Показывает, как применить вторичные цвета к фрагментам, прошедшим текстурирование.

- **Стек матриц текстур.** Описывает способы управления стеком матриц текстур и использование q-координаты.

В версии OpenGL 1.1 было представлено несколько операций наложения текстур:

- дополнительные форматы внутреннего представления изображений текстур;
- текстуры-посредники, для оценки возможности выделения ресурсов для изображений текстур;
- текстуры в составных частях изображения, позволяющие заменить как все изображение текстуры, так и его части, вместо удаления (создания) текстуры, дающего тот же эффект;
- задание данных текстуры из памяти буфера кадров (а также из системной памяти);
- объекты текстур, включая резидентные текстуры и расстановку приоритетов.

Версия 1.2 к ним добавила:

- трехмерные изображения текстур;
- новый режим управления координатами текстуры, `GL_CLAMP_TO_EDGE`, вырабатывающий текселы из крайних пикселов изображения текстуры, но не из его границ;
- усиленный контроль над текстурами при использовании множественного отображения для представления различных уровней детализации (levels of detail, LOD);
- вычисление отраженного света после операций текстурирования.

Версия 1.3 подарила еще больше новых операций:

- сжатие текстур;
- текстуры, представленные кубами (cube map);
- мультитекстурирование, когда несколько текстур используется при построении одного примитива;
- режим `GL_CLAMP_TO_BORDER`;
- режимы наложения окружения: `GL_ADD` и `GL_COMBINE` (включая скалярное произведение).

Версия 1.4 предоставила в ваше распоряжение следующие возможности:

- режим `GL_MIRRORED_REPEAT`;
- автоматическая генерация MIP-текстур с помощью `GL_GENERATE_MIPMAP`;
- параметр `GL_TEXTURE_LOD_BIAS`, предназначенный для изменения уровня детализации текстур;
- применение вторичного цвета (указываемого `glSecondaryColor*()`) после текстурирования;
- возможность использовать цвета из различных модулей текстур в качестве исходных, в процессе комбинирования текстур;

- использование глубины (r-координата) в составе внутреннего компонента формата текстуры и режимов текстурирования, сравнивающих глубину текселов для принятия решения об отображении текстур.

Если вы попытались использовать одну из этих операций и не нашли ее, проверьте номер версии своей реализации OpenGL. (См. раздел «Какую версию я использую?» главы 14). В некоторых реализациях определенная функция может быть доступна только через расширение.

Например, в версии OpenGL 1.2 мультитекстурирование одобрено ARB (Architecture Review Board, совет, рецензирующий архитектуру OpenGL) как необязательное дополнение. В реализации 1.2 поддержанные функции мультитекстурирования и названия констант имеют суффикс `ARB`, например `glActiveTextureARB(GL_TEXTURE1_ARB)`. В OpenGL 1.3 мультитекстурирование стало неотъемлемым элементом, и суффикс `ARB` был удален.

Общий обзор и пример

Этот раздел содержит краткий обзор действий, необходимых для наложения текстур. Здесь также представлен пример относительно простой программы, выполняющей наложение текстур. Конечно, вы должны понимать, что наложение текстур может быть очень сложным процессом.

Действия при наложении текстур

Чтобы использовать наложение текстур, вы должны предпринять следующие действия:

1. Создать объект текстуры и определить текстуру для этого объекта.
2. Указать, как текстура будет применена к каждому пикселу.
3. Включить режим наложения текстур.
4. Отрисовать сцену с использованием как координат текстуры, так и геометрических координат.

Имейте в виду, что наложение текстур работает только в режиме RGBA. Его результаты в режиме индексации цвета непредсказуемы.

Создание объекта текстуры и определение текстуры

Под текстурой обычно понимается нечто двумерное, как и большинство изображений, но она может быть также одномерной или трехмерной. Данные, описывающие текстуру, могут состоять из одного, двух, трех или четырех элементов на тексел и представляться четверкой (R, G, B, A), постоянной модуляции и компонентом глубины.

В листинге 9.1, который очень прост, всего один объект текстуры создан для поддержания одной несжатой двухмерной текстуры. Эта программа не пытается узнать, сколько доступно памяти. Поскольку создается единственная текстура, не

предпринимаются действия по определению приоритетов или иному управлению рабочим набором объектов текстур. Другие улучшения технологии, такие как обработка границ текстур, текстуры с различными уровнями детализации или «кубические» текстуры, не используются в этом простом примере.

Указание, как применить текстуру к каждому пикселу

В вашем распоряжении одна из четырех доступных функций для вычисления конечного RGBA-значения, исходя из цвета фрагмента и данных изображения текстуры. Одна возможность — это просто использование цвета текстуры в качестве результирующего; это режим *замещения*, в котором текстура окрашивает собой фрагмент, как будто на него наклеивается переводная картинка. (Режим замещения использован в листинге 9.1.) Другой способ — использовать текстуру для *модуляции* (modulate), или масштабирования, цвета фрагмента; эта техника полезна для комбинирования эффекта текстурирования с освещением. Наконец, постоянный цвет может быть смешан с цветом фрагмента на основе значений текстуры.

Включение наложения текстур

Вам нужно активировать текстурирование перед рисованием своей сцены. Текстурирование включается и выключается командами `glEnable()` и `glDisable()`, с параметрами `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` или `GL_TEXTURE_CUBE_MAP` для одномерных, двухмерных, трехмерных или кубических текстур соответственно.

ПРИМЕЧАНИЕ

Если активны режимы в двух или во всех трех измерениях, то используется определенный для старшей размерности. Если используются кубические текстуры, они обладают наивысшим приоритетом. Это ваше дело, но ради чистоты программы лучше указывать только действительно задействуемый режим.

Воспроизведение сцены с учетом координат текстуры и геометрических координат

Вам следует указать, как текстура должна быть выровнена относительно фрагментов, в которые она будет «вклеена». То есть вам нужно задать не только геометрические координаты объектов сцены, но и координаты текстуры. Например, для двухмерных карт текстур диапазон координат простирается от 0.0 до 1.0 в обоих направлениях, но координаты элементов текстур могут различаться. Так, чтобы применить текстуру кирпича к стене, в допущении, что стена квадратная и, значит, может быть представлена одной копией текстуры, код, по всей видимости, присвоит координаты текстуры (0, 0), (1, 0), (1, 1) и (0, 1) четырем углам стены. Если стена протяженная, вы можете захотеть раскрасить ее несколькими экземплярами текстур. Если сделать так, карта текстуры должна быть построена с учетом того, чтобы кирпичи с левого края соответствовали кирпичам с правой стороны, и аналогично нужно поступить с верхними и нижними краями кладки.

Вы должны также указать, как следует обрабатывать координаты, выходящие за диапазон [0.0, 1.0]. Повторить ли текстуру вдоль всего объекта или фиксировать ее в определенных границах?

Пример программы

Одна из проблем с написанием простых программ, иллюстрирующих наложение текстур, — то, что представляющие интерес текстуры велики. Обычно текстуры читаются из файла, так как их программное описание может включить в себя тысячи строк кода. В листинге 9.1 текстура, состоящая из перемежающихся черных и белых квадратов, подобно шахматной доске, генерируется программой. Программа применяет эту текстуру к двум квадратам, воспроизводимым с учетом перспективы. Один из них обращен «лицом» (находится под прямым углом) к наблюдателю, а другой наклонен назад на 45°, как видно из рис. 9.2. В координатах объекта оба квадрата имеют одинаковый размер.

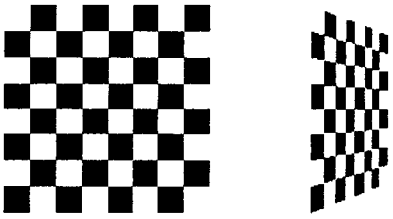


Рис. 9.2. Текстурированные квадраты

Листинг 9.1. Текстурированная шахматная доска: checker.c

```
/* Создание текстуры шахматной доски */
#define checkImageWidth 64
#define checkImageHeight 64

static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLuint texName;

void makeCheckImage(void)
{
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {
      for (j = 0; j < checkImageWidth; j++) {
         c = ((((i&0x8)==0)^((j&0x8))==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
         checkImage[i][j][3] = (GLubyte) 255;
      }
   }
}

void init(void)
```

```
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glEnable(GL_DEPTH_TEST);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glGenTextures(1, &texName);
   glBindTexture(GL_TEXTURE_2D, texName);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
      checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, checkImage);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glEnable(GL_TEXTURE_2D);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
   glBindTexture(GL_TEXTURE_2D, texName);
   glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
   glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
   glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
   glEnd();
   glFlush();
   glDisable(GL_TEXTURE_2D);
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 30.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.0, -3.6);
}
/* keyboard() и main() удалены с целью сокращения объема */
```

Текстура в клетку генерируется процедурой `makeCheckImage()`, а вся инициализация процесса наложения текстур происходит в теле `init()`. Команда `glGenTextures()` присваивает имя объекту текстуры, а `glBindTexture()` создает этот объект для изображения текстуры. (См. раздел «Объекты текстур».) Карта текстуры с полным разрешением задается командой `glTexImage2D()`, прини-

мающей в аргументах размер, тип, положение и другие свойства изображения текстуры. (Подробнее о функции `glTexImage2D()` см. раздел «Описание текстуры».)

Четыре вызова `glTexParameter*()` определяют, как текстура будет обработана на границах (wrapped) и то, как должны фильтроваться цвета в случае неточного совпадения текселов и пикселов экрана. (См. разделы «Фильтрация» и «Повторяемые и обрезанные текстуры».)

В процедуре `display()` команда `glEnable()` включает текстурирование. Команда `glTexEnv*()` устанавливает режим рисования `GL_REPLACE`, где текстурируемые многоугольники рисуются с использованием цветов карты текстуры (оригинальный цвет многоугольников во внимание не принимается).

Затем рисуются оба многоугольника. Заметьте, что координаты текстуры задаются вместе с координатами вершин. Команда `glTexCoord*()` ведет себя подобно команде `glNormal()`, она устанавливает текущие координаты текстуры; любая последующая команда для вершин ассоциирует свои данные с координатами текстуры до тех пор, пока не будет снова вызвана `glTexCoord*()`.

ПРИМЕЧАНИЕ

Изображение шахматной доски на наклонном многоугольнике может выглядеть неверно, когда вы компилируете и исполняете код на своем компьютере, — например, оно может напоминать два треугольника с различными проекциями шахматной доски. Если это так, попробуйте установить параметр GL_PERSPECTIVE_CORRECTION_HINT в GL_NICEST (команда glHint()) и запустите пример на исполнение еще раз.

Описание текстуры

Команда `glTexImage2D()` определяет двухмерную текстуру. Она принимает несколько аргументов, кратко описанных ниже и более подробно в следующем разделе. Связанные команды для одномерных и трехмерных текстур – `glTexImage1D()` и `glTexImage3D()`, — описаны в разделах «Одномерные текстуры» и «Трехмерные текстуры» соответственно.

void `glTexImage2D`(GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLsizei *height*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Определяет двухмерную текстуру. Аргумент *target* задается одной из следующих констант: `GL_TEXTURE_2D`, `GL_PROXY_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z` или `GL_PROXY_TEXTURE_CUBE_MAP`. (См. раздел «„Кубические“ текстуры» для получения информации об использовании констант `GL_*CUBE_MAP*` совместно с `glTexImage2D` и связанными функциями.)

Аргумент *level* используется, если вы хотите иметь карты текстур с различными разрешениями; для единственного разрешения значение уровня детализации (*level*) должно быть равно 0. (См. раздел «MIP-отображение: детализация на множестве уровней».)

Следующий аргумент, внутренний формат (*internalFormat*), указывает, какие компоненты (RGBA, глубина, яркость или интенсивность) выбраны для текселов изображения. Значение внутреннего формата представляется номером от 1 до 4 или одной из следующих символических констант: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_COMPRESSED_ALPHA, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_RGB, GL_COMPRESSED_RGBA, GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, GL_DEPTH_COMPONENT32, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12 или GL_RGBA16. (См. раздел «Функции текстур» для справки о том, как применяются выбранные компоненты, и см. раздел «Сжатые изображения текстур» по теме обработки сжатых текстур.)

Внутренний формат может предъявить специфические требования к размерности компонентов. Например, если *internalFormat* установлен в значение GL_R3_G3_B2, вы запрашиваете для текселов по 3 бита для красного компонента, 3 бита — для зеленого и 2 бита для синего. OpenGL не гарантирует, что удовлетворит требования, а только обязуется подобрать внутреннее представление, примерно соответствующее запросу. По определению в этом отношении терпимы параметры GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_DEPTH_COMPONENT, GL_RGB и GL_RGBA, поскольку они не интересуются конкретным разрешением. (В целях совместимости с редакцией OpenGL 1.0 для внутреннего формата поддержаны численные значения 1, 2, 3 и 4, которые эквивалентны символическим именам GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB и GL_RGBA соответственно.)

Ширина (*width*) и высота (*height*) задают размерность изображения текстуры; аргумент *border* указывает толщину границы, равную либо 0 (нет границы), либо 1. Ширина и высота должны удовлетворять выражению $2m + 2b$, где m — неотрицательное целое число (оно может быть различным для аргументов *width* и *height*), а b — значение аргумента *border*. Максимальный размер текстуры зависит от реализации OpenGL, но он никогда не меньше 64 × 64 (или 66 × 66 с границами).

Аргументы формат (*format*) и тип (*type*) описывают формат и тип данных изображения текстуры. Они имеют те же значения, что и для команды glDrawPixels(). (См. раздел «Конвейерная обработка изображений» главы 8.) Фактически данные текстуры имеют тот же формат, что и данные для glDrawPixels(), и к ним применимы установки для glPixelStore*() и glPixelTransfer*(). (В программе из листинга 9.1 вызов glPixelStorei(GL_UNPACK_ALIGNMENT, 1) делается потому, что данные в примере не выровнены относительно конца рядов текселов.) Значения аргумента *format* могут быть следующими: GL_COLOR_INDEX, GL_DEPTH_COMPONENT, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE или GL_LUMINANCE_ALPHA — то есть те же, которые допустимы для glDrawPixels(), за исключением GL_STENCIL_INDEX.

Аналогично, аргумент *type* задается константами GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_BITMAP или одним из типов упакованных пикселов.

Наконец, собственно *текселы (texels)* содержат данные изображения текстуры. Эти данные описывают изображение текстуры, равно как и его границы.

Внутренний формат изображения текстуры может сказаться не в пользу производительности текстурирования. Например, отдельные реализации выполняют текстурирование в режиме GL_RGBA быстрее, чем в режиме GL_RGB, так как компоненты оптимально выровнены в оперативной памяти. Поскольку это поведение может измениться, вам следует проверить информацию, имеющую отношение к специфике реализации OpenGL.

Внутренний формат изображения текстуры также может влиять на то, сколько памяти выделяется под это изображение. Так, текстура с форматом GL_RGBA8 использует 32 бита/тексел, в то время как текстура в формате GL_R3_G3_B2 требует только 8 бит/тексел. Разумеется, это отражает соотношение между требованиями к памяти и глубиной цветности.

Текстура GL_DEPTH_COMPONENT чаще сохраняется с информацией глубины, для ее последующего полезного использования. В разделе «Глубина текстур» вы можете прочитать, как глубина текстур может быть использована при воспроизведении теней.

Хотя результаты наложения текстур в режиме индексированных цветов не определены, вам все еще позволено указывать параметр GL_COLOR_INDEX. В этом случае для преобразования индексов в значения RGBA применяются операции при передаче пикселов, согласно таблице подстановки.

Если ваша реализация OpenGL поддерживает расширение Imaging Subset и включены какие-либо его возможности, изображение текстуры включается в область их применения. Так, если фильтр двухмерной сверки включен, свертка будет выполнена и для изображения текстуры. (Свертка может изменить высоту и ширину изображения.)

Количество текселов, указанное для ширины и высоты изображения текстуры, не включая необязательную границу, должно быть степенью 2. Если ваше оригинальное изображение по размерам не соответствует этому ограничению, вы можете использовать подпрограмму GLU gluScaleImage() для переопределения размеров своих текстур.

int gluScaleImage(GLenum *format*, GLint *widthin*, GLint *heightin*, GLenum *typein*, const void **datain*, GLint *widthout*, GLint *heightout*, GLenum *typeout*, void **dataout*);

Масштабирует изображение с использованием соответствующих режимов хранения пикселов. Распаковываемые данные извлекаются из массива *datain*. Аргументы *format*, *typein* и *typeout* могут ссылаться на любой из форматов и типов данных, поддерживаемых glDrawPixels(). Изображение масштабируется с помощью линейной интерполяции и фильтрации (от размеров *widthin* и *heightin* к размерам *widthout* и *heightout*), результирующие данные записыва-

ются в массив *dataout*, с использованием режимов хранения пикселов `GL_PACK*`. При вызове `gluScaleImage()` нужно выделить достаточное пространство для выходного буфера. Функция возвращает 0 при успехе и код ошибки GLU при неудаче.

ПРИМЕЧАНИЕ

В GLU 1.3 команда gluScaleImage() поддерживает форматы упакованных пикселов (и связанные с ними типы данных).

Сам буфер кадров может также выступать в роли источника данных для текстуры. Команда `glCopyTexImage2D()` читает прямоугольник пикселов из буфера кадров и использует этот прямоугольник как текселы новой текстуры.

void `glCopyTexImage2D`(GLenum *target*, GLint *level*, GLint *internalFormat*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*, GLint *border*);

Создает двухмерную текстуру, выбирая данные из буфера кадров для определения текселов. Пикселы читаются из текущего буфера `GL_READ_BUFFER` и обрабатываются так, как это делает команда `glCopyPixels()`, но вместо поступления в буфер кадров пикселы помещаются в память текстуры. Здесь применимы настройки `glPixelTransfer*()` и других операций при передаче пикселов.

Аргумент *target* должен быть одной из констант: `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z` или `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`. (См. раздел «„Кубические" текстуры» для справки об использовании констант `*CUBE_MAP*`.) Аргументы *level*, *internalFormat* и *border* задаются так же, как и для `glTexImage2D()`. Массив текстуры выбирается из выровненного относительно экрана прямоугольника пикселов с координатами левого нижнего угла, равными значению аргументов *x* и *y*. Ширина (*width*) и высота *(height)* задают размеры прямоугольника пикселов. Оба этих аргумента должны удовлетворять выражению $2m + 2b$, где m —неотрицательное целое (оно может быть различным для *width* и *height*), а b — значение аргумента *border*.

Следующие разделы более подробно описывают текстурирование, включая использование аргументов приемника (*target*), границы (*border*) и уровня детализации (*level*). Аргумент *target* может быть использован для запроса точного размера текстуры (командой `glTexImage*D()` создается текстура-посредник) с целью выяснить, допустима ли такая текстура с точки зрения выделяемых реализацией OpenGL ресурсов. Переопределение части текстуры описано в разделе «Замещение изображения текстуры или его части». Одномерные и трехмерные текстуры обсуждаются в разделах «Одномерные текстуры» и «Трехмерные текстуры». Границам текстур, которыми можно управлять с помощью параметра *border*, уделено внимание в разделе «Сжатые изображения текстур». Аргумент *level* используется для задания текстур с различным разрешением, соединяемых воедино специальной техникой *MIP-отображения* (mipmapping), которой посвящен раздел «MIP-отображение: детализация на множестве уровней». Использование MIP-отображений требует также понимания особенностей применения к ним техники фильтрации, которая рассматривается в одноименном разделе.

Текстуры-посредники

Для OpenGL-программиста, использующего текстуры, критичными являются их размеры. Ресурсы для текстур обычно ограничены, а ограничения на форматы текстур рознятся от реализации к реализации. Поэтому предусмотрены специальные текстуры-посредники (texture proxy), ориентированные на оценку возможностей конкретной реализации OpenGL в отношении определенных формата и размеров текстуры.

Команда `glGetIntegerv(GL_MAX_TEXTURE_SIZE,...)` выдает нижний предел наибольших доступных ширины и высоты (без границ) изображения текстуры; обычно это размеры наибольшей квадратной текстуры. Для трехмерных текстур с аналогичной целью можно использовать параметр `GL_MAX_3D_TEXTURE_SIZE`; в этом случае, помимо ширины и высоты, возвращается предельная глубина трехмерного изображения текстуры. Для кубических (cube map) текстур имеется аналогичный параметр `GL_MAX_CUBE_MAP_TEXTURE_SIZE`.

Вместе с тем одни только запросы `GL_MAX*TEXTURE_SIZE` не учитывают влияние внутреннего формата и другие факторы. Изображение текстуры, текселы которого хранятся в формате `GL_RGBA16` — 64 бита/тексел, по площади в 16 раз меньше изображения, хранимого в формате `GL_LUMINANCE4`. Текстуры с границами или MIP-картами предъявляют еще большие требования к памяти.

Специальный «местозаполнитель», иначе *посредник* (proxy), изображения текстуры позволяет программе узнать более точно, в состоянии ли OpenGL разместить текстуру с желаемым внутренним форматом.

Например, чтобы выяснить, достаточно ли ресурсов для размещения стандартной 2D-текстуры, вызовите `glTexImage2D()` со значением аргумента *target*, равным `GL_PROXY_TEXTURE_2D`, с заданными уровнем детализации, внутренним форматом, шириной, высотой, границами, форматом и типом. Для случая текстур-посредников вам следует передать `NULL` в аргументе *texels*. (Для кубических карт используйте `glTexImage2D()` со значением *target*, равным `GL_PROXY_TEXTURE_CUBE_MAP`. Для одномерных и трехмерных текстур используйте соответствующие подпрограммы и символические константы.)

После того как текстура-посредник создана, ее параметры состояния запрашиваются командой `glGetTexLevelParameter*()`. Если ресурсов недостаточно, параметры состояния для ширины, высоты, толщины границ и размерности сбрасываются в 0.

void `glGetTexLevelParameter{if}v`(GLenum *target*, GLint *level*, GLenum *pname*, *TYPE* **params*);

Возвращает массив значений параметров (*params*) текстуры для указанного уровня детализации (*level*). Аргумент *target* определяет целевую текстуру как `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`, `GL_PROXY_TEXTURE_1D`, `GL_PROXY_TEXTURE_2D`, `GL_PROXY_TEXTURE_3D` или `GL_PROXY_TEXTURE_CUBE_MAP`.

(Значение GL_TEXTURE_CUBE_MAP недействительно, поскольку нельзя указать определенную поверхность «кубической» карты.) Допустимые значения для *pname* — это GL_TEXTURE_WIDTH, GL_TEXTURE_HEIGHT, GL_TEXTURE_DEPTH, GL_TEXTURE_BORDER, GL_TEXTURE_INTERNAL_FORMAT, GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE, GL_TEXTURE_BLUE_SIZE, GL_TEXTURE_ALPHA_SIZE, GL_TEXTURE_LUMINANCE_SIZE и GL_TEXTURE_INTENSITY_SIZE.

Листинг 9.2 демонстрирует использование текстуры-посредника для случая текстуры с размерами 64 × 64 тексела и RGBA-компонентами 8-битовой размерности. При успехе команда glGetTexLevelParameteriv() сохраняет значение внутреннего формата (в нашем случае GL_RGBA8) в переменной.

Листинг 9.2. Использование текстуры-посредника

```
GLint width;

glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA8,
  64, 64, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
  GL_TEXTURE_WIDTH, &width);
```

ПРИМЕЧАНИЕ

Для текстур-посредников есть одно главное ограничение: они дают ответ на вопрос, можно ли разместить структуру в оперативной памяти, но без скидки на требования других текстур. Поэтому, если текстура-посредник отвечает утвердительно, это не значит, что текстура может быть резидентной (то есть готова стать частью высокопроизводительного рабочего набора текстур). То есть текстура-посредник не дает ответа на вопрос, хватит ли ресурсов для манипулирования целевой текстурой. (См. раздел «Объекты текстур» для получения большей информации относительно управления резидентными текстурами).

Замещение изображения текстуры или его части

Создание текстуры может быть в отношении расходования вычислительных мощностей более затратно, нежели модификация существующей. Часто лучше заменить часть изображения текстуры или его полностью новой информацией, чем создавать ее заново. Это бывает полезно для определенного типа приложений, таких как приложения реального времени, которые захватывают видеоизображения как изображения текстур. Для таких приложений имеет смысл создать единственную текстуру и обращаться к команде glTexSubImage2D() для многократного замещения данных текстуры из новых видеоизображений. Также те ограничения, которые вынуждают делать размеры изображения равными степени двойки, не принципиальны для glTexSubImage2D().

ПРИМЕЧАНИЕ

Это полезно для обработки видеокадров, размеры которых в общем случае произвольны. Тем не менее вы должны загружать видеокадры в исходное, большое изображение, с размерами в 2^n текселов в каждом измерении, а для вкладываемых изображений подстраивать координаты текстуры.

void `glTexSubImage2D`(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Определяет двухмерное изображение текстуры, замещающее собой существующее двухмерное изображение или его некусочный участок (в 2D это просто прямоугольник). Аргумент *target* должен быть установлен аналогично команде `glCopyTexImage2D`.

Аргументы *level*, *format* и *type* используются аналогично `glTexImage2D()`. Уровень детализации (*level*) — это номер уровня детализации для MIP-отображения. Не является ошибкой задание нулевых ширины и высоты, но эффекта на изображение это никакого не производит. Аргументы *format* и *type* описывают формат и тип данных изображения текстуры. На неполное (парциальное) изображение также влияют режимы, установленные `glPixelStore*()` и `glPixelTransfer*()`, и другие операции при передаче пикселов.

Массив *texels* содержит данные текстуры для парциального изображения. Ширина (*width*) и высота (*height*) — это размеры подобласти, замещающей изображение или его часть. Аргументы *xoffset* и *yoffset* указывают смещение тексела в x- и y-направлениях — (0, 0) соответствует левому нижнему углу текстуры, — то есть положение в массиве текстуры, в которое будет помещено неполное изображение. Эта область не может включать в себя любые текселы вне изначально определенного массива текстуры.

В программе из листинга 9.3 присутствует код из листинга 9.1, модифицированный так, чтобы по нажатию клавиши `s` в существующее изображение вставлялось небольшое изображение «в шахматную клетку». (Результирующая текстура показана на рис. 9.3.) При нажатии клавиши `r` восстанавливается исходное изображение. Листинг 9.3 содержит две процедуры, `makeCheckImages()` и `keyboard()`, которые были основательно изменены. (См. раздел «Объекты текстур» для получения большей информации о команде `glBindTexture()`.)

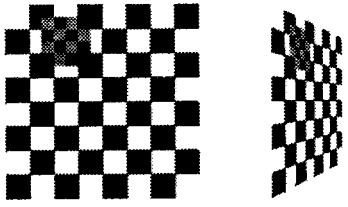


Рис. 9.3. Текстура с добавленным неполным изображением

Листинг 9.3. Замещение части изображения текстуры: texsub.c

```
/* Создание клеточных текстур */
#define checkImageWidth 64
#define checkImageHeight 64
#define subImageWidth 16
#define subImageHeight 16

static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLubyte subImage[subImageHeight][subImageWidth][4];
```

```
void makeCheckImages(void)
{
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {
      for (j = 0; j < checkImageWidth; j++) {
         c = (((i&0x8)==0) ^ ((j&0x8)==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
         checkImage[i][j][3] = (GLubyte) 255;
      }
   }
   for (i = 0; i < subImageHeight; i++) {
      for (j = 0; j < subImageWidth; j++) {
         c = (((i&0x4)==0) ^ ((j&0x4)==0))*255;
         subImage[i][j][0] = (GLubyte) c;
         subImage[i][j][1] = (GLubyte) 0;
         subImage[i][j][2] = (GLubyte) 0;
         subImage[i][j][3] = (GLubyte) 255;
      }
   }
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 's':
      case 'S':
         glBindTexture(GL_TEXTURE_2D, texName);
         glTexSubImage2D(GL_TEXTURE_2D, 0, 12, 44,
         subImageWidth, subImageHeight, GL_RGBA,
            GL_UNSIGNED_BYTE, subImage);
         glutPostRedisplay();
         break;
      case 'r':
      case 'R':
         glBindTexture(GL_TEXTURE_2D, texName);
         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
            checkImageWidth, checkImageHeight, 0,
            GL_RGBA, GL_UNSIGNED_BYTE, checkImage);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
         default:
         break;
   }
}
```

Напомним еще раз, что сам буфер кадров может быть использован в качестве источника данных текстуры — здесь это часть изображения текстуры. Команда `glCopyTexSubImage2D()` читает прямоугольник пикселов из буфера кадров и за-

мещает им часть существующего объекта текстуры. (Эта команда похожа на команды `glCopyTexImage2D()` и `glTexSubImage2D()`.)

void `glCopyTexSubImage2D`(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

Применяет данные изображения из буфера кадров для замены существующего двухмерного изображения текстуры или его некусочной подобласти. Пикселы читаются из текущего буфера `GL_READ_BUFFER` и обрабатываются аналогично команде `glCopyPixels()`, но вместо поступления в буфер кадров они помещаются в память текстуры. К этой команде применимы установки для команды `glPixelTransfer*()` и другие операции при передаче пикселов.

Аргумент *target* устанавливается аналогично команде `glCopyTexImage2D`. Аргумент *level* — номер уровня детализации для MIP-текстур. Аргументы *xoffset* и *yoffset* указывают смещение текселя в x- и y-направлениях — (0, 0) соответствует левому нижнему углу текстуры, — то есть положение в массиве текстуры, в которое будет помещено парциальное изображение. Замещающее изображение берется из выровненного относительно экрана прямоугольника пикселов с координатами левого нижнего угла (x, y). Аргументы *width* и *height* задают размеры прямоугольника подобласти.

Одномерные текстуры

Иногда достаточно одномерной текстуры, например, если вы рисуете текстурированные полосы и все изменения однонаправлены. Одномерная текстура ведет себя подобно двухмерной с высотой *(height)*, равной 1, и без верхней и нижней границ. Все двухмерные текстуры и их подтекстуры имеют соответствующие «одномерные» процедуры-аналоги. Чтобы создать простую одномерную текстуру, используйте команду `glTexImage1D()`.

void `glTexImage1D`(GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Определяет одномерную текстуру. Все принимаемые аргументы имеют те же значения, что и для команды `glTexImage2D()`, с той разницей, что *texels* – здесь одномерный массив. Как и раньше, ширина (*width)* или равна 2^m (или $2^m + 2$, с учетом границ), где m — неотрицательное целое число. Вы также можете применять множественное отображение и текстуру-посредник (установив аргумент *target* в `GL_PROXY_TEXTURE_1D`), и все так же доступны параметры фильтрации.

Пример простой программы, использующей одномерную карту текстуры, приведен в листинге 9.8.

Если ваша реализация OpenGL поддерживает расширение Imaging Subset и если включен одномерный фильтр свертки (`GL_CONVOLUTION_1D`), эта свертка применяется и к изображению текстуры. (Свертка может изменить его ширину.) Сохраняют силу и другие операции над пикселами.

Для замены одномерной текстуры целиком или части ее текселов используйте команду `glTexSubImage1D()`.

void glTexSubImage1D(GLenum *target*, GLint *level*, GLint *xoffset*, GLsizei *width*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Определяет массив одномерной текстуры, которая замещает собой существующее целое изображение одномерной текстуры или его некусочную область (для 1D это ряд). Аргумент *target* должен быть установлен в значение GL_TEXTURE_1D.

Аргументы *level*, *format* и *type* подобны используемым для glTexImage1D(). Уровень (*level*) — это номер уровня детализации MIP-отображения. Аргументы *format* и *type* описывают формат и тип данных изображения текстуры. В отношении парциального изображения также эффективны режимы, устанавливаемые командами glPixelStore*(), glPixelTransfer*() или другими операциями при передаче пикселов.

Массив *texels* содержит данные текстуры для неполного изображения. Ширина (*width*) — это количество замещающих текселов. Аргумент *xoffset* указывает смещение текселя в существующем массиве текстуры, куда будет помещено неполное изображение.

Чтобы задействовать буфер кадров в качестве источника новой одномерной текстуры или для подмены старой текстуры либо ее части, используйте команды glCopyTexImage1D() или glCopyTexSubImage1D().

void glCopyTexImage1D(GLenum *target*, GLint *level*, GLint *internalFormat*, GLivnt *x*, GLint *y*, GLsizei *width*, GLint *border*);

Создает одномерную текстуру с использованием данных буфера кадров для определения ее текселов. Пикселы читаются из текущего буфера GL_READ_BUFFER и обрабатываются аналогично команде glCopyPixels(), но вместо поступления в буфер кадров пикселы помещаются в память текстуры. Здесь сохраняют силу установки для glPixelStore*() и glPixelTransfer*().

Аргумент *target* должен быть установлен в значение GL_TEXTURE_1D. Уровень детализации (*level*), внутренний формат (*internalFormat*) и границы (*border*) задаются подобно glCopyTexImage2D(). Массив текстуры берется из строки пикселов с координатами левого конца (*x*, *y*). Ширина (*width*) указывает количество пикселов в строке. Это значение должно быть равно 2^m (или $2^m + 2$, без учета границ), где m — неотрицательное целое число.

void glCopyTexSubImage1D(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *x*, GLint *y*, GLsizei *width*);

Применяет данные изображения в буфере кадров для замены существующего одномерного изображения текстуры или ее некусочной подобласти. Пикселы читаются из текущего буфера GL_READ_BUFFER и обрабатываются аналогично команде glCopyPixels(), но вместо поступления в буфер кадров они помещаются в память текстуры. Сохраняют силу установки для glPixelTransfer*() и других операций при передаче пикселов.

Аргумент *target* должен быть установлен в значение GL_TEXTURE_1D. Аргумент *level* — это номер уровня детализации MIP-отображения. Аргумент *xoffset* указывает смещение текселя в пределах массива существующей текстуры, по которому помещается парциальное изображение. Массив текстуры участка изображения берется из строки пикселов с координатами левого конца (*x*, *y*). Ширина (*width*) задает количество пикселов в ряду.

Трехмерные текстуры

Трехмерные текстуры обычно используются в медицинских и геологических приложениях. В первых такие текстуры могут потребоваться для послойно представленных результатов компьютерной томографии (CT) или для изображений, полученных при магнитной резонансной томографии (MRI). При изысканиях нефти или газа трехмерные текстуры помогут смоделировать структуру горных пород.

ПРИМЕЧАНИЕ

Трехмерное текстурирование — это часть более общей категории приложений, выполняющих объемную визуализацию. Некоторые такие усовершенствованные приложения имеют дело с вокселами (voxels), трехмерными аналогами пикселов.

По причине своего размера трехмерные текстуры могут потреблять значительный объем ресурсов. Даже относительно грубая трехмерная текстура может требовать в 16 или 32 раза больше памяти, чем ее двухмерное подобие. Для большинства двухмерных текстур и подтекстур имеются соответствующие «трехмерные» процедуры-аналоги.

Об изображении трехмерной текстуры можно думать как о стопке двухмерных прямоугольников. В памяти эти прямоугольники упорядочены последовательно. Чтобы создать простую трехмерную текстуру, используйте команду `glTexImage3D()`.

void `glTexImage3D`(GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLsizei *height*, GLsizei *depth*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Определяет трехмерную текстуру. Все аргументы принимают те же значения, что и для команды `glTexImage2D()`, с той разницей, что здесь *texels* — трехмерный массив, образованный добавлением параметра глубины (*depth*). Значение глубины должно быть равно 2^m (или $2^m + 2$ с учетом границ), где m — неотрицательное целое число. Вы можете применять MIP-отображение и текстуры-посредники (установив аргумент *target* в значение `GL_PROXY_TEXTURE_3D`), и по-прежнему доступны параметры фильтрации.

ПРИМЕЧАНИЕ

Трехмерные свертки не поддержаны расширением Imaging Subset. Тем не менее в вашем распоряжении имеются 2D-фильтры, которые применимы и к трехмерным изображениям текстур.

Листинг 9.4 представляет часть программы, использующей трехмерную карту текстуры.

Листинг 9.4. Трехмерное текстурирование: texture3d.c

```
#define iWidth 16
#define iHeight 16
#define iDepth 16

static GLubyte image [iDepth][iHeight][iWidth][3];
static GLuint texName;
```

```
/* Создание 16x16x16x3 массива с различными значениями цвета
 * для каждого элемента массива [r, g, b]. Диапазон значений – от 0 до 255
*/
void makeImage(void)
{
   int s, t, r;

   for (s = 0 ; s < 16 ; s++)
      for (t = 0 ; t < 16 ; t++)
         for (r = 0 ; r < 16 ; r++) {
            image[r][t][s][0] = s * 17;
            image[r][t][s][1] = t * 17;
            image[r][t][s][2] = r * 17;
         }
}
/* Инициализация состояния: 3D-объект текстуры и его изображение */
void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glEnable(GL_DEPTH_TEST);
   makeImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glGenTextures(1, &texName);
   glBindTexture(GL_TEXTURE_3D, texName);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, L_NEAREST);
   glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, iWidth, iHeight,
      iDepth, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
}
```

Чтобы заменить все текселы трехмерной текстуры или их часть, используйте команду `glTexSubImage3D()`.

void `glTexSubImage3D`(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *zoffset*, GLsizei *width*, GLsizei *height*, GLsizei *depth*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Определяет массив трехмерной текстуры, замещающий собой существующее изображение трехмерной текстуры или его некусочную область. Аргумент *target* должен быть установлен в `GL_TEXTURE_3D`.

Аргументы *level*, *format* и *type* используются аналогично команде `glTexImage3D()`. Аргумент *level* – номер уровня детализации MIP-отображения. Аргументы *format* и *type* описывают формат и тип данных изображения текстуры. Для неполных изображений сохраняют силу режимы, установленные `glPixelStore*()`, `glPixelTransfer*()`, а также другие операции при передаче пикселов.

Чтобы использовать буфер кадров в качестве исходных данных для замещения существующей трехмерной структуры или ее части, используйте команду `glCopyTexSubImage3D()`.

void glCopyTexSubImage3D(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *zoffset*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

Применяет данные изображения из буфера кадров для замены существующего изображения трехмерной текстуры или его некусочной области. Пикселы читаются из текущего буфера GL_READ_BUFFER и обрабатываются так, как если бы была вызвана команда glCopyPixels(), но вместо поступления в буфер кадров они помещаются в память текстуры. Сохраняют силу установки для glPixelTransfer*() и других операций при передаче пикселов.

Аргумент *target* должен быть установлен в значение GL_TEXTURE_3D. Аргумент *level* — номер уровня детализации множественного отображения. Массив парциального изображения текстуры берется из выровненного относительно экрана прямоугольника пикселов с координатами левого нижнего угла (*x*, *y*). Ширина (*width*) и высота (*height*) указывают размеры прямоугольника неполного изображения. Аргументы *xoffset*, *yoffset* и *zoffset* задают смещение в текселах в пределах существующего массива текстуры, по которому помещается неполное изображение. Поскольку парциальное изображение является двухмерным прямоугольником, замещается только одна вырезка (slice) трехмерной текстуры (вырезка, фиксированная по *zoffset*).

Режимы хранения пикселов для трехмерных текстур

Значения переменных режимов хранения пикселов управляют межстрочным пространством для каждого слоя (другими словами, одного двухмерного прямоугольника). Команда glPixelStore*() устанавливает эти режимы с помощью таких параметров, как *ROW_LENGTH, *ALIGNMENT, *SKIP_PIXELS и *SKIP_ROWS (где * — GL_UNPACK_ или GL_PACK_). Эти параметры управляют обращением к вложенному или к полному прямоугольнику пикселов или текселов. (Данные режимы ранее обсуждались в разделе «Режимы хранения пикселов».)

Вышеупомянутые режимы сохраняют свою полезность для двух из трех измерений, но появляется потребность в дополнительных режимах для поддержки обращения к вложенным объемам данных трехмерных текстур. За их установку отвечают новые параметры, *IMAGE_HEIGHT и *SKIP_IMAGES, позволяющие подпрограммам glTexImage3D(), glTexSubImage3D() и glGetTexImage() обращаться к желаемым подобъемам (subvolume).

Если трехмерная текстура в памяти больше определяемого объема, вам нужно указать высоту отдельного неполного изображения с помощью параметра *IMAGE_HEIGHT. Также, если подобъем не начинается с первого слоя, следует установить параметр *SKIP_IMAGES.

*IMAGE_HEIGHT — это параметр, определяющий высоту (количество рядов) отдельного слоя изображения трехмерной текстуры. Если его значение нулевое (отрицательные числа недействительны), то количество рядов в каждом двухмерном прямоугольнике и есть значение *height*, которое передается в процедуру glTexImage3D() или glTexSubImage3D(). (Это, может быть, лишние слова, так как значение *IMAGE_HEIGHT равно нулю по умолчанию.) Иначе высота отдельного слоя равна значению *IMAGE_HEIGHT.

На рис. 9.4 показано, как параметр *IMAGE_HEIGHT определяет высоту изображения (когда *height* определяет только высоту неполного изображения). Здесь изображена трехмерная текстура только с двумя слоями.

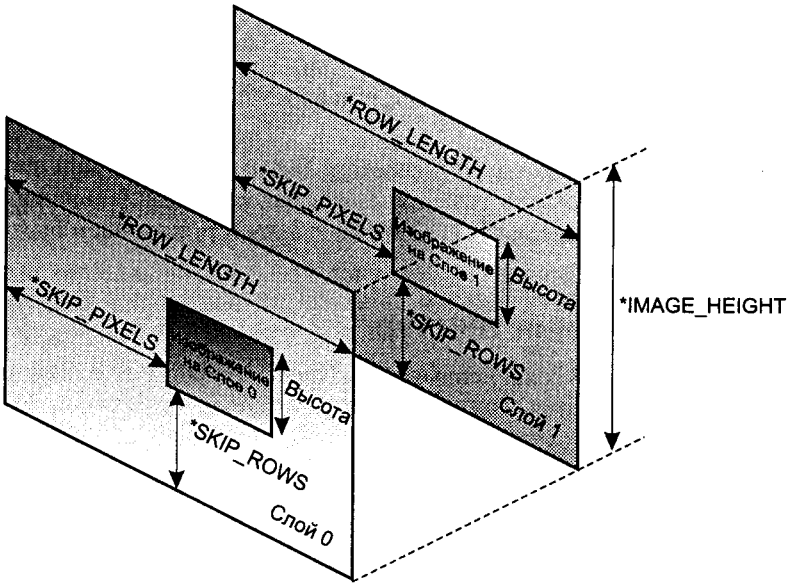


Рис. 9.4. Режим сохранения пикселов *IMAGE_HEIGHT

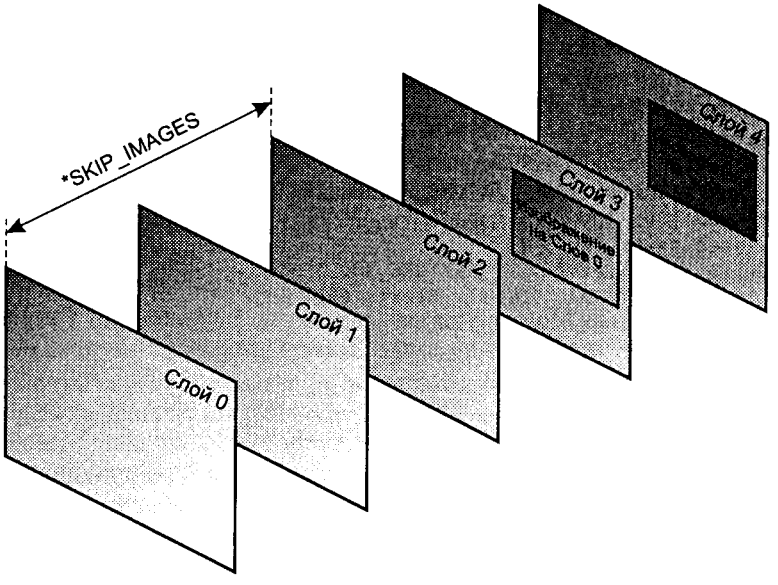


Рис. 9.5. Режим сохранения пикселов *SKIP_IMAGES

Параметр `*SKIP_IMAGES` определяет, сколько слоев пропускается до начала данных подобъема. Если этот параметр инициирован положительным целым числом (назовем его *n*), указатель на данные изображения текстуры продвигается сквозь несколько слоев (*n* * — размер одного слоя текселов). Результирующий подобъем берет начало в слое *n* и имеет глубину в несколько слоев — их количество задается аргументом *depth*, передаваемым в функцию `glTexImage3D()` или `glTexSubImage3D()`. Если значение параметра `*SKIP_IMAGES` равно нулю (по умолчанию), доступ к данным текстуры выполняется с первого слоя, описанного в массиве текстуры.

На рис. 9.5 показано, как благодаря параметру `*SKIP_IMAGES` можно пропустить несколько слоев, чтобы попасть туда, где в действительности начинается парциальный объем. В этом примере `*SKIP_IMAGES` = 3, то есть подобъем берет начало в слое с номером 3.

Сжатые изображения текстур

Карты текстур могут быть сохранены внутри OpenGL в формате со сжатием с целью возможного уменьшения объема памяти, занимаемого текстурой. Изображение текстуры может быть сжато после его загрузки или сразу загружено в формате со сжатием.

Сжатие изображения текстуры во время его загрузки

Чтобы получить сжатое изображение текстуры по ходу его загрузки, укажите одно из именованных значений `GL_COMPRESSED_*` для аргумента *internalformat*. Изображение будет автоматически сжато после того, как текселы пройдут обработку какими-либо активными операциями при сохранении пикселов (см. раздел «Режимы хранения пикселов» главы 8) или при передаче пикселов (см. раздел «Операции при передаче пикселов» главы 8). После загрузки изображения вы можете выяснить, сжато ли оно, и в какой формат, с помощью следующего кода:

```
GLboolean compressed;
GLenum textureFormat;
GLsizei imageSize;
glGetTexLevelParameteriv(GL_TEXTURE_2D, GL_TEXTURE_COMPRESSED,
   &compressed);
if (compressed == GL_TRUE) {
   glGetTexLevelParameteriv(GL_TEXTURE_2D,
      GL_TEXTURE_INTERNAL_FORMAT, &textureFormat);
   glGetTexLevelParameteriv(GL_TEXTURE_2D,
      GL_TEXTURE_COMPRESSED_IMAGE_SIZE, &imageSize);
}
```

Загрузка ранее сжатых изображений текстур

OpenGL не предписывает какой-либо определенный внутренний формат сжатия для текстур; каждая реализация позволяет уточнить расширения OpenGL, отвечающие за реализацию определенного формата сжатия. Если структура загружается в уже сжатом виде, важно знать, в каком формате она сохранена, и проверить, допустим ли этот формат в вашей реализации OpenGL.

Чтобы загрузить текстуру, ранее сохраненную в формате со сжатием, используйте команды `glCompressedTexImage*D()`.

void `glCompressedTexImage1D`(GLenum *target*, GLint *level*, GLenum *internalformat*, GLsizei *width*, GLint *border*, GLsizei *imageSize*, GLvoid **texels*);

void `glCompressedTexImage2D`(GLenum *target*, GLint *level*, GLenum *internalformat*, GLsizei *width*, GLsizei *height*, GLint *border*, GLsizei *imageSize*, GLvoid **texels*);

void `glCompressedTexImage3D`(GLenum *target*, GLint *level*, GLenum *internalformat*, GLsizei *width*, GLsizei *height*, GLsizei *depth*, GLint *border*, GLsizei *imageSize*, GLvoid **texels*);

Получают одномерную, двухмерную или трехмерную текстуру из предварительно сжатого изображения текстуры.

Используйте аргумент *level*, если требуются текстуры с несколькими разрешениями; если вас интересует только одно разрешение, значение уровня детализации (*level*) должно быть нулем. (См. раздел «MIP-отображение: детализация на множестве уровней».)

Аргумент *internalformat* задает формат сжатия изображения текстуры. Этот формат должен быть поддержан в реализации, загружающей текстуру, иначе генерируется ошибка `GL_INVALID_ENUM`. За справкой о том, какие форматы в каких реализациях поддерживаются, обращайтесь к приложению Б.

Ширина (*width*), высота (*height*) и глубина (*depth*) — измерения изображения одномерной, двухмерной и трехмерной текстур. Как и для несжатых текстур, аргумент *border* указывает толщину границы — 0 (нет границы) или 1. Значения измерений должны быть равны $2^m + 2b$, где m — неотрицательное целое число, а b — значение аргумента *border*.

Сжатые текстуры могут быть использованы, равно как несжатые, для замены уже загруженной текстуры или ее части. Это делается командами `glCompressedTexSubImage*D()`.

void `glCompressedTexSubImage1D`(GLenum *target*, GLint *level*, GLint *xoffset*, GLsizei *width*, GLenum *format*, GLsizei *imageSize*, GLvoid **texels*);

void `glCompressedTexSubImage2D`(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffet*, GLsizei *width*, GLsizei *height*, GLsizei *imageSize*, GLvoid **texels*);

void `glCompressedTexSubImage3D`(GLenum *target*, GLint *level*, GLint *xoffset* GLint *yoffset*, GLint *zoffset*, GLsizei *width*, GLsizei *height*, GLsizei *depth*, GLsizei *imageSize*, GLvoid **texels*);

Получают одномерную, двухмерную или трехмерную текстуру из предварительно сжатого изображения текстуры.

Аргументы *xoffset*, *yoffset* и *zoffset* указывают смещение в пикселах в пределах существующего массива текстуры для ее соответствующего измерения, по которому будет помещено новое изображение.

Аргументы *width*, *height* и *depth* определяют ширину, высоту и глубину текстуры с соответствующим числом измерений, используемой для обновления изображения текстуры.

Аргумент *imageSize* задает количество байтов, хранимых в массиве текселов (*texels*).

Использование границ текстуры

Если вам нужно получить карту текстуры по размерам большую, чем это позволяет реализация OpenGL, вы можете формировать такие структуры посредством плиточного размещения (tiling) нескольких различных текстур. Например, если вас интересует квадратная текстура в два раза больше максимального размера, нарисуйте рядом четыре меньших квадрата (два на два) и загрузите в них текстуры перед рисованием.

Поскольку одномоментно доступна только одна карта текстуры, этот подход может упредить «краевые» проблемы с текстурами, особенно если включена линейная фильтрация какого-либо вида. Значение текстуры, используемое для пикселов на краях изображений, должно быть усреднено с чем-то по ту сторону границы, и это что-то, в идеале, должно браться из смежной карты текстуры. Если вы определили границы для каждой текстуры, значения текселов которых равны значениям текселов на краю соседней карты текстуры, результаты останутся корректны и после линейной фильтрации.

Чтобы все было корректно, учтите, что каждая карта имеет восемь соседей — примыкающих к ее сторонам и касающихся ее углов. Значения текселов в углах должны соответствовать значениям текселов в картах текстур, которые касаются углов данной текстуры. Если одна ваша текстура имеет ребро или точку соприкосновения с другой, решите, какие значения разумно присвоить текселам границы. Проще всего и разумнее достичь этого копированием значений смежных текселов двух карт с помощью команды `glTexSubImage2D()`.

Цвет границ также принимает участие в этом, если текстура применяется так, что лишь частично покрывает собой примитив. (См. раздел «Повторяемые и обрезанные текстуры».)

MIP-отображение: детализация на множестве уровней

Визуализуемые объекты текстур, подобно другим объектам сцены, располагаются на различных расстояниях от точки наблюдения. В динамических сценах, где текстурированный объект перемещается и удаляется от точки наблюдения, карта текстуры должна уменьшаться в размере вместе с изменением размеров спроецированного изображения. Чтобы обеспечить это, OpenGL накладывает фильтр на карту текстуры, подгоняя ее до размеров, нужных для отображения на объект, без внесения нарушающих визуальное равновесие дефектов, таких как блики, вспышки и иные световые флуктуации. Например, чтобы воспроизвести кирпичную стену, вы можете использовать большое изображение текстуры (скажем, 128 × 128 текселов), когда стена находится близко к наблюдателю. Когда же стена начнет двигаться в направлении от наблюдателя, до тех пор, пока не выродится в единственный пиксел экрана, фильтрованные текстуры в некоторых переходных точках могут производить впечатление дергающихся.

Во избежание таких дефектов вы можете задать группу предварительно отфильтрованных текстур, последовательно уменьшая их разрешение на разных уровнях

детализации, как показано на рис. 9.6. Термин *множественное отображение* (mipmap) ввел в обиход Ланс Вильямс (Lance Williams), когда представлял эту концепцию в своей статье *Pyramidal Parametrics* (труды конференции SIGGRAPH, 1983). *Mip* — это акроним от латинизма *multum in parvo* («многое в малом»), в интерпретации «много вещей в маленьком месте». В MIP-отображении нашли применение некоторые искусные методы упаковки данных изображений в памяти.

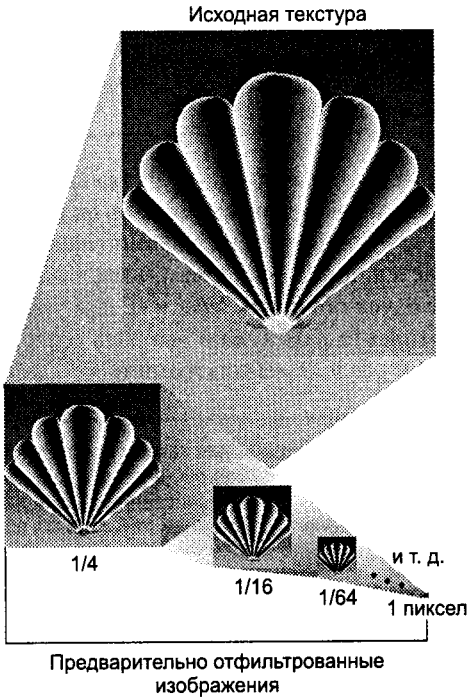


Рис. 9.6. MIP-карты

ПРИМЕЧАНИЕ

Чтобы до конца разобраться с MIP-картами, вам нужно понимать работу фильтров миниатюризации (minification), которые описаны в разделе «Фильтрация».

Когда используется множественное отображение, OpenGL автоматически вычисляет, какие карты текстур нужно использовать, отталкиваясь от размера (в пикселах) объекта, на который производится наложение текстур. При таком подходе уровень детализации карты текстуры соответствует изображению, рисуемому на сцене, — размер карты текстуры синхронно уменьшается с уменьшением размеров изображения. MIP-отображение требует некоторых дополнительных вычислений и лишнюю память для хранения текстур; тем не менее, когда оно не используется, текстуры, накладываемые на небольшие объекты, могут «мерцать» и «полыхать» при движении этих объектов.

Чтобы использовать множественное отображение, вы должны сделать размеры своих текстур равными степени 2, начиная от карты 1 × 1. Например, если ваша карта с самым высоким разрешением имеет размеры 64 × 16, следует также подготовить карты 32 × 8, 16 × 4, 8 × 2, 4 × 1, 2 × 1 и 1 × 1. Самые маленькие карты обычно представляют собой фильтрованные версии самой большой карты, каждый тексел в которых является средним значением от соответствующих четырех текселов карты с более высоким разрешением.

ПРИМЕЧАНИЕ

Так как OpenGL не предъявляет особых требований к методам расчета карт с низким разрешением, текстуры с различными размерами могли бы быть совершенно не связанными. На практике не связанные между собой текстуры делали бы переключения между версиями текстур очень даже заметными, как на цветном рисунке 20.

Для задания этих текстур вызовите команду `glTexImage2D()` по разу на каждое разрешение с различными значениями аргументов *level*, *width*, *height* и *image*. Первое, с чего нужно начать, — уровень детализации *(level)*. Он указывает, какая из текстур серии определяется; в предыдущем примере для текстуры с наивысшим разрешением 64 × 16 следует объявить *level* = 0, для текстуры 32 × 8 — *level* = 1 и т. д. Для полного комплекта вам нужно выбрать один из возможных методов фильтрации, описанных в разделе «Фильтрация».

ПРИМЕЧАНИЕ

Данное описание MIP-изображения не изобилует такими подробностями, как коэффициент масштабирования (известный как λ) между размером текселя и размером многоугольника. Что такое λ подробнее рассказывается в разделах «Вычисление уровня детализации множественного отображения» и «Управление уровнем детализации MIP-отображения».

Листинг 9.5 иллюстрирует применение серии из шести различных текстур с размерами от 32 × 32 до 1 × 1. Эта программа рисует прямоугольник, удаляющийся от переднего плана и в конце концов сходящийся в точку, как показано на цветном рисунке 20. Заметьте, что координаты текстуры лежат в диапазоне от 0.0 до 8.0, что требует выстраивания плиткой 64 экземпляров карт текстур — по восемь в каждом направлении. Каждая карта имеет различный цвет, с целью показать, как карты сменяют друг друга.

Листинг 9.5. MIP-текстуры: mipmap.c

```
GLubyte mipmapImage32[32][32][4];
GLubyte mipmapImage16[16][16][4];
GLubyte mipmapImage8[8][8][4];
GLubyte mipmapImage4[4][4][4];
GLubyte mipmapImage2[2][2][4];
GLubyte mipmapImage1[1][1][4];

static GLuint texName;

void makeImages(void)
{
   int i, j;
```

```
   for (i = 0; i < 32; i++) {
      for (j = 0; j < 32; j++) {
         mipmapImage32[i][j][0] = 255;
         mipmapImage32[i][j][1] = 255;
         mipmapImage32[i][j][2] = 0;
         mipmapImage32[i][j][3] = 255;
      }
   }
   for (i = 0; i < 16; i++) {
      for (j = 0; j < 16; j++) {
         mipmapImage16[i][j][0] = 255;
         mipmapImage16[i][j][1] = 0;
         mipmapImage16[i][j][2] = 255;
         mipmapImage16[i][j][3] = 255;
      }
   }
   for (i = 0; i < 8; i++) {
      for (j = 0; j < 8; j++) {
         mipmapImage8[i][j][0] = 255;
         mipmapImage8[i][j][1] = 0;
         mipmapImage8[i][j][2] = 0;
         mipmapImage8[i][j][3] = 255;
      }
   }
   for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++) {
         mipmapImage4[i][j][0] = 0;
         mipmapImage4[i][j][1] = 255;
         mipmapImage4[i][j][2] = 0;
         mipmapImage4[i][j][3] = 255;
      }
   }
   for (i = 0; i < 2; i++) {
      for (j = 0; j < 2; j++) {
         mipmapImage2[i][j][0] = 0;
         mipmapImage2[i][j][1] = 0;
         mipmapImage2[i][j][2] = 255;
         mipmapImage2[i][j][3] = 255;
      }
   }
   mipmapImage1[0][0][0] = 255;
   mipmapImage1[0][0][1] = 255;
   mipmapImage1[0][0][2] = 255;
   mipmapImage1[0][0][3] = 255;
}

void init(void)
{
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
   glTranslatef(0.0, 0.0, -3.6);
   makeImages();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

продолжение ⇩

Листинг 9.5 (*продолжение*)

```
    glGenTextures(1, &texName);
    glBindTexture(GL_TEXTURE_2D, texName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
       GL_NEAREST_MIPMAP_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0,
       GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage32);
    glTexImage2D(GL_TEXTURE_2D, 1, GL_RGBA, 16, 16, 0,
       GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage16);
    glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 8, 8, 0,
       GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage8);
    glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 4, 4, 0,
       GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage4);
    glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 2, 2, 0,
       GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage2);
    glTexImage2D(GL_TEXTURE_2D, 5, GL_RGBA, 1, 1, 0,
       GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage1);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glEnable(GL_TEXTURE_2D);
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, texName);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 8.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(8.0, 8.0); glVertex3f(2000.0, 1.0, -6000.0);
    glTexCoord2f(8.0, 0.0); glVertex3f(2000.0, -1.0, -6000.0);
    glEnd();
    glFlush();
}
```

Каждому уровню детализации в листинге 9.5 назначен свой цвет с целью показать, как один замещает другой. В реальной ситуации вы определяете множественные отображения так, чтобы сделать переходы незаметными, насколько это возможно. Поэтому уровни низкого разрешения обычно являются фильтрованными версиями уровней высокого разрешения.

Формирование серии таких карт обычно делается в процессе программирования, а не является частью OpenGL, которая просто не что иное, как библиотека для визуализации. Поскольку конструирование уровней детализации является важной работой, библиотека инструментов OpenGL (GLU) содержит подпрограммы, которые помогают манипулировать изображениями, используемыми в качестве MIP-текстур, как описано в разделе «Автоматическое генерирование MIP-карт».

Вычисление уровня детализации множественного отображения

Вычисление уровня детализации MIP-отображения для текстуры, накладываемой на определенный многоугольник, зависит от фактора масштабирования

между изображением текстуры и размерами текстурируемого многоугольника (в пикселах). Обозначим этот фактор масштабирования как λ, а также определим второе значение, ρ, где $\lambda = \log_2 \rho + \text{lod}_{\text{bias}}$. Так как изображения текстур могут быть многомерными, важно пояснить, что ρ — максимальный фактор масштабирования по всем измерениям.

Поправка lod_{bias} — это уровень детализации отклонения (bias), константа, устанавливаемая командой `glTexEnv*()` для подстройки λ. (За справкой об использовании `glTexEnv*()` для установки уровня детализации отклонения обращайтесь к разделу «Функции текстур».) По умолчанию $\text{lod}_{\text{bias}} = 0.0$, поэтому не вносит вклад в масштабирование. Лучше начать с этого значения по умолчанию и при необходимости вносить небольшие коррективы.

Если $\lambda \leq 0{,}0$, текстура меньше многоугольника, то есть используется фильтр увеличения. Если $\lambda > 0.0$, используется фильтр миниатюризации. В случае уменьшающего фильтра λ указывает уровень MIP-отображения. (Точка переключения от одного фильтра к другому обычно $\lambda = 0.0$, но не всегда. Определенный фильтр MIP-отображения может сдвинуть эту точку.)

Например, если размеры изображения текстуры 64 × 64 тексела, а размеры многоугольника — 32 × 32 пиксела, тогда $\rho = 2.0$ (не 4.0) и, следовательно, $\lambda = 1.0$. Если размеры изображения текстуры равны 64 × 32 тексела и размеры многоугольника 8 × 16 пикселов, тогда $\rho = 8.0$ (x увеличивается с коэффициентом 8.0, а фактор масштабирования для y — 2.0) и, соответственно, $\lambda = 3.0$.

Управление уровнем детализации MIP-отображения

По умолчанию вы должны подготовить карты для каждого разрешения, уменьшая их до одного тексела в каждом направлении. В некоторых случаях таких чрезвычайно маленьких изображений, представляющих ваши данные, можно избежать. Например, можно обратиться к технике, называемой *складыванием мозаики* (mosaicing), где несколько маленьких изображений комбинируются в одну текстуру. Один из примеров мозаики показан на рис. 9.7, здесь множество символов представлено одной текстурой, которая бывает более эффективной, чем создание отдельных текстур для каждого символа. Чтобы наложить на изображение только один символ из текстуры, вам нужно целенаправленно использовать координаты текстуры для отделения интересующей вас буквы.

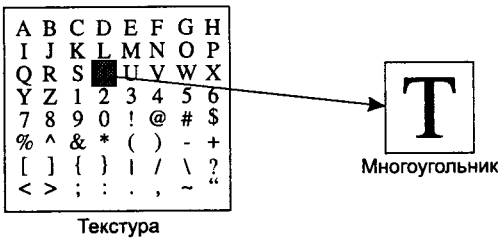


Рис. 9.7. Использование мозаичных текстур

Если вы примените совсем миниатюрные MIP-карты, для таких карт сложение мозаики будет неприемлемым вследствие вырождения деталей начертания букв. Соответственно, может потребоваться наложить ограничения на разрешения. В общем случае нужный результат достигается добавлением или удалением уровней детализации MIP-карт.

Другая проблема визуализации MIP-карт — «скачки» (popping) — внезапные переходы от одной карты к другой, совершенно отличающейся по разрешению, имеющие место, когда текстурируемый многоугольник становится больше или меньше.

ПРИМЕЧАНИЕ

В поздних версиях OpenGL возможности MIP-отображения значительно расширились относительно ранних. Проверьте версию своей реализации на предмет поддерживаемых ею возможностей. В некоторых версиях какие-то из них могут быть доступны как расширение.

Уровни детализации MIP-отображения управляются константами `GL_TEXTURE_BASE_LEVEL`, `GL_TEXTURE_MAX_LEVEL`, `GL_TEXTURE_MIN_LOD` и `GL_TEXTURE_MAX_LOD`, передаваемыми функции `glTexParameter*()`. Первые две константы (для краткости до конца раздела будем употреблять урезанные наименования — `BASE_LEVEL` и `MAX_LEVEL`) отвечают за то, как используются уровни детализации, то есть какие уровни следует задать. Другие две константы (сокращенные нами до `MIN_LOD` и `MAX_LOD`) определяют активный диапазон вышеупомянутого фактора масштабирования λ.

Эти параметры текстуры связаны с несколькими ранее описанными проблемами. Эффективно использовать `BASE_LEVEL` и `MAX_LEVEL` можно, сократив количество задаваемых карт, тем самым демонстрируя рациональный подход к расходованию ресурсов. Избирательное использование `MAX_LOD` может сохранить представительность мозаичных текстур, а `MIN_LOD` смягчит эффект «прыжков» между версиями с резко различающимися разрешениями.

Параметры `BASE_LEVEL` и `MAX_LEVEL` задают граничные значения уровней MIP-отображения. Параметр `BASE_LEVEL` — это уровень для самого высокого разрешения (наибольшей текстуры), по умолчанию его значение равно 0. Тем не менее, вы можете изменять базовый уровень `BASE_LEVEL`, добавляя дополнительные текстуры с высоким разрешением «на лету». Аналогично, `MAX_LEVEL` накладывает ограничение на самое низкое разрешение. Значение по умолчанию для `MAX_LEVEL` равно 1000, а это практически всегда означает, что минимальная по размерам текстура будет представлена одним текселом.

Чтобы задать базовый и максимальный уровни, используйте процедуру `glTexParameter*()` с первым аргументом, установленным в `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` или `GL_TEXTURE_CUBE_MAP`, в зависимости от самой текстуры. Второй аргумент, принимаемый процедурой, это один из параметров, описанных в табл. 9.1. Третий аргумент передает значение указанного параметра.

В коде ли тинга 9.6 базовый и максимальный уровни устанавливаются в значения 2 и 5 соответственно. Так как изображение базового уровня (уровня 2) имеет

разрешение 64 × 32 тексела, карты уровней 3, 4 и 5 должны иметь подходящие более низкие разрешения.

Таблица 9.1. Параметры управления уровнями MIP-отображения

Параметр	Описание	Значения
GL_TEXTURE_BASE_LEVEL	Уровень детализации текстуры с самым высоким разрешением (с минимальным номером)	Любое неотрицательное целое число
GL_TEXTURE_MAX_LEVEL	Уровень детализации текстуры с самым низким разрешением (с наибольшим номером)	Любое неотрицательное целое число

Листинг 9.6. Задание базового и максимального уровней детализации MIP-отображения

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 2);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 5);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 64, 32, 0, GL_RGBA,
  GL_UNSIGNED_BYTE, image1);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 32, 16, 0, GL_RGBA,
  GL_UNSIGNED_BYTE, image2);
glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 16, 8, 0, GL_RGBA,
  GL_UNSIGNED_BYTE, image3);
glTexImage2D(GL_TEXTURE_2D, 5, GL_RGBA, 8, 4, 0, GL_RGBA,
  GL_UNSIGNED_BYTE, image4);
```

Позднее вы можете решить добавить другие карты, как низкого, так и высокого разрешения. Например, вы можете добавить в этот набор текстуру 128 × 64 тексела с уровнем 1, при этом нужно будет переустановить значение параметра BASE_LEVEL.

ПРИМЕЧАНИЕ

Чтобы MIP-отображение работало, все карты от BASE_LEVEL до максимально возможного уровня включительно должны быть загружены. Максимально допустимый уровень — это наименьшее, выбранное из значения MAX_LEVEL или уровня, для которого размер карты составляет только 1 тексел (одно из 1, 1 × 1 или 1 × 1 × 1). Если загрузка карты для необходимого уровня будет неудачной, не удивляйтесь загадочному отключению текстурирования. Если MIP-отображение и текстурирование вдруг не происходят, обеспечьте загрузку для каждого требуемого уровня детализации допустимой текстурой.

Подобно BASE_LEVEL и MAX_LEVEL, команда `glTexParameter*()` устанавливает параметры MIN_LOD и MAX_LOD. Таблица 9.2 содержит список допустимых их значений.

Следующий код представляет пример использования `glTexParameter*()` для задания параметров уровней детализации:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_LOD, 2.5);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_LOD, 4.5);
```

Параметры MIN_LOD и MAX_LOD определяют минимальное и максимальное значения для λ (коэффициент масштабирования от изображения текстуры к многоугольнику) для миниатюризации текстур, косвенно задавая, какие уровни MIP-отображения используются.

Таблица 9.2. Управление параметрами детализации MIP-отображения

Параметр	Описание	Значения
GL_TEXTURE_MIN_LOD	Минимальное значение λ (фактор масштабирования изображения текстуры относительно размеров многоугольника)	Любое
GL_TEXTURE_MAX_LOD	Максимальное значение λ	Любое

Если вы имеете многоугольник 64 × 64 пиксела и оставляете MIN_LOD в значении по умолчанию — 0,0, для миниатюризации можно использовать уровень 0 карты текстуры 64 × 64 текселя (BASE_LEVEL = 0; как правило, BASE_LEVEL ≤ MIN_LOD). Однако, если значение MIN_LOD равно 2.0, размеры самой большой карты, допустимой при миниатюризации, составят 16 × 16 текселов, что соответствует λ = 2.0.

Параметр MAX_LOD вносит свой вклад, только если он меньше максимальной величины λ (которая равна значению MAX_LEVEL или тому, когда карта вырождается в тексел). В случае карты текстуры 64 × 64 текселя соотношение λ = 6.0 соответствует карте 1 × 1. Если же значение MAX_LOD равно 4.0, для миниатюризации не будут использоваться карты меньше 4 × 4 текселя.

Вы можете ориентироваться на то, что значение MIN_LOD, незначительно большее BASE_LEVEL, или значение MAX_LOD, немногим меньшее MAX_LEVEL, — лучшее решение для подавления паразитных эффектов визуализации (таких как «рывки»), связанных с переходами между MIP-картами.

Автоматическое генерирование MIP-карт

Допустим, что вы подготовили карту уровня 0, то есть наивысшего разрешения. Тогда подпрограммы gluBuild1DMipmaps(), gluBuild2DMipmaps() или gluBuild3DMipmaps() сами строят и определяют «пирамиду» MIP-карт, спускающуюся к разрешению 1 × 1 (или 1 для одного измерения, или 1 × 1 × 1 для трехмерной текстуры). Если ваше исходное изображение имеет размеры, не равные в точности степеням двойки, команда gluBuild*DMipmaps() масштабирует изображение к ближайшим степеням 2. Также, если ваша текстура очень велика, gluBuild*DMipmaps() уменьшает размер изображения до вписывания в нужные пределы (оцениваемые механизмом текстур-посредников, GL_PROXY_TEXTURE).

int gluBuild1DMipmaps(GLenum *target*, GLint *internalFormat*, GLint *width*, GLenum *format*, GLenum *type*, void **texels*);

int gluBuild2DMipmaps(GLenum *target*, GLint *internalFormat*, GLint *width*, GLint *height*, GLenum *format*, GLenum *type*, void **texels*);

int gluBuild3DMipmaps(GLenum *target*, GLint *internalFormat*, GLint *width*, GLint *height*, GLint *depth*, GLenum *format*, GLenum *type*, void **texels*);

Строит набор MIP-карт и вызывает glTexImage*D() для загрузки изображений. Значения аргументов *target*, *internalFormat*, *width*, *height*, *depth*, *format*, *type* и *texels* в точности те же, что и для glTexImage1D(), glTexImage2D() и glTexImage3D(). Если все карты сформированы успешно, возвращается значение 0; в противном случае возвращается код ошибки GLU.

При повышенном контроле над уровнем детализации (используя BASE_LEVEL, MAX_LEVEL, MIN_LOD и MAX_LOD) может потребоваться создать только поднабор карт, определенный gluBuild*DMipmaps(). Например, вы можете захотеть остановиться на изображении с размерами 4 × 4 текселя, не доходя до наименьшей текстуры 1 × 1. Чтобы вычислить уровни детализации и загрузить поднабор, вы можете вызвать команду gluBuild*DMipmapLevels().

int gluBuild1DMipmapLevels(GLenum *target*, GLint *internalFormat*, GLint *width*, GLenum *format*, GLenum *type*, GLint *level*, GLint *base*, GLint *max*, void **texels*);

int gluBuild2DMipmapLevels(GLenum *target*, GLint *internalFormat*, GLint *width*, GLint *height*, GLenum *format*, GLenum *type*, GLint *level*, GLint *base*, GLint *max*, void **texels*);

int gluBuild3DMipmapLevels(GLenum *target*, GLint *internalFormat*, GLint *width*, GLint *height*, GLint *depth*, GLenum *format*, GLenum *type*, GLint *level*, GLint *base*, GLint *max*, void **texels*).

Формирует набор MIP-карт и вызывает glTexImage*D() для загрузки изображений. Аргумент *level* указывает уровень детализации MIP-отображения изображения *texels*. Аргументы *base* и *max* определяют деление изображения *texels* на уровни. Другие значения — *target*, *internalFormat*, *width*, *height*, *depth*, *format*, *type* и *texels* такие же, как для glTexImage1D(), glTexImage2D() и glTexImage3D(). При удачном построении всех карт возвращается значение; иначе функция возвращает код ошибки GLU.

Если вы рассчитываете на изменения любых текселов изображения множественной текстуры, вам следует иметь полный набор связанных MIP-карт. Если вы используете glTexParameter*() для установки параметра GL_GENERATE_MIPMAP в значение GL_TRUE, изменения текселов (внутренней области или границ) будут происходить для всех текстур с уровнями BASE_LEVEL + 1 до MAX_LEVEL. Текстуры для всех других уровней, включая BASE_LEVEL, будут оставаться неизменными.

Фильтрация

Текстура — это квадрат или прямоугольник, но после их наложения на многоугольник или поверхность и преобразования в координаты экрана отдельные текселы текстуры редко соответствуют пикселам конечного изображения. В зависимости от примененных преобразований и самого наложения текстуры, один пиксел на экране может соответствовать чему угодно, от крохотной горстки текселов (увеличение) до большого их набора (миниатюризация), как показано на рис. 9.8. В таких случаях сразу неочевидно, как изменятся, усреднятся и интерполируются значения текселов. В силу этой причины OpenGL позволяет вам указывать любые из некоторого количества настроек фильтрации для конкретизации этих вычислений. Эти настройки приводят к компромиссу между быстродействием и качеством изображения. Также вы можете задавать независимые методы фильтрации для увеличения изображений и их миниатюризации.

В иных случаях неочевидно, что выбрать — увеличение или уменьшение, например, если карта текстуры нуждается в растяжении (или в сжатии) в обоих направлениях, x и y. Если карту нужно растянуть лишь в одном направлении и сжать в другом, OpenGL делает выбор между уменьшением и увеличением так, чтобы результат был оптимальным в большинстве случаев. Но лучше избегать подобных деформаций за счет использования координат текстур. (См. раздел «Вычисление подходящих координат текстуры».)

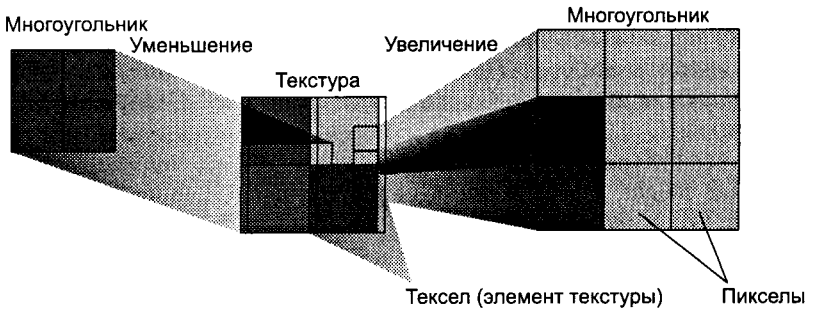


Рис. 9.8. Увеличение и миниатюризация текстур

В следующих строках кода приведены примеры использования команды `glTexParameter*()` для задания методов фильтрации при увеличении и уменьшении карт текстур:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Первый аргумент `glTexParameter*()` — это `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` или `GL_TEXTURE_CUBE_MAP`, согласно типу текстуры. Второй аргумент — `GL_TEXTURE_MAG_FILTER` или `GL_TEXTURE_MIN_FILTER`, для указания того, где будет применяться метод фильтрации — при увеличении или уменьшении. Третий аргумент задает сам метод фильтрации; в табл. 9.3 содержится список допустимых значений.

Таблица 9.3. Методы фильтрации для увеличения и миниатюризации текстур

Параметр	Значения
GL_TEXTURE_MAG_FILTER	GL_NEAREST или GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST или GL_LINEAR_MIPMAP_LINEAR

Если вы выберете `GL_NEAREST`, при фильтрации будет использоваться тексел с координатами, наиболее близкими к центру пиксела. Результатом могут быть неровности в виде ступенек (иногда проявляющиеся сильно). Если вы выберете `GL_LINEAR`, будет использоваться средневзвешенное значение по массиву текселов 2 × 2, ближайших к центру пиксела, как для увеличения, так и для уменьшения. (Для трехмерных текстур это массив 2 × 2 × 2; для одномерных — среднее по двум текселам.) У краев карты текстуры в массив 2 × 2 может попасть нечто, что

находится за ее пределами. В таких случаях выбор значений текселов подчиняется режиму обработки границ (wrapping) (см. раздел «Повторяемые и обрезанные текстуры»). Приближение GL_NEAREST требует меньше вычислений, чем GL_LINEAR, и, следовательно, может работать более быстро, но GL_LINEAR выдает более плавно меняющиеся результаты.

При увеличении, даже при наличии MIP-карт, используется только карта базового уровня. При миниатюризации вы можете уточнить метод фильтрации так, чтобы использовать наиболее подходящую карту или две карты, как описано в следующем абзаце. (Если для уменьшения указано значение GL_NEAREST или GL_LINEAR, используется только карта базового уровня.)

Как видно из табл. 9.3, четыре другие настройки фильтрации доступны только при миниатюризации. В пределах отдельной MIP-карты вы вправе выбрать установку в значение ближайшего текселя — GL_NEAREST_MIPMAP_NEAREST или линейную интерполяцию — GL_LINEAR_MIPMAP_NEAREST. Первый метод быстрее, но результаты далеки от желаемых. Конкретная карта выбирается исходя из степени требуемой миниатюризации и точки переключения между картами. Во избежание резких переходов используйте GL_NEAREST_MIPMAP_LINEAR или GL_LINEAR_MIPMAP_LINEAR с целью линейной интерполяции значений двух ближайших текселов. В варианте GL_NEAREST_MIPMAP_LINEAR выбирается тексел, ближайший в каждой из двух карт, и затем эти два значения линейно интерполируются. В случае GL_LINEAR_MIPMAP_LINEAR линейная интерполяция сначала используется для вычисления в каждой из двух карт, и затем она применяется к двум промежуточным значениям. Как и следовало ожидать, метод GL_LINEAR_MIPMAP_LINEAR обычно производит результаты с лучшим качеством по разрешению, но он требует больше вычислений и, соответственно, более медленный.

ВНИМАНИЕ

Если вы запросили фильтрацию для множественного отображения, но не подготовили полный и непротиворечивый набор карт (все изображения текстур с корректными размерами между уровнями GL_TEXTURE_BASE_LEVEL и GL_TEXTURE_MAX_LEVEL), OpenGL неявно выключит текстурирование без генерации каких-либо ошибок. Если вы пытаетесь использовать множественное отображение, но текстурирования не происходит, убедитесь в наличии изображений текстур для всех уровней детализации.

Некоторые из этих фильтров известны под более привычными названиями. Метод GL_NEAREST часто называют *выборкой по точке* (point sampling), а GL_LINEAR — *билинейной выборкой* (bilinear sampling), поскольку в фильтрации для двухмерных текстур участвует массив 2 × 2 тексела.

Вариант GL_LINEAR_MIPMAP_LINEAR иногда называют «*трехлинейной*» выборкой (trilinear sampling), так как здесь имеет место линейное среднее по двум билинейным выборкам.

ПРИМЕЧАНИЕ

Точка переключения от уменьшения к увеличению обычно соответствует $\lambda = 0.....0$, но на нее влияет тип выбранного фильтра миниатюризации. Если увеличение выполняется согласно GL_LINEAR, а фильтр уменьшения —GL_NEAREST_MIPMAP_NEAREST и GL_NEAREST_MIPMAP_LINEAR, тогда переключение между фильтрами выполняется для $\lambda = 0.5$. Это предотвращает несоответствие внешнего вида маленьких текстур с их большими эквивалентами.

Пособие по текстурам от Нэйта Робинса

Если вы загрузили комплект обучающих программ Нэйта Робинса, запустите сейчас пособие по текстурам. (За справкой о том, где взять эти программы, обращайтесь к разделу «Обучающие программы по OpenGL от Нэйта Робинса».) Она поможет вам в экспериментах с фильтрацией при наложении текстур, и вы увидите наглядно разницу между `GL_NEAREST` и `GL_LINEAR`.

Объекты текстур

Объект текстуры хранит ее данные и делает их легко доступными. Вы можете управлять множеством текстур и возвращаться к ранее загруженным текстурам. Использование объектов текстур — обычно самый быстрый способ применения текстур, дающий хорошие результаты по производительности, поскольку практически всегда быстрее прикрепить (для многократного использования) к текстуре объект, чем непосредственно перезагружать изображение текстуры с помощью `glTexImage*D()`.

Также некоторые реализации поддерживают ограниченный по количеству *рабочий набор* (working set) высокоэффективных текстур. Вы можете использовать объекты текстур для загрузки наиболее часто востребуемых вами текстур в эту область данных. Чтобы использовать объекты текстур для своих данных текстур, выполните следующие действия.

1. Сгенерируйте имена текстур.
2. Свяжите (создайте их) объекты текстур с данными текстур, включая массивы изображений и свойства текстур.
3. Если ваша реализация поддерживает рабочий набор высокопроизводительных текстур, поглядите, достаточно ли в нем места для ваших объектов. Если свободного пространства мало, вы можете назначить приоритеты объектам текстур так, чтобы наиболее часто востребуемые постоянно присутствовали в рабочем наборе.
4. Открепляйте и повторно прикрепляйте объекты текстур, делая их данные доступными в нужный момент для отображения текстурируемых моделей.

Присвоение имен объектам текстур

В качестве «имени» текстуры может быть использовано любое целое беззнаковое число. Чтобы избежать дубликатов имен, обеспечьте их согласование между собой с помощью команды `glGenTextures()`.

void `glGenTextures`(GLsizei *n*, GLuint **textureNames*);
Возвращает *n* неиспользуемых на данный момент имен для объектов текстур в массиве *textureNames*. Значения в этом массиве могут не быть набором последовательных целых.

Имена в массиве *textureNames* помечаются как используемые, причем они наследуют состояние текстуры и ее размерность (1D, 2D или 3D) только при первом использовании.

Нуль — зарезервированное имя, которое никогда не возвращается функцией `glGenTextures()`.

Команда `glIsTexture()` выясняет, используется ли имя текстуры. Если возвращенное командой `glGenTextures()` имя еще не было закреплено за объектом (вызовом `glBindTexture()`) с именем хотя бы раз, то `glIsTexture()` возвращает `GL_FALSE`.

GLboolean `glIsTexture`(GLuint *textureName*);

Возвращает `GL_TRUE`, если *textureName* оказывается именем текстуры, которое было назначено текстуре и впоследствии не высвобождено, и возвращает `GL_FALSE`, когда *textureName* равно нулю или это имя ненулевое и отсутствует среди выделенных текстурам имен.

Создание и использование объектов текстур

Создает и использует объекты текстур одна и та же команда `glBindTexture()`. Если имя текстуры уже было прикреплено (посредством `glBindTexture()`), объект новой текстуры создается со значениями по умолчанию для параметров изображения текстуры и ее свойств. Последующие вызовы `glTexImage*()`, `glTexSubImage*()`, `glCopyTexImage*()`, `glCopyTexSubImage*()`, `glTexParameter*()` и `glPrioritizeTextures()` сохраняют свои данные в объекте текстуры. Объект текстуры может содержать изображение текстуры и связанные и множественные изображения (если надо), включая информацию о ширине, высоте, толщине границ, внутреннем формате, размерности компонентов и свойствах текстуры. В сохраненных свойствах в том числе указываются фильтры увеличения и уменьшения, режимы обработки границ (wrapping), цвет границ и приоритет текстуры.

Когда в дальнейшем объект текстуры связывается повторно, его данные становятся текущим состоянием структуры. (Замещают собой текущее состояние.)

void `glBindTexture`(GLenum *target*, GLuint *textureName*);

Делает три вещи. Когда для *textureName* используется ненулевое беззнаковое целое число, причем в первый раз, создается новый объект текстуры и ему присваивается это имя. Если обнаруживается факт предыдущего связывания имени и объекта текстуры, этот объект становится активным. Когда указывается нулевое значение аргумента *textureName*, OpenGL останавливает использование объектов текстур и возвращает неименованную текстуру по умолчанию.

Если объект текстуры связывается в первый раз (то есть создается), ему назначается размерность целевой текстуры *target*, выбираемая из `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` или `GL_TEXTURE_CUBE_MAP`. Сразу после первичного связывания состояние объекта текстуры идентично состоянию текстуры по умолчанию размерности *target*. В этом начальном состоянии свойства текстур, такие как фильтры увеличения и уменьшения, режимы обертки, цвет границы и приоритет текстуры, приобретают значения по умолчанию.

В программе из листинга 9.7 в теле процедуры `init()` создаются два объекта текстур. В процедуре `display()` каждый из них привлекается к воспроизведению различных четырехсторонних многоугольников.

Листинг 9.7. Прикрепление объектов текстур: texbind.c

```
#define checkImageWidth 64
#define checkImageHeight 64

static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLubyte otherImage[checkImageHeight][checkImageWidth][4];
static GLuint texName[2];

void makeCheckImages(void)
{
   int i, j, c;

   for (i = 0; i < checkImageHeight; i++) {
      for (j = 0; j < checkImageWidth; j++) {
         c = (((i&0x8)==0)^((j&0x8)==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
         checkImage[i][j][3] = (GLubyte) 255;
         c = (((i&0x10)==0)^((j&0x10)==0))*255;
         otherImage[i][j][0] = (GLubyte) c;
         otherImage[i][j][1] = (GLubyte) 0;
         otherImage[i][j][2] = (GLubyte) 0;
         otherImage[i][j][3] = (GLubyte) 255;
      }
   }
}

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glEnable(GL_DEPTH_TEST);

   makeCheckImages();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

   glGenTextures(2, texName);
   glBindTexture(GL_TEXTURE_2D, texName[0]);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
      checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, checkImage);

   glBindTexture(GL_TEXTURE_2D, texName[1]);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

```
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
    checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, otherImage);
  glEnable(GL_TEXTURE_2D);
}
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glBindTexture(GL_TEXTURE_2D, texName[0]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
  glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
  glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
  glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
  glEnd();
  glBindTexture(GL_TEXTURE_2D, texName[1]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
  glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
  glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
  glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
  glEnd();
  glFlush();
}
```

После каждого повторно связывания объекта текстуры вы можете «редактировать» его содержимое. Любые команды, вызываемые вами для изменения объекта изображения и его свойств, изменяют содержимое текущего объекта, за которым закреплено данное имя, равно как состояние текущей текстуры.

В листинге 9.7, после завершения отображения `display()`, ваши объекты все еще связаны с именами, содержащимися в *texName*[1]. Будьте осторожны, чтобы не вызвать по невнимательности подпрограмму, которая изменит данные вашего объекта текстуры.

Если используются множественные отображения, все связанные с одним изображением текстуры карты должны быть помещены в один объект текстуры. В листинге 9.5 изображения текстур уровней 0–5 объединены в объекте с именем *texName*.

Удаление объектов текстур

Фиксируете ли вы за текстурами имена или разъединяете их с объектами текстур, данные текстур все еще остаются где-то среди выделенных ресурсов. Если ресурсы ограничены, удаление текстур может быть единственным способом высвобождения памяти.

void `glDeleteTextures`(GLsizei *n*, const GLuint **textureNames*);
Удаляет *n* объектов текстур, соответствующих именам в массиве *textureNames*. Открепленные имена могут быть использованы повторно (например, командой `glGenTextures()`).

Если удаляется связанная текущая текстура, это равносильно возврату к текстуре по умолчанию, как если бы была вызвана команда `glBindTexture()` с нулем вместо указателя на *textureName*. Попытки удалить несуществующие имена или нулевое имя игнорируются без генерации ошибки.

Рабочий набор резидентных текстур

Некоторые реализации OpenGL поддерживают рабочий набор высокопроизводительных текстур, которые называются *резидентными* (resident). Обычно эти реализации нацелены на специальные аппаратные средства, выполняющие операции над текстурами, и на аппаратный кэш фиксированного объема для хранения изображений текстур. В этом случае использование объектов текстур даже больше, чем рекомендуется, так как вы в состоянии загрузить много текстур в рабочий набор и затем управлять ими.

Если совокупность текстур, требующихся приложению, не умещается целиком в кэш, часть текстур не может быть резидентна. Если вы хотите проследить судьбы ваших резидентных структур, используйте команду `glGetTexParameter*v()` для получения признака, указывающего на состояние резидентности — `GL_TEXTURE_RESIDENT`. Если вас интересует судьба сразу многих резидентных сразу, используйте команду `glAreTexturesResident()`.

GLboolean `glAreTexturesResident`(GLsizei *n*, const GLuint **textureNames*, GLboolean **residences*);

Запрашивает статус резидентности для *n* объектов текстур, согласно их именам в массиве *textureNames*. Аргумент *residences* — это массив, в который возвращаются признаки резидентности в порядке следования имен объектов в массиве *textureNames*. Если оказывается, что резидентны все интересующие вас объекты, функция возвращает `GL_TRUE`, а содержимое массива *residences* не изменяется. Если хотя бы одна из указанных текстур не резидентна, возвращается `GL_FALSE`, а элементы массива *residences*, соответствующие нерезидентным объектам текстур в порядке их перечисления в *textureNames*, также устанавливаются в `GL_FALSE`.

Заметьте, что команда `glAreTexturesResident()` возвращает текущий статус резидентности. Ресурсы текстур очень динамичны, и статусы могут обратиться в любое время. Отдельные реализации кэшируют текстуры при их первом использовании. И, может быть, иногда стоит нарисовать текстуру перед тем, как проверять ее на резидентность.

Если ваша реализация OpenGL не предоставляет рабочего набора текстур, объекты текстур всегда полагаются резидентными. В этом случае команда `glAreTexturesResident()` всегда выдает `GL_TRUE`, а практической пользы от возвращаемой информации нет.

Стратегии резидентности

Если вы можете создать рабочий набор текстур и хотите на этом выиграть в производительности, вы должны опереться на специфику ваших реализаций

OpenGL и приложения. Например, при визуальном моделировании или в видеоиграх вы должны отдавать предпочтение производительности во всех ситуациях. Тогда вам не стоит никогда обращаться к нерезидентным текстурам. Для таких приложений вам следует загрузить все свои текстуры при инициализации и сделать их все резидентными. При отсутствии достаточного количества ресурсов может потребоваться пожертвовать размерами, разрешением, глубиной детализации MIP-изображений, или вы можете использовать команду `glTexSubImage*()` для повторного использования одной и той же памяти, выделенной под текстуры.

ПРИМЕЧАНИЕ

Если у вас есть несколько недолгоживущих текстур одинакового размера, вы можете использовать команду glTexSubImage*() для перезагрузки существующих объектов текстур различными изображениями. Эта техника «зачистки» бывает эффективнее удаления текстур и их создания заново.

Для приложений, создающих текстуры «на лету», нерезидентные текстуры — неизбежность. Если какие-то текстуры востребуются чаще других, вы можете сделать их объекты высокоприоритетными, тем самым увеличив их способность к резидентности. Удаление объектов текстур также освобождает место под другие. И наоборот, назначение объекту текстуры низкого приоритета может поставить ее в ряд первых претендентов на вывод из состава рабочего набора, как неактуального ресурса. Для назначения приоритетов объектам текстур используется команда `glPrioritizeTextures()`.

void `glPrioritizeTextures`(GLsizei *n*, const GLuint **textureNames*, const GLclampf **priorities*);

Для *n* объектов текстур с именами, перечисленными в массиве *textureNames*, присваивает приоритет резидентности согласно элементам массива *priorities*.

Значения в массиве *priorities* ограничиваются диапазоном [0.0, 1.0]. Нуль — это самый низкий приоритет (такие текстуры в первую очередь превращаются в нерезидентные), а 1 — наивысший приоритет.

Команда не требует, чтобы все имена *textureNames* были связаны с объектами текстур. Соответственно, приоритеты никак не сказываются в отношении пока еще не связанных объектов.

Приоритет одиночной текстуре можно также присвоить командой `glTexParameter*()`, но только если она текущая и связанная. Фактически команда `glTexParameter*()` — это лишь способ установки приоритета текстуре по умолчанию.

В случае равенства приоритетов текстур обычные реализации OpenGL применяют стратегию самого давнего использования (least recently used, LRU) в решении об исключении объекта из рабочего набора. Если вы уверены, что вашей реализации OpenGL присущ именно этот алгоритм, присвойте всем объектам текстур равные приоритеты, тем самым вы получите благоразумную LRU-систему для перераспределения ресурсов текстур.

Если же ваша реализация OpenGL не использует принцип LRU для объектов текстур с равными приоритетами (или вы вообще не знаете, какой принцип в нее

заложен), воспроизведите LRU-стратегию и тщательное управление приоритетами собственными силами. Когда текстура используется (связана), вы можете поднять ее приоритет до максимального, отражая тем самым ее недавнее использование. Затем через регулярные промежутки времени вы можете снижать ее приоритет.

ПРИМЕЧАНИЕ

Вероятны проблемы с фрагментацией памяти текстур, особенно если вы удаляете и заново создаете множество текстур. Хотя возможно загрузить все объекты текстур в рабочий набор, связывая их в определенной последовательности, последующее связывание их в другой последовательности способно привести к тому, что некоторые текстуры станут нерезидентными.

Функции текстур

Во всех примерах, представленных ранее в этой главе, значения в картах текстур использовались в качестве цветов, которыми окрашивались отображаемые поверхности. Вы можете использовать эти значения также для модуляции цвета, которым поверхность была бы отрисована в отсутствие текстурирования, или для комбинирования цветов карты с оригинальными цветами поверхности. Вы выбираете функцию текстурирования, передавая соответствующие аргументы в процедуру `glTexEnv*()`.

void `glTexEnv{if}`(GLenum *target*, GLenum *pname*, *TYPE param*);
void `glTexEnv{if}v`(GLenum *target*, GLenum *pname*, *TYPE *param*);
Устанавливает текущую функцию текстурирования. Аргумент *target* выбирается из `GL_TEXTURE_FILTER_CONTROL` и `GL_TEXTURE_ENV`.
Если значение *target* — `GL_TEXTURE_FILTER_CONTROL`, аргумент *pname* должен быть задан как `GL_TEXTURE_LOD_BIAS`, а *param* должен быть одиночным значением с плавающей запятой, вносящим поправку (bias) в параметр уровня детализации отображения.
Если *target* — `GL_TEXTURE_ENV` и если *pname* — `GL_TEXTURE_ENV_MODE`, тогда *param* выбирается из `GL_DECAL`, `GL_REPLACE`, `GL_MODULATE`, `GL_BLEND`, `GL_ADD` или `GL_COMBINE`. Эти значения задают способ использования цветов обрабатываемых фрагментов. Если *pname* — `GL_TEXTURE_ENV_COLOR`, аргумент *param* представляет собой массив из четырех чисел в формате с плавающей запятой (R, G, B, A), образующих цвет, используемый в режиме `GL_BLEND`.

ПРИМЕЧАНИЕ

Это только неполный список допустимых для glTexEnv*() значений. За всеми подробностями насчет GL_COMBINE и других настроек аргументов pname и param обращайтесь к разделу «Комбинирование функций текстур» и табл. 9.8.

Сумма функции текстурирования и основного внутреннего формата отвечает за то, как применяется каждый компонент текстуры. Функция текстурирования оперирует выбранными компонентами, а также значениями цвета в отсутствие текстурирования. (Заметьте, что выбор делается после применения операций при передаче пикселов.) Когда вы указываете свою карту текстуры с помощью `glTexImage*D()`, в третьем аргументе сообщается, как будет внутри системы хра-

ниться каждый тексел. Имеется шесть основных внутренних форматов: GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_INTENSITY, GL_RGB и GL_RGBA. Другие, такие как GL_LUMINANCE6_ALPHA2 или GL_R3_G3_B2, задают желаемую размерность компонентов текстуры и могут быть сопоставлены одному из этих шести базовых форматов.

Вычисления по текстурированию в конечном счете выполняются над RGBA-компонентами, но некоторые внутренние форматы не относятся к модели RGB. В табл. 9.4 показано, как RGBA-значения цвета образуются для различных форматов текстур, включая в рассмотрение далеко не очевидные производные.

Таблица 9.4. Производные значения цвета для различных форматов

Основной внутренний формат	Вторичный цвет источника (R, G, B, A)
GL_ALPHA	(0, 0, 0, A)
GL_LUMINANCE	(L, L, L, 1)
GL_LUMINANCE_ALPHA	(L, L, L, A)
GL_INTENSITY	(I, I, I, I)
GL_RGB	(R, G, B, 1)
GL_RGBA	(R, G, B, A)

Таблицы 9.5 и 9.6 показывают, какой вклад вносят функция текстурирования (за исключением GL_COMBINE) и базовый внутренний формат в формулу, используемую для каждого компонента текстуры.

В табл. 9.5 и 9.6 приняты следующие обозначения нижних индексов:

- s — цвет исходной текстуры, согласно табл. 9.4;
- f — значение для поступившего на обработку фрагмента;
- c — значения, связанные с GL_TEXTURE_ENV_COLOR;
- отсутствие нижнего индекса — конечное вычисленное значение.

В этих таблицах под умножением триплета цветов на скалярные значения понимается умножение каждого из RGB-компонентов на скалярную величину; перемножение (или сложение) двух цветов из триады означает умножение (сложение) каждого компонента второй тройки на (с) соответствующий компонент первой.

Таблица 9.5. Функции текстур, предназначенные для замещения, модуляции и техники деколи

Основной внутренний формат	GL_REPLACE	GL_MODULATE	GL_DECAL
GL_ALPHA	$C = C_f$; $A = A_s$	$C = C_f$; $A = A_f A_s$	Не определено
GL_LUMINANCE	$C = C_s$; $A = A_f$	$C = C_f C_s$; $A = A_f$	Не определено
GL_LUMINANCE_ALPHA	$C = C_s$; $A = A_s$	$C = C_f C_s$; $A = A_f A_s$	Не определено
GL_INTENSITY	$C = C_s$; $A = C_s$	$C = C_f C_s$; $A = A_f C_s$	Не определено
GL_RGB	$C = C_s$; $A = A_f$	$C = C_f C_s$; $A = A_f$	$C = C_s$; $A = A_f$
GL_RGBA	$C = C_s$; $A = A_s$	$C = C_f C_s$; $A = A_f A_s$	$C = C_f (1 - A_s) + C_s A_s$; $A = A_f$

Таблица 9.6. Аддитивная функция и функция для смешивания

Основной внутренний формат	GL_BLEND	GL_ADD
GL_ALPHA	$C = C_f$; $A = A_f A_s$	$C = C_f$; $A = A_f A_s$
GL_LUMINANCE	$C_f(1 - C_s) + C_c C_s$; $A = A_f$	$C = C_f + C_s$; $A = A_f$
GL_LUMINANCE_ALPHA	$C = C_f(1 - C_s) + C_c C_s$; $A = A_f A_s$	$C = C_f + C_s$; $A = A_f A_s$
GL_INTENSITY	$C = C_f(1 - C_s) + C_c C_s$; $A = A_f(1 - A_s) + A_c A_s$	$C = C_f + C_s$; $A = A_f + A_s$
GL_RGB	$C = C_f(1 - C_s) + C_c C_s$; $A = A_f$	$C = C_f + C_s$; $A = A_f$
GL_RGBA	$C = C_f(1 - C_s) + C_c C_s$; $A = A_f A_s$	$C = C_f + C_s$; $A = A_f A_s$

Функция с замещением просто берет цвет, который был бы использован для окраски в отсутствие какого-либо наложения текстур (цвет фрагмента), попросту отбрасывает его и замещает цветом текстуры. Вы используете функцию замещения в ситуациях, когда к объекту нужно применить непрозрачную текстуру — например, при рисовании объекта с непрозрачной этикеткой.

Функция деколи (decal) напоминает случай замещения с той разницей, что она работает только для внутренних форматов RGB и RGBA и обрабатывает альфа-компонент иначе. В варианте с RGBA цвета фрагментов смешиваются с цветом текстуры в пропорции, определяемой альфа-компонентом текстуры, а альфа-компонент фрагментов не изменяется. Данная функция может быть использована для взвешенной по альфа-каналу текстуры, такой как эмблема на крыле самолета.

При модуляции цвета фрагментов модулируются содержимым текстуры. Если в качестве базового внутреннего формата выбран `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA` или `GL_INTENSITY`, значения цвета умножаются на некоторое значение, так что карта текстуры выводит производную по определенному закону от значения цвета фрагмента (если яркость или интенсивность равна 1) до черного (если это 0). Для внутренних форматов `GL_RGB` и `GL_RGBA` каждый компонент входящего цвета умножается на соответствующее (возможно, другое) значение текстуры. Если это значение альфа-компонента, оно умножается на значение альфа-компонента фрагмента. Модуляция особенно хороша в условиях освещения, так как цвет освещенного многоугольника может быть использован для затухания цвета текстуры. Большинство примеров, иллюстрирующих наложение текстур, наносят «покрытие» с помощью модуляции именно по этой причине. Белые, «зеркальные» многоугольники часто используются для воспроизведения освещенных текстурированных объектов, а изображение текстуры обеспечивает рассеивание цвета.

Аддитивная функция просто складывает цвет текстуры с цветом фрагмента. Если это значение альфа-компонента, оно умножается на значение альфа-компонента фрагмента, кроме случая формата `GL_INTENSITY`, где складываются интенсивность текстуры и значение альфа-канала фрагмента. Если не отнестись со всем вниманием к выбору такой функции, можно получить в результате пересыщенные или неправильные тона.

Функция смешивания единственная из всех использует цвет, заданный параметром GL_TEXTURE_ENV_COLOR. Использование значения яркости, интенсивности или цвета отчасти напоминает использование значения альфа-компонента для смешивания цвета фрагмента с GL_TEXTURE_ENV_COLOR. (См. «Пример смешивания» в главе 6, демонстрирующий технику билбординга (billboarding).)

Пособие по текстурам от Нэйта Робинса

Если вы загрузили пакет программ самоучителя Нэйта Робинса, запустите пособие по текстурам. Изменяйте атрибуты среды наложения текстур и смотрите на эффект, производимый различными функциями текстур. Когда дело дойдет до GL_MODULATE, обратите внимание на эффект для цвета, указанного `glColor4f()`. Экспериментируя с GL_BLEND, смотрите, что получится при изменении цвета, задаваемого массивом *env_color*.

Присвоение координат текстур

Отображая свою сцену с наложением текстур, вы должны подготовить координаты объекта и координаты текстур для каждой вершины. После преобразования координаты объекта определяют местоположение вершин на экране. Координаты текстуры отвечают за то, какой тексел в карте текстуры будет связан с каждой вершиной. Подобным образом происходит интерполяция цвета между двумя вершинами тонированного многоугольника или линии, и также интерполируются координаты текстуры. (Вспомните, что текстура — это обычный прямоугольный массив данных.)

Координаты текстуры могут вмещать в себя одну, две, три или четыре координаты. Обращение к ним производится обычно как к *s*-, *t*-, *r*- и *q*-координатам с целью их различения от координат объекта (*x*, *y*, *z* и *w*) и параметрических координат (*u* и *v*; см. главу 12). Для одномерных текстур вы используете *s*-координату; для двухмерных — *s* и *t*; а для трехмерных — *s*, *t* и *r*. Координата *q*, подобная *w*, обычно задается значением 1 и может быть использована для получения однородных координат; это описано в разделе «Координата *q*». Команда `glTexCoord*()`, задающая координаты текстуры, аналогична `glVertex*()`, `glColor*()` и `glNormal*()` — все эти разновидности команд используются одинаково между `glBegin()` и `glEnd()`. Обычно диапазон значений координат текстур ограничен 0 и 1; тем не менее возможны и выходящие за этот диапазон значения, как это описано в разделе «Повторяемые и обрезанные текстуры».

void `glTexCoord`{1234}{sifd}(*TYPE coords*);
void `glTexCoord`{1234}{sifd}`v`(*TYPE *coords*);
Устанавливает координаты текущей текстуры (*s*, *t*, *r*, *q*). При последующих вызовах `glVertex*()` вершины будут связаны с координатами текстуры. Команда `glTexCoord1*()` устанавливает значение *s*-координаты, *t* и *r* сбрасываются в 0, а *q* устанавливается единицей. С помощью команды `glTexCoord2*()` вы задаете *s* и *t*; *r* и *q* устанавливаются в 0 и 1 соответственно. Команда `glTexCoord3*()` устанавливает *q* в 1, а остальные инициируются согласно за-

данным значениям. Задать явно все координаты сразу позволяет команда `glTexCoord4*()`. Используйте соответствующий суффикс (`s`, `i`, `f` или `d`) и значение типа *TYPE* (`GLshort`, `GLint`, `GLfloat` или `GLdouble`) для уточнения типа данных координат. Вы можете подготовить все координаты по отдельности, а можете и использовать векторную версию команды, чтобы сделать это из одного массива. Координаты текстуры умножаются на 4 × 4 матрицу текстуры перед тем, как будет выполнено наложение текстур. (См. раздел «Стек матриц текстур».) Обратите внимание, что целочисленные координаты текстуры интерпретируются непосредственно, а не проецируются в диапазон [–1, 1].

В следующем разделе объясняется, как рассчитать соответствующие координаты текстур. Вместо явного их присвоения вами, можно воспользоваться средствами OpenGL для автоматического их вычисления. (См. раздел «Автоматическая генерация координат текстур».)

Пособие по текстурам от Нэйта Робинса

При наличии у вас самоучителя Нэйта Робинса запустите пособие по текстурам и экспериментируйте с параметрами `glTexCoord2f()` для четырех различных вершин. Посмотрите, как можно наложить часть полной текстуры. Что случится, если вы сделаете координаты текстуры меньше 0 или больше 1?

Вычисление подходящих координат текстуры

Двухмерные текстуры — это квадратные или прямоугольные изображения, которые обычно проецируются на многоугольники, образующие полигональную (polygon – многоугольник) модель. В простейшем случае вы отображаете прямоугольник текстуры в такую же «прямоугольную» модель — например, вашей текстурой может быть сканированное изображение кирпичной стены, и ваш прямоугольник может представлять один кирпич в кладке. Для простоты предположим, что кирпичная стена имеет квадратную форму и текстура тоже квадратная, и вы желаете наложить всю текстуру на всю стену. Координаты текстуры — (0, 0), (1, 0), (1, 1) и (0, 1) обходятся против часовой стрелки. Рисуя стену, просто задайте эти координаты как координаты текстуры, когда указываете координаты вершин стены, в порядке против часовой стрелки.

Теперь усложним. Пусть отношение высоты стены к ее ширине — два к трем, но текстура все еще квадратная. Чтобы избежать искажения текстуры, вам следует спроецировать на стену часть карты текстуры так, чтобы сохранились пропорции. Допустим, что вы решили использовать нижние две трети карты для текстурирования стены. В таком случае используйте координаты текстуры (0, 0), (1, 0), (1, 2/3) и (0, 2/3), раз вершины стены обходятся против часовой стрелки.

Внесем еще немного сложности. Будем считать, что вас почему-то интересует отображение на экран консервной жестяной банки с наклеенной на нее этикеткой. Чтобы получить текстуру, вы сходили за банкой в магазин, отодрали этикетку и отсканировали ее. Предположим, что этикетка имеет 4 условные единицы длины поперек и 12 единиц вдоль, то есть соотношение сторон — 3 к 1. Поскольку текстуры должны удовлетворять соотношению 2^n к 1, вы можете просто не ис-

пользовать верхнюю треть этикетки или вырезать из нее кусок с необходимыми пропорциями. Допустим, вы отказались от использования верхней трети. Далее введем допущение, что цилиндр банки аппроксимируется 30 прямоугольниками с длиной 4 единицы (высота банки) и шириной 12/30 (1/30 периметра крышки). Вы можете использовать следующие координаты текстур для 30 аппроксимирующих прямоугольников:

1: (0, 0), (1/30, 0), (1/30, 2/3), (0, 2/3)

2: (1/30, 0), (2/30, 0), (2/30, 2/3), (1/30, 2/3)

3: (2/30, 0), (3/30, 0), (3/30, 2/3), (2/30, 2/3)

...

30: (29/30, 0), (1, 0), (1, 2/3), (29/30, 2/3)

Только несколько кривых поверхностей, таких как конусы и цилиндры, могут быть спроецированы на плоскую поверхность без геодезических искажений. Любые другие фигуры претерпят деформацию. В целом, чем выше скругленность (curvature) поверхности, тем больше искажается текстура.

Если вас не тревожит искажение текстуры, найти простое рациональное решение отображения несложно. Так, рассмотрим сферу с координатами поверхности $(\cos\theta\cos\varphi, \cos\theta\sin\varphi, \sin\theta)$, где $0 \le \theta \le 2\pi$ и $0 \le \varphi \le \pi$. Карта текстуры может быть напрямую спроецирована в прямоугольник θ-φ, но по мере приближения к полюсам все сильнее будут проявляться аномалии. Весь верхний край карты текстуры попадет на «северный» полюс, и весь нижний край — на «южный» полюс. Для других поверхностей, таких как тор («бублик») с большим отверстием, координаты поверхности преобразуются к координатам текстуры способом, который вносит лишь чуть-чуть искажений, поэтому он годится для многих приложений. На рис. 9.9 показаны два тора, один с небольшим отверстием (и, следовательно, со значительной деформацией вблизи центра) и один с дырой чуть ли не до краев (и потому с незначительными искажениями).

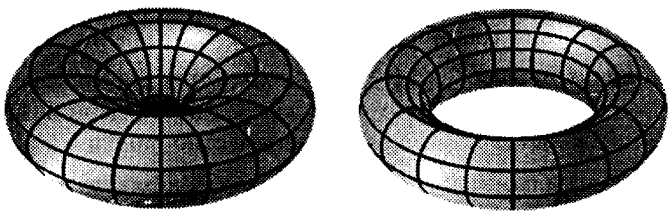


Рис. 9.9. Искажение карты текстуры

Если вы текстурируете сплайновые поверхности, сгенерированные вычислителями (см. главу 12), параметры u и v поверхности могут в отдельных случаях использоваться как координаты текстуры. Тем не менее в целом успешное наложение текстур на полигональные приближения кривых поверхностей требует от вас высокого мастерства.

Повторяемые и обрезанные текстуры

Вы вправе задать координаты текстур, выходящие за диапазон [0, 1], и затем «выпилить» этот диапазон или же тиражировать карту текстуры. Второй вариант подходит тогда, когда вы имеете, например, изображение большого самолета с координатами текстуры от 0.0 до 10.0 в двух направлениях. Тогда вы получаете 100 копий текстуры, сопряженных на экране в плиточную поверхность. При размножении текстур целые части координат текстур игнорируются, благодаря чему копии укладываются рядом, не перекрываясь. Для большинства приложений, использующих повторение текстур, текселы наверху текстуры должны соответствовать пикселам снизу, и то же верно для правого и левого краев.

Допускается «зеркальное» расположение, где плитки чередуются в порядке «туда-сюда» (flip-flop), другими словами — наподобие триггерного переключения. Например, в пределах диапазона координат текстур [0, 1] текстура может выглядеть ориентированной слева направо (или сверху вниз или от ближнего конца к дальнему), но здесь слово «зеркальное» обозначает смену ориентации текстуры в направлении «справа налево» для диапазона [1, 2], снова «слева направо» для координат [2, 3] и т. д.

Другая возможность состоит в ограничивании (clamp) координат текстур: любые значения большие 1.0 устанавливаются в 1.0, и любые значения меньше 0.0 устанавливаются в 0.0. Такое вырезание полезно для приложения, где вам нужна одна копия текстуры на большую поверхность. Если диапазон координат текстуры для поверхности охватывает значения от 0.0 до 10.0 в обоих направлениях, в левом нижнем углу поверхности появится только один экземпляр текстуры.

Если вы используете текстуры с границами или задали для границы определенный цвет, на интерпретацию информации границ будут влиять как режим обработки границ (wrapping), так и метод фильтрации (см. раздел «Фильтрация»).

Если фильтрация выполняется по принципу `GL_NEAREST`, используется ближайший тексел текстуры. Для большинства режимов наложения граница (или ее цвет) игнорируется. Тем не менее, если координаты текстур выходят за рамки [0, 1] и задан режим наложения `GL_CLAMP_TO_BORDER`, выбирается ближайший тексел на границе. (При отсутствии границы используется некоторый константный цвет.)

При фильтрации по методу `GL_LINEAR` используется взвешенное значение по массиву 2 × 2 (для двухмерных текстур) данных цвета. При задании границы и/или ее цвета оба свойства применяются совместно, по следующим правилам:

- в режиме обработки границ `GL_REPEAT` граница игнорируется во всех ситуациях. К вычислениям привлекается массив 2 × 2 тексела с противоположного края текстуры. Таким образом, текселы с правого края усредняются с пикселами левого края, и то же верно для левого и правого краев;
- в режиме `GL_CLAMP` для тексела на границе (или в режиме `GL_TEXTURE_BORDER_COLOR`) используется средневзвешенное по массиву 2 × 2 тексела;
- в режиме `GL_CLAMP_TO_EDGE` граница игнорируется всегда. Для вычислений привлекаются текселы вблизи краев текстуры;

- в режиме GL_CLAMP_TO_BORDER, если координаты текстуры выходят за диапазон [0, 1], в расчет принимаются только текселы границы (при отсутствии границы основываются на константном цвете границы). Для близких к краевым координатам текстуры в массив 2 × 2 тексела могут быть выбраны текселы как с границы, так и из внутренней области текстуры.

Если вы используете ограничивание, вам следует избегать краевых эффектов при взаимодействии текстуры и поверхности. Для этого используйте значения альфа-компонента 0 для краев (или границ, если они заданы) текстуры. Функция деколи непосредственно использует значения альфа-компонента текстуры в своих вычислениях. Если вы используете несколько функций текстур, может потребоваться также включить смешивание с подобающими факторами влияния для источника и приемника. (См. раздел «Смешивание» главы 6.)

Чтобы увидеть эффект от обработки границ, вам следует рискнуть и задать координаты вне диапазона [0.0, 1.0]. Для начала возьмите пример из листинга 9.1 и измените координаты текстуры для квадратов с 0.0 на 4.0, как это сделано ниже:

```
glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
   glTexCoord2f(0.0, 4.0); glVertex3f(-2.0, 1.0, 0.0);
   glTexCoord2f(4.0, 4.0); glVertex3f(0.0, 1.0, 0.0);
   glTexCoord2f(4.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
   glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
   glTexCoord2f(0.0, 4.0); glVertex3f(1.0, 1.0, 0.0);
   glTexCoord2f(4.0, 4.0); glVertex3f(2.41421, 1.0, -1.41421);
   glTexCoord2f(4.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
glEnd();
```

Для режима GL_REPEAT результат продемонстрирован на рис. 9.10.

Здесь текстура дублируется сразу в двух направлениях, *s* и *t*, это делают вызовы glTexParameter*():

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Некоторые реализации OpenGL поддерживают режим GL_MIRRORED_REPEAT, обращающий ориентацию целочисленных координат текстур. На рис. 9.11 показана разница между обыкновенным размножением с переносом (слева на рисунке) и зеркальным повторением (справа).

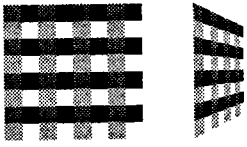


Рис. 9.10. Повторение текстуры

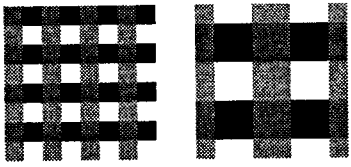


Рис. 9.11. Режим GL_REPEAT в сравнении с GL_MIRRORED_REPEAT

На рис. 9.12 продемонстрировано действие режима GL_CLAMP в применении к обоим направлениям. В местах, где координата *s* или *t* больше единицы, используется тексел, взятый оттуда, где значение координаты текстуры в точности равно 1.

Рис. 9.12. Ограничивание текстуры

Режимы обработки границ по каждому направлению независимы. Вы вправе также применить ограничение в одном измерении и повторение для другого, как показано на рис. 9.13.

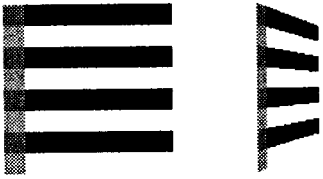


Рис. 9.13. Размножение текстуры вместе с ограничиванием

Теперь, после рассмотрения всех аргументов `glTexParameter*()`, суммируем все в одной подпрограмме.

void `glTexParameter`{if}(GLenum *target*, GLenum *pname*, *TYPE param*);
void `glTexParameter`{if}`v`(GLenum *target*, GLenum *pname*, *TYPE* **param*);
Устанавливает различные параметры, управляющие обработкой текстуры, применением ее к фрагментам и сохранением в объекте текстуры. Аргумент *target* выбирается из `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` или `GL_TEXTURE_CUBE_MAP` в соответствии с характером текстуры. Возможные значения для аргументов *pname* и *param* приведены в табл. 9.7. Вы можете задавать параметры по отдельности обычной версией команды, и в вашем распоряжении также ее векторная форма, работающая с массивом значений для случая `GL_TEXTURE_BORDER_COLOR`. Если значения представлены целыми числами, они преобразуются в формат с плавающей запятой согласно табл. 4.1; кроме того, они ограничиваются диапазоном [0, 1].

Таблица 9.7. Аргументы команды glTexParameter*()

Параметр	Значения
GL_TEXTURE_WRAP_S	GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, GL_REPEAT, GL_MIRRORED_REPEAT
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, GL_REPEAT, GL_MIRRORED_REPEAT
GL_TEXTURE_WRAP_R	GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, GL_REPEAT, GL_MIRRORED_REPEAT
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR

Параметр	Значения
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR
GL_TEXTURE_BORDER_COLOR	Любые четыре числа из диапазона [0,0; 1,0]
GL_TEXTURE_PRIORITY	[0,0; 1,0] для текущего объекта текстуры
GL_TEXTURE_MIN_LOD	Любое значение с плавающей запятой
GL_TEXTURE_MAX_LOD	Любое значение с плавающей запятой
GL_TEXTURE_BASE_LEVEL	Любое неотрицательное целое
GL_TEXTURE_MAX_LEVEL	Любое неотрицательное целое
GL_DEPTH_TEXTURE_MODE	GL_LUMINANCE, GL_INTENSITY, GL_ALPHA
GL_TEXTURE_COMPARE_MODE	GL_NONE, GL_COMPARE_R_TO_TEXTURE
GL_TEXTURE_COMPARE_FUNC	GL_LEQUAL, GL_GEQUAL
GL_GENERATE_MIPMAP	GL_TRUE, GL_FALSE

Упражнение

Рисунки 9.12 и 9.13 получены с использованием GL_NEAREST для фильтров увеличения и миниатюризации. Что же произойдет, если вы измените это значение на GL_LINEAR? Результирующее изображение должно выглядеть не таким резким.

Информация границ может потребоваться во время обсчета текстурирования. Для простейшей демонстрации этого установите GL_TEXTURE_BORDER_COLOR в хорошо заметный цвет. Установив фильтры в GL_NEAREST и режим обработки границ GL_CLAMP_TO_BORDER, оцените эффект от цвета границы текстуры на текстурируемый объект (для диапазона координат текстуры вне [0, 1]). Граница также оказывает воздействие на текстурирование с фильтрами GL_LINEAR и режимом обработки границ GL_CLAMP.

Что теперь случится, если включить режим GL_CLAMP_TO_EDGE или GL_REPEAT? Для обоих случаев цвет границы игнорируется.

Пособие по текстурам от Нэйта Робинса

Запустите пособие по текстурам Нэйта Робинса и оцените влияние параметров GL_REPEAT и GL_CLAMP. Для этого вам может потребоваться сделать текстурные координаты вершин (аргументы glTexCoord2f()) меньше 0 и/или больше 1.

Автоматическая генерация координат текстур

Вы можете использовать наложение текстур для образования контуров своих моделей или для имитации отражения от произвольного окружения для «зеркальной» модели. Чтобы достигнуть этого эффекта, позвольте OpenGL генерировать координаты текстур за вас, вместо явного их задания командой glTexCoord*().

Чтобы сгенерировать координаты текстур автоматически, используйте команду `glTexGen()`.

void `glTexGen`{ifd}(GLenum *coord*, GLenum *pname*, *TYPE param*);
void `glTexGen`{ifd}`v`(GLenum *coord*, GLenum *pname*, *TYPE* **param*);
Задает функции для автоматической генерации координат текстур. Первый аргумент, *coord*, должен быть задан как `GL_S`, `GL_T`, `GL_R` или `GL_Q` согласно типу генерируемой координаты — *s*, *t*, *r* или *q*. Значение аргумента *pname* выбирается из констант `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE` или `GL_EYE_PLANE`. Если это `GL_TEXTURE_GEN_MODE`, аргумент *param* должен быть целым числом (или, для векторной версии команды, должен указывать на целое число), одним из следующих: `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR`, `GL_SPHERE_MAP`, `GL_REFLECTION_MAP` или `GL_NORMAL_MAP`. Эти символические имена определяют, какая функция будет использоваться для порождения координат текстур. Для других допустимых значений *pname* аргумент *param* является указателем на массив значений (векторная версия), задающих параметры генерирования.

Для различных областей применения используются различные методы генерации координат текстур. Указание опорной плоскости (reference plane) в координатах объекта предпочтительно тогда, когда изображение текстуры остается привязанным к движущемуся объекту. Таким образом, константа `GL_OBJECT_LINEAR` может быть использована для покрытия древесными прожилками площадки. Задание опорной плоскости в координатах наблюдателя (`GL_EYE_LINEAR`) — лучший способ для получения контуров линий подвижных объектов. Параметр `GL_EYE_LINEAR` находит применение у специалистов в области геологических наук, которые занимаются нефтедобычей или газодобычей. Так как бур проходит сквозь земную кору, могут понадобиться различные цвета для представления срезов породы с ростом глубины. Параметры `GL_SPHERE_MAP` и `GL_REFLECTION_MAP` главным образом используются для наложения сферообразного окружения, а `GL_NORMAL_MAP` — для «кубических» карт текстур. (См. разделы «„Сферические“ карты» и «„Кубические“ текстуры»).

Создание контуров

Когда заданы параметры `GL_TEXTURE_GEN_MODE` и `GL_OBJECT_LINEAR`, функция генерации является линейной комбинацией значений вершин в координатах объекта (x_o, y_o, z_o, w_o):

сгенерированная координата = $p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$.

Значения p_1,..., p_4 передаются в аргументе *param* команде `glTexGen*v()`, с *pname*, установленным в `GL_OBJECT_PLANE`. С корректно нормализованными p_1,..., p_4 эта функция выдает расстояние от вершины до плоскости. Например, если $p_2 = p_3 = p_4 = 0$ и $p_1 = 1$, это расстояние $x = 0$. Оно положительно с одной стороны плоскости и отрицательно с другой, и равно нулю, когда вершина принадлежит плоскости.

В листинге 9.8 первым делом рисуются равно дистанцированные линии чайника; они указывают расстояние от плоскости $x = 0$. Коэффициенты для такой плоскости помещены в массив:

```
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
```

Поскольку здесь показывается только одно свойство (расстояние от плоскости), достаточно одномерной текстуры. Карта текстуры имеет сплошную зеленую окраску, за исключением красных меток для равноотстоящих интервалов. Так как чайник находится в плоскости xy, все контуры перпендикулярны их основанию. На цветном рис. 18 показана картинка, полученная этой программой. Здесь нажатие клавиши s изменяет параметры плоскости отсчета на

```
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
```

Полосы контуров параллельны плоскости $x + y + z = 0$, и пересекают чайник под углом, как показано на цветном рисунке 18. Чтобы восстановить исходные значения, $x = 0$, нажмите клавишу x.

Листинг 9.8. Автоматическое генерирование координат текстур: texgen.c

```
#define stripeImageWidth 32
GLubyte stripeImage[4*stripeImageWidth];

static GLuint texName;

void makeStripeImage(void)
{
   int j;
   for (j = 0; j < stripeImageWidth; j++) {
      stripeImage[4*j] = (GLubyte) ((j<=4) ? 255: 0);
      stripeImage[4*j+1] = (GLubyte) ((j>4) ? 255: 0);
      stripeImage[4*j+2] = (GLubyte) 0;
      stripeImage[4*j+3] = (GLubyte) 255;
   }
}

/* Плоскости для генерации координат текстур */
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
static GLfloat *currentCoeff;
static GLenum currentPlane;
static GLint currentGenMode;

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_SMOOTH);

   makeStripeImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

   glGenTextures(1, &texName);
   glBindTexture(GL_TEXTURE_1D, texName);
```

продолжение ⇩

Листинг 9.8 (*продолжение*)

```
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
   glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, stripeImageWidth, 0,
      GL_RGBA, GL_UNSIGNED_BYTE, stripeImage);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
   currentCoeff = xequalzero;
   currentGenMode = GL_OBJECT_LINEAR;
   currentPlane = GL_OBJECT_PLANE;
   glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
   glTexGenfv(GL_S, currentPlane, currentCoeff);

   glEnable(GL_TEXTURE_GEN_S);
   glEnable(GL_TEXTURE_1D);
   glEnable(GL_CULL_FACE);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);
   glFrontFace(GL_CW);
   glCullFace(GL_BACK);
   glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix();
   glRotatef(45.0, 0.0, 0.0, 1.0);
   glBindTexture(GL_TEXTURE_1D, texName);
   glutSolidTeapot(2.0);
   glPopMatrix();
   glFlush();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-3.5, 3.5, -3.5*(GLfloat)h/(GLfloat)w,
              3.5*(GLfloat)h/(GLfloat)w, -3.5, 3.5);
   else
      glOrtho(-3.5*(GLfloat)w/(GLfloat)h,
              3.5*(GLfloat)w/(GLfloat)h, -3.5, 3.5, -3.5, 3.5);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 'e':
```

```
      case 'E':
         currentGenMode = GL_EYE_LINEAR;
         currentPlane = GL_EYE_PLANE;
         glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 'o':
      case 'O':
         currentGenMode = GL_OBJECT_LINEAR;
         currentPlane = GL_OBJECT_PLANE;
         glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 's':
      case 'S':
         currentCoeff = slanted;
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 'x':
      case 'X':
         currentCoeff = xequalzero;
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
         default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(256, 256);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Генерация текстур для координаты *s* включается передачей параметра `GL_TEXTURE_GEN_S` в функцию `glEnable()`; для других координат это параметры `GL_TEXTURE_GEN_T`, `GL_TEXTURE_GEN_R` и `GL_TEXTURE_GEN_Q`. Для отключения этого режима

используется команда `glDisable()` с подходящей константой. Также обратите внимание на использование `GL_REPEAT` с целью получения линий контура, повторенных поперек чайника.

Функция текстуры `GL_OBJECT_LINEAR` вычисляет координаты текстуры в системе координат модели. В программе из листинга 9.8 сначала применяется эта функция, так что контуры линий перпендикулярны основанию чайника, без оглядки на то, что чайник может вращаться или измениться точка наблюдения. Однако, если вы нажмете клавишу `e`, вы переключитесь из режима `GL_OBJECT_LINEAR` в режим `GL_EYE_LINEAR`, и контуры будут вычисляться относительно системы координат наблюдателя. (Нажатие клавиши `o` снова возвращает режим `GL_OBJECT_LINEAR`.) Если опорная плоскость $x = 0$, на чайнике рисуются красные полосы, параллельные плоскости yz от точки наблюдения, как показано на цветном рис. 18. Математически это описывается так: вы умножаете вектор $(p_1\ p_2\ p_3\ p_4)$ на обратную модельно-видовую матрицу, чтобы получить значения, используемые в вычислении расстояния до плоскости. Координата текстуры генерируется следующей функцией:

$$\text{сгенерированная координата} = p_1'x_e + p_2'y_e + p_3'z_e + p_4'w_e,$$

где $(p_1'\ p_2'\ p_3'\ p_4') = (p_1\ p_2\ p_3\ p_4)M^{-1}$.

В этом случае (x_e, y_e, z_e, w_e) — координаты вершины в системе наблюдателя, а значения $p_1,\ldots, p_4$ передаются аргументом *param* в процедуру `glTexGen*()`, при этом аргумент *pname* установлен в значение `GL_EYE_PLANE`. Значения со штрихами вычисляются только во время их задания, поэтому данная операция не настолько затратна в вычислительном отношении.

Во всех этих примерах для генерации контуров используется одна координата. Координаты s, t и (если это нужно) r могут быть сгенерированы независимо для того, чтобы обозначить расстояния до двух или трех различных плоскостей. Соответствующе сформированная двух- или трехмерная карта текстуры вырабатывает два или три набора контуров, которые можно наблюдать одновременно. Чтобы внести дополнительную сложность, вы можете смешать функции генерации координат. Например, можно вычислить координату s, используя режим `GL_OBJECT_LINEAR`, а координату t — с помощью `GL_EYE_LINEAR`.

«Сферические» карты

Задача наложения на рисуемый объект его окружения в таком изменении цветов поверхности объекта, как если бы он отражал свое ближайшее окружение. Другими словами, если вы смотрите на безупречно отполированный, абсолютно отражающий серебряный объект в комнате, вы видите на поверхности объекта отражение всех стен, пола и других элементов, присутствующих в комнате. (Классический пример наложения окружения — злобный киборг в фильме «Терминатор-2».) Видимые вами отражения зависят от положения ваших глаз и от положения и угла наклона серебряного объекта. Для получения данного эффекта вам следует создать соответствующую карту текстуры, а задачу сгенерировать ее координаты поручить OpenGL.

Наложение окружения — это приближение, основанное на допущении, что элементы окружения расположены далеко от поверхности зеркального объекта — то есть имеет место ситуация маленького объекта в большом помещении. При таком допущении, чтобы вычислить цвет точки поверхности, проводят луч от наблюдателя к поверхности и отражают его в сторону. Направление отражения луча и определяет цвет, которым будет окрашена точка поверхности. Определение цвета для каждого направления на плоской карте текстуры эквивалентно помещению идеально отполированной сферы в центре относительно своего окружения и снятия картинки с нее камерой, расположенной на большом расстоянии и оснащенной оптической системой, фокусное расстояние которой настолько же велико. Переходя от приближения к математике, это фокусное расстояние бесконечно далеко и камера также удалена в бесконечность. Следовательно, «декодирование» цвета затрагивает округлую область, касающуюся верхнего, нижнего, левого и правого краев карты. Значения текстуры снаружи окружности не принципиальны, так как они не участвуют в процессе наложения окружения.

Чтобы получить безупречно корректную карту текстуры, вам нужно «сделать фотографию» бесконечно большой серебряной сферы в ее «интерьере» с помощью бесконечно удаленной камеры с бесконечно большим фокусным расстоянием, а затем эту фотографию «отсканировать». Чтобы приблизиться к этому бесконечно идеальному результату, вы можете сделать снимок сферы очень широкоугольным объективом («рыбьим глазом»). На цветном рисунке 21 показана фотография, полученная таким образом, и результаты ее использования для наложения окружения.

При наличии необходимой текстуры вы можете активировать алгоритм OpenGL для наложения окружения. Этот алгоритм находит точки поверхности сферы с нужной касательной к поверхности и окрашивает эту точку объекта цветом, видимым в соответствующей точке сферы.

Чтобы сгенерировать координаты текстуры автоматически с учетом наложения окружения, используйте этот код в своей программе:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

При использовании константы `GL_SPHERE_MAP` создаются надлежащие для наложения окружения координаты текстуры. Как видно из кода, вам нужно задать ее для обоих направлений — *s* и *t*. При этом от вас не требуется задание каких-либо параметров для функции генерации координат текстур.

В случае функции `GL_SPHERE_MAP` координаты текстуры генерируются согласно последовательности следующих математических операций:

1. **u** — единичный вектор, направленный от начала координат к вершине (в координатах наблюдателя).
2. **n'** — текущий вектор нормали после преобразования в координаты наблюдателя.
3. **r** — вектор отражения, $(r_x\, r_y\, r_z)^T$, который вычисляется по формуле $\mathbf{u} - 2\mathbf{n}'\mathbf{n}'^T\mathbf{u}$.

4. Временное промежуточное значение, m, вычисляется по формуле $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$.

5. Наконец, координаты текстуры s и t рассчитываются как

   $s = r_x/m + 1/2$

   и

   $t = r_y/m + 1/2$.

«Кубические» текстуры

«Кубические» карты текстур (cube map texture) — это специальная техника, использующая набор из шести двухмерных изображений текстур для получения куба с центром в начале координат. Координаты текстуры (s, t, r) для каждого фрагмента обрабатываются как вектор, при этом каждый тексел представляет что-то, что видно на кубе при «взгляде» из начала координат. Кубические карты идеальны для эффектов наложения окружения, отражения и освещения. Кубические карты могут также «обернуть» сферический объект текстурами, распределяя текселы относительно каждой из ее сторон.

Кубические карты подготавливаются с помощью шести вызовов `glTexImage2D()`, где значение аргумента *target* указывает грань куба (+X, –X, +Y, –Y, +Z или –Z). Такое именование означает, каждая кубическая текстура должна иметь те же размеры, что и грани куба, сформированного с одинаковым количеством текселов на каждой грани, как показано в следующем коде, где значение *imageSize* устанавливается в степень 2:

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA,
   imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image1);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGBA,
   imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image4);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGBA,
   imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image2);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGBA,
   imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image5);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGBA,
   imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image3);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGBA,
   imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image6);
```

Изображения «кубических» текстур генерируются помещением (фактическим или имитацией) камеры в начало координат сцены, этой камерой делаются шесть «моментальных снимков» с полем зрения в 90°, с ориентацией вдоль положительных и отрицательных осей. «Моментальные снимки» разбивают полное трехмерное пространство на шесть пирамид с верхушками в начале координат.

К кубическим картам применимы все стандартные возможности текстурирования, такие как границы текстур, множественные отображения, копирование изображений, парциальные изображения и мультитекстурирование. Для них также предусмотрены специальные текстуры-посредники (`GL_PROXY_TEXTURE_CUBE_`

MAP), поскольку кубическим картам требуется в шесть раз больше памяти, чем обыкновенным двухмерным текстурам. Параметры текстур и объекты текстур применяются к кубической карте в целом, а не для каждой отдельной грани куба. Следующий код представляет собой пример установки режима обработки границ и методов фильтрации для аргумента *target*, установленного в значение GL_TEXTURE_CUBE_MAP:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Чтобы узнать, какую текстуру (и текселы) использовать для данного фрагмента, сначала выбираются текущие координаты (s, t, r) одной из шести текстур, отталкиваясь от того, где s, t и r имеют наибольшие абсолютные значения (главные оси), и знака этих значений (ориентация). Из оставшихся двух координат выделяются координаты с наибольшим значением для определения новых координат (s', t'), которые используются для подстановки соответствующего тексела(-ов) выбранной текстуры кубической карты.

Хотя вы можете вычислить и задать координаты текстуры явно, это, как правило, трудоемко и вовсе не нужно. Практически всегда для генерации координат «кубических» текстур используется команда `glTexGen*()`, предполагающая работу в одном из режимов: GL_REFLECTION_MAP или GL_NORMAL_MAP.

Режим отражения GL_REFLECTION_MAP используется для осуществления тех же вычислений (после шага 3 алгоритма вычисления «сферических» координат, описанного в разделе «„Сферические" карты»), что и выполняемые в режиме генерации координат текстур GL_SPHERE_MAP. Здесь вместо определения (r_x r_y r_z) они проводятся для координат (s, t, r). Режим отражения хорошо подходит для наложения окружения и является альтернативой «сферическому» наложению.

Нормальный режим GL_NORMAL_MAP особенно полезен при визуализации сцен с бесконечно удаленными (или локальными) источниками света и рассеянным отражением. Здесь используется модельно-видовая матрица для преобразования нормалей вершин к системе координат наблюдателя. Результирующие значения (n_x n_y n_z) становятся координатами текстуры (s, t, r). В листинге 9.9 генерация координат текстур выполняется для нормального режима, а также включено текстурирование с применением кубических карт.

Листинг 9.9. Генерация координат кубических текстур: cubemap.c

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
/* turn on cube map texturing */
glEnable(GL_TEXTURE_CUBE_MAP);
```

Мультитекстурирование

В процессе стандартного текстурирования к многоугольнику применяется отдельное изображение текстуры. Мультитекстурирование позволяет применить несколько текстур, одна за другой в очереди операций над текстурами, к одному многоугольнику. Эта совокупность так называемых *модулей текстур* (texture units), где каждый модуль участвует в отдельной операции над текстурами и результаты передаются на следующий этап конвейера до тех пор, пока не будут обработаны все определенные модули. На рис. 9.14 показано, как фрагменты подвергаются четырем операциям текстурирования — по одной на каждый из четырех модулей текстуры.

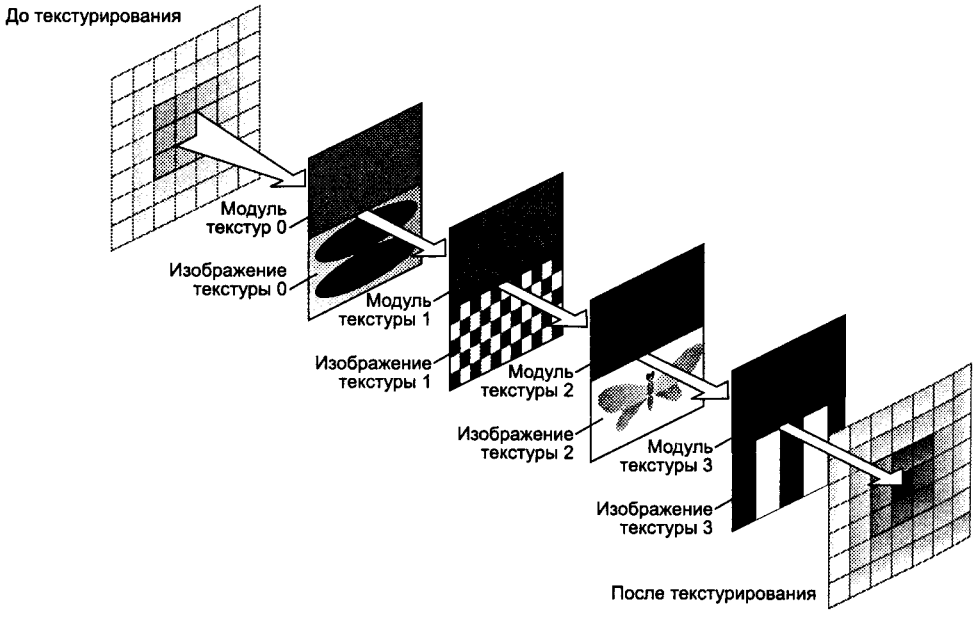


Рис. 9.14. Конвейер мультитекстурирования

Мультитекстурирование относится к поддержанным на расширенном уровне технологиям визуализации, таким как освещение, деколи, композиция и детализация текстур.

Последовательность действий при мультитекстурировании

Чтобы написать код, использующий мультитекстурирование, выполните следующие действия:

ПРИМЕЧАНИЕ

В режиме обратной связи мультитекстурирование не определено вне первого модуля текстуры.

1. Для каждого модуля текстурирования определите состояние, включая изображение текстуры, фильтр, окружение, координаты текстур и матрицу. Используйте команду `glActiveTexture()` для изменения состояния текущего модуля текстуры. Этот вопрос рассматривается в следующем разделе, «Работа с модулями текстур». Вы можете также вызвать команду `glGetIntegerv(GL_MAX_TEXTURE_UNITS,...)`, чтобы увидеть, сколько модулей может быть доступно в вашей реализации. Даже в худшем случае это не меньше двух модулей текстур.
2. Во время описания вершин используйте команду `glMultiTexCoord*()` для задания более чем одной координаты на вершину. Для каждого модуля текстурирования могут быть использованы различные координаты текстур. Каждая координата текстуры будет использована на различных проходах текстурирования. Автоматическая генерация координат текстур и задание координат текстур в массиве вершин — это специальные случаи, которые рассматриваются в разделе «Другие методы задания координат текстур».

Работа с модулями текстур

Мультитекстурирование организуется множеством модулей текстур, которые обрабатываются на дополнительных проходах при работе приложения. Каждый модуль текстуры имеет идентичные возможности, но индивидуальные элементы состояния, к которым относятся:

- изображение текстуры;
- параметры фильтрации;
- среда, в которой выполняется приложение;
- стек матриц текстур;
- автоматическая генерация координат текстур;
- задание массивов вершин (если это нужно).

Каждый модуль текстуры комбинирует предыдущий цвет фрагмента с собственным изображением текстуры, согласно ее состоянию. Результирующий цвет фрагмента передается следующему модулю текстуры, если тот активен.

Каждому модулю текстуры его информация присваивается командой `glActiveTexture()`, выбирающей (устанавливающей) текущий модуль текстуры для его модификации. После этого вызовы `glTexImage*()`, `glTexParameter*()`, `glTexEnv*()`, `glTexGen*()` и `glBindTexture()` воздействуют только на текущий модуль текстуры. Запросы состояния текстур также выполняются через активный модуль текстуры, равно как и запросы текущих координат текстуры и текущих растровых координат текстуры.

void `glActiveTexture`(GLenum *texUnit*);

Выбирает модуль текстуры для его последующей модификации подпрограммами текстурирования. Аргумент *texUnit* — это символическая константа в виде `GL_TEXTUREi`, где i — принадлежит диапазону от 0 до $k - 1$, где k — максимальное количество модулей текстур.

Если используются объекты текстур, вы можете связать текстуру с текущим модулем. Текущий модуль текстуры получит значения состояния текстуры, характеризующие объект текстуры (включая собственно изображение).

Следующий фрагмент кода, листинг 9.10, поделен на две обособленные части. В первой части создаются два обыкновенных объекта текстуры (массивы *texels0* и *texels1* определяют изображения текстур). Во второй части эти два объекта используются для установки двух модулей текстуры.

Листинг 9.10. Инициализация модулей текстуры для мультитекстурирования: multitex.c

```
/* Создаются два обычных объекта текстуры */
GLuint texNames[2];
glGenTextures(2, texNames);
glBindTexture(GL_TEXTURE_2D, texNames[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0, GL_RGBA,
   GL_UNSIGNED_BYTE, texels0);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glBindTexture(GL_TEXTURE_2D, texNames[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 16, 16, 0, GL_RGBA,
   GL_UNSIGNED_BYTE, texels1);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
   GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
   GL_CLAMP_TO_EDGE);

/* Два объекта текстур используются для определения двух модулей текстур
*/
glActiveTexture(GL_TEXTURE0);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texNames[0]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glMatrixMode(GL_TEXTURE);
   glLoadIdentity();
   glTranslatef(0.5f, 0.5f, 0.0f);
   glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
   glTranslatef(-0.5f, -0.5f, 0.0f);
glMatrixMode(GL_MODELVIEW);
glActiveTexture(GL_TEXTURE1);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texNames[1]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

Когда приходит время нарисовать текстурируемый многоугольник, это делается с помощью двух модулей текстур. Для первого модуля применяется изображение *texels0* с ф льтрацией по ближайшим текселам (GL_NEAREST), режимом обработки границ GL_REPEAT, замещением окружения (GL_REPLACE) и с матрицей тексту-

ры, поворачивающей изображение текстуры. После завершения обработки первого модуля только что текстурированный многоугольник передается второму модулю текстуры (GL_TEXTURE1), где обработка выполняется с участием массива *texels1* для изображения текстуры, с линейной фильтрацией, ограничиванием, модуляцией окружения и с единичной (все единицы) матрицей текстур по умолчанию.

ПРИМЕЧАНИЕ

Операции над группой атрибутов текстур (с использованием glPushAttrib(), glPushClientAttrib(), glPopAttrib() или glPopClientAttrib()) сохраняют или восстанавливают состояния для всех модулей текстур (за исключением стека матриц).

Задание вершин и координат текстур для них

При мультитекстурировании одного набора координат текстур на вершину недостаточно — для каждого модуля должен быть свой набор. Вместо использования `glTexCoord*()` вы должны использовать `glMultiTexCoord*()`, которая и задает эти значения.

void `glMultiTexCoord{1234}{sifd}`(GLenum *texUnit*, *TYPE coords*);
void `glMultiTexCoord{1234}{sifd}v`(GLenum *texUnit*, *TYPE *coords*);
Устанавливает данные координат текстур (*s*, *t*, *r*, *q*) из массива *coords* для использования с модулем текстуры *texUnit*. Значения *texUnit* (нумерованные) указываются так же, как и для `glActiveTexture()`.

В листинге 9.11 треугольнику придаются два набора координат текстур, необходимых для мультитекстурирования с помощью двух модулей текстур.

Листинг 9.11. Задание вершин для мультитекстурирования

```
glBegin(GL_TRIANGLES);
glMultiTexCoord2f(GL_TEXTURE0, 0.0, 0.0);
glMultiTexCoord2f(GL_TEXTURE1, 1.0, 0.0);
glVertex2f(0.0, 0.0);
glMultiTexCoord2f(GL_TEXTURE0, 0.5, 1.0);
glMultiTexCoord2f(GL_TEXTURE1, 0.5, 0.0);
glVertex2f(50.0, 100.0);
glMultiTexCoord2f(GL_TEXTURE0, 1.0, 0.0);
glMultiTexCoord2f(GL_TEXTURE1, 1.0, 1.0);
glVertex2f(100.0, 0.0);
glEnd();
```

ПРИМЕЧАНИЕ

Используя команду glTexCoord*(), вы задаете координаты текстуры для первого модуля. Другими словами, использование glTexCoord*() эквивалентно использованию glMultiTexCoord*(GL_TEXTURE0,...).

В редких случаях, когда вы подвергаете мультитекстурированию битовый образ или прямоугольник растрового изображения, вам нужно связать несколько координат текстур с каждой растровой позицией. Следовательно, вы должны вызвать команду `glMultiTexCoord*()` несколько раз, для каждого активного модуля тек-

стуры, для каждого вызова `glRasterPos*()` или `glWindowPos*()`. (Поскольку подразумевается только одна текущая позиция растра для всего битового образа или растрового изображения, то есть только одна соответствующая координата текстуры на модуль, так что с точки зрения эстетики здесь все крайне запущено.)

Другие методы задания координат текстур

Явный вызов `glMultiTexCoord*()` — лишь один из трех способов задания координат текстур при мультитекстурировании. Другие два способа предполагают автоматическую генерацию координат текстур (`glTexGen*()`) или хранение их в массивах вершин (`glTexCoordPointer()`).

Если вы хотите привлечь автоматическую генерацию координат, тогда нужно командой `glActiveTexture()` явно указать, на какой модуль будут воздействовать следующие подпрограммы для автоматической генерации координат:

- `glTexGen*(...);`
- `glEnable(GL_TEXTURE_GEN_*);`
- `glDisable(GL_TEXTURE_GEN_*).`

При желании в ходе мультитекстурирования задавать координаты текстуры в массивах вершин явно укажите командой `glClientActiveTexture()` модуль текстуры, для которого команда `glTexCoordPointer()` будет задавать данные координат текстур.

void `glClientActiveTexture`(GLenum *texUnit*);

Устанавливает текущий модуль текстуры для задания данных координат текстур в массивах вершин. Аргумент *texUnit* — это символическая константа в виде `GL_TEXTURE`*i*, аналогичная подобному аргументу `glActiveTexture()`.

Возвращаясь к одному модулю текстуры

Если вы используете мультитекстурирование и хотите работать с отдельным модулем текстуры, вам нужно отключить текстурирование для всех модулей, за исключением модуля 0, как это показано в листинге 9.12.

Листинг 9.12. Использование при мультитекстурировании лишь одного модуля 0

```
/* отключает текстурирование для всех остальных модулей текстуры */
glActiveTexture (GL_TEXTURE1);
glDisable (GL_TEXTURE_2D);
glActiveTexture (GL_TEXTURE2);
glDisable (GL_TEXTURE_2D);
/* делает модуль 0 текущим */
glActiveTexture (GL_TEXTURE0);
```

Комбинирование функций текстур

OpenGL эволюционирует, и если раньше система фокусировалась на обработке вершин (преобразование, отсечение), то теперь больше внимания уделяется растеризации и операциям над фрагментами. Текстурирование также функционально совершенствуется в направлении обработки фрагментов.

В добавление к многопроходным техникам наложения текстур программистам предоставляются гибкие функции комбинирования текстур с полным контролем над смешиванием фрагментов с текстурами и другими значениями цвета. Функции комбинирования подразумевают получение эффектов высокой реалистичности, таких как «выдавливание» рельефа (bump mapping), более реалистичное зеркальное освещение и обесцвечивание текстур (например, интерполяцией между двумя текстурами). Функции комбинирования берут данные цвета и альфа-канала из трех источников и затем их обрабатывают, результатом чего являются RGBA-значения.

Команда `glTexEnv*()` повсеместно используется для настройки функций комбинирования. В разделе «Функции текстур» вы уже сталкивались с кратким описанием `glTexEnv*()`, и теперь настало время раскрыть подробности.

void `glTexEnv{if}`(GLenum *target*, GLenum *pname*, *TYPE param*);
void `glTexEnv{if}v`(GLenum *target*, GLenum *pname*, *TYPE *param*);
Устанавливает текущую функцию текстурирования. Аргумент *target* выбирается из двух значений: `GL_TEXTURE_FILTER_CONTROL` и `GL_TEXTURE_ENV`.
Если это `GL_TEXTURE_FILTER_CONTROL`, аргумент *pname* должен быть установлен в значение `GL_TEXTURE_LOD_BIAS`, а *param* должен быть отдельным значением с плавающей запятой, используемым для внесения поправки (bias) к параметру уровня детализации MIP-отображения.
Если значением *target* является `GL_TEXTURE_ENV`, допустимые значения для второго и третьего аргументов (*pname* и *param*) см. в табл. 9.8. Когда значение *pname* равно `GL_TEXTURE_ENV_MODE`, аргумент *param* отвечает за то, как значения текстур комбинируются со значениями цвета обрабатываемого фрагмента. Несколько режимов окружения (`GL_BLEND`, `GL_COMBINE`, `GL_COMBINE_RGB` и `GL_COMBINE_ALPHA`) определяют, полезны ли другие режимы окружения.
Если режим окружения текстуры — `GL_BLEND`, используются настройки `GL_TEXTURE_ENV_COLOR`.
Если этот режим — `GL_COMBINE`, также используются параметры `GL_COMBINE_RGB`, `GL_COMBINE_ALPHA`, `GL_RGB_SCALE` или `GL_ALPHA_SCALE`. Для функции `GL_COMBINE_RGB` также могут быть указаны параметры `GL_SOURCE`*i*`_RGB` и `GL_OPERAND`*i*`_RGB` (где *i* — 0, 1 или 2). Аналогично могут указываться параметры `GL_SOURCEi_ALPHA` и `GL_OPERAND`*i*`_ALPHA` для функции `GL_COMBINE_ALPHA`.

Ниже приведен порядок действий по применению функций комбинирования. При мультитекстурировании вы вправе использовать различные функции для каждого модуля текстуры и повторить эти действия для каждого модуля.

- Чтобы использовать любую функцию комбинирования, вы должны вызвать

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
```

- Вы должны указать, как вы хотите комбинировать RGB- или альфа-значения (см. табл. 9.9). Например, в листинге 9.13 для вычитания RGB- и альфа-значений одного источника из другого задействуется текущий модуль текстуры.

Таблица 9.8. Параметры текстуры, если значением аргумента target является GL_TEXTURE_ENV

Аргумент pname команды glTexEnv	Аргумент param команды glTexEnv
GL_TEXTURE_ENV_MODE	GL_DECAL, GL_REPLACE, GL_MODULATE, GL_BLEND, GL_ADD или GL_COMBINE
GL_TEXTURE_ENV_COLOR	Массив из четырех чисел с плавающей запятой: (R, G, B, A)
GL_COMBINE_RGB	GL_REPLACE, GL_MODULATE, GL_ADD, GL_ADD_SIGNED, GL_INTERPOLATE, GL_SUBTRACT, GL_DOT3_RGB или GL_DOT3_RGBA
GL_COMBINE_ALPHA	GL_REPLACE, GL_MODULATE, GL_ADD, GL_ADD_SIGNED, GL_INTERPOLATE или GL_SUBTRACT
GL_SOURCE*i*_RGB или GL_SOURCE*i*_ALPHA (где *i* — 0, 1 или 2)	GL_TEXTURE, GL_TEXTURE*n* (где *n* обозначает *n*-й модуль текстуры и включено мультитекстурирование), GL_CONSTANT, GL_PRIMARY_COLOR или GL_PREVIOUS
GL_OPERAND*i*_RGB (где *i* — 0, 1 или 2)	GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA или GL_ONE_MINUS_SRC_ALPHA
GL_OPERAND*i*_ALPHA (где *i* — 0, 1 или 2)	GL_SRC_ALPHA или L_ONE_MINUS_SRC_ALPHA
GL_RGB_SCALE	Фактор масштабирования цвета в формате с плавающей запятой
GL_ALPHA_SCALE	Фактор масштабирования альфа-компонента в формате с плавающей запятой

Листинг 9.13. Настройки функций комбинирования

```
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_SUBTRACT);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_ALPHA, GL_SUBTRACT);
```

Таблица 9.9. Функции GL_COMBINE_RGB и GL_COMBINE_ALPHA

Аргумент param команды glTexEnv	Функция комбинирования
GL_REPLACE	Arg0
GL_MODULATE (по умолчанию)	Arg0 × Arg1
GL_ADD	Arg0 + Arg1
GL_ADD_SIGNED	Arg0 + Arg1 – 0,5
GL_INTERPOLATE	Arg0 × Arg2 + Arg1 × (1 – Arg2)
GL_SUBTRACT	Arg0 – Arg1
GL_DOT3_RGB GL_DOT3_RGBA	$4 \times ((Arg0_r - 0,5) \times (Arg1_r - 0,5) + (Arg0_g - 0,5) \times (Arg1_g - 0,5) + (Arg0_b - 0,5) \times (Arg1_b - 0,5))$

ПРИМЕЧАНИЕ

Параметры GL_DOT3_RGB и GL_DOT3_RGBA используются только GL_COMBINE_RGB и не используются для GL_COMBINE_ALPHA.

Режимы GL_DOT3_RGB и GL_DOT3_RGBA различаются немногим. При использовании GL_DOT3_RGB одно и то же скалярное произведение присваивается всем трем компонентам (R, G, B). Для GL_DOT3_RGBA результат помещается в четыре компонента (R, G, B, A).

- Задайте источник для *i*-го аргумента функции комбинирования с константой GL_SOURCE*i*_RGB. Количество аргументов (до трех) зависит от типа выбранной функции. Как показано в табл. 9.9, режим GL_SUBTRACT требует двух аргументов, которые могут быть установлены кодом из листинга 9.14.

Листинг 9.14. Настройка источников функции комбинирования

```
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_RGB, GL_PREVIOUS);
```

- Если значение *pname* равно GL_SOURCE*i*_RGB, источник определяется следующими значениями аргумента *param*:
  - GL_TEXTURE — источник для *i*-го аргумента — текстура из текущего модуля текстуры;
  - GL_TEXTURE*n* — текстура, связанная с модулем текстуры *n* (если вы используете этот источник, модуль текстуры *n* должен быть включен и действителен, иначе результат будет неопределен);
  - GL_CONSTANT — постоянный цвет, установленный GL_TEXTURE_ENV_COLOR;
  - GL_PRIMARY_COLOR — входящий фрагмент для модуля текстуры 0, это цвет ранее текстурированного фрагмента;
  - GL_PREVIOUS — входящий фрагмент из предыдущего модуля текстуры (для модуля 0 это то же, что GL_PRIMARY_COLOR).

В предположении, что код в листинге 9.14 устанавливает для модуля текстуры 2 режим GL_SUBTRACT, выходные данные для модуля 1 (GL_PREVIOUS, *Arg1*) вычитаются из данных модуля 2 (GL_TEXTURE, *Arg0*).

- Укажите, какие значения (RGB или альфа-компонента) источника используются и как они используются:
  - GL_OPERAND*i*_RGB сравнивается с соответствующим GL_SOURCE*i*_RGB и определяет значения цвета для текущей функции GL_COMBINE_RGB. Если значение *pname* равно GL_OPERAND*i*_RGB, значение *param* должно быть одним из GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA или GL_ONE_MINUS_SRC_ALPHA;
  - Аналогично, GL_OPERAND*i*_ALPHA сравнивается с соответствующим GL_SOURCE*i*_ALPHA и определяет альфа-значения для текущей функции GL_COMBINE_ALPHA. Здесь аргумент *param* ограничен выбором между GL_SRC_ALPHA и GL_ONE_MINUS_SRC_ALPHA.

Когда для функции GL_COMBINE_RGB используется режим GL_SRC_ALPHA, значения альфа-компонента источника интерпретируются как RGB-значения. В листинге 9.15 заданы три компонента (R, G, B) для *Arg2*: (0.4, 0.4, 0.4).

Листинг 9.15. Использование значений альфа-компонента для операций RGB-комбинирования

```
static GLfloat constColor[4] = {0.1, 0.2, 0.3, 0.4};
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, constColor);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE2_RGB, GL_CONSTANT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND2_RGB, GL_SRC_ALPHA);
```

В листинге 9.15, если имеющийся операнд заменить на GL_SRC_COLOR, RGB-значения будут такими: (0.1, 0.2, 0.3). Для режимов GL_ONE_MINUS* значения образуют дополнения (то есть 1 – цвет или 1 – «альфа»), используемые при вычислениях. Если же этот операнд заменить на GL_ONE_MINUS_SRC_COLOR, RGB-компоненты станут следующими: (0.9; 0.8; 0.7). Для случая GL_ONE_MINUS_SRC_ALPHA результат — (0.6, 0.6, 0.6).

- Факультативно выберите факторы масштабирования значений RGB- или альфа-компонента. По умолчанию это

```
glTexEnvf(GL_TEXTURE_ENV, GL_RGB_SCALE, 1.0);
glTexEnvf(GL_TEXTURE_ENV, GL_ALPHA_SCALE, 1.0);
```

- В последнюю очередь нарисуйте геометрические объекты, тем самым обеспечив вершины, связываемые с координатами текстур.

Интерполирующие функции комбинирования

Интерполирующие функции помогают иллюстрировать комбинирование текстур, поскольку они используют максимальное количество аргументов, несколько источников и режимов операндов. Листинг 9.16 является частью простой программы combiner.c.

Листинг 9.16. Интерполирующие функции комбинирования: combiner.c

```
/* для использования константного цвета текстуры */
static GLfloat constColor[4] = {0.0, 0.0, 0.0, 0.0};

constColor[3] = 0.2;
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, constColor);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_INTERPOLATE);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_RGB, GL_PREVIOUS);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND1_RGB, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE2_RGB, GL_CONSTANT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND2_RGB, GL_SRC_ALPHA);

/* здесь рисуются геометрические объекты */
```

В листинге 9.16 имеется только один активный модуль текстуры. Так как здесь функция комбинирования — GL_INTERPOLATE, есть три аргумента, и они комбинируются между собой по следующей формуле: (Arg0 × Arg2) + (Arg1 × (1 – Arg2)). Вот что они из себя представляют:

- *Arg0*, GL_TEXTURE, изображение текстуры, связанное с текущим прикрепленным объектом текстуры (*texName*[0]);
- *Arg1*, GL_PREVIOUS, результат обработки предыдущего модуля текстуры, но поскольку здесь — это модуль 0, GL_PREVIOUS — это текстурируемый фрагмент;

- *Arg2*, GL_CONSTANT, константный цвет; в нашем случае — (0.0, 0.0, 0.0, 0.2). Результат интерполяции вы получаете как взвешенное смешивание изображения текстуры и фрагмента. Поскольку для режима операнда GL_OPERAND2_RGB указано значение GL_SRC_ALPHA, для взвешивания служит значение альфа-компонента константного цвета *(Arg2)*.

Если вы запустите программу-пример combiner.c, вы увидите 20% текстуры, смешанной с 80% плавно тонированного многоугольника. В программе combiner.c также варьируется значение альфа-компонента для константного цвета, поэтому вы сможете оценить результаты различного взвешивания.

Исследование функции интерполяции показывает, зачем здесь используются несколько значений по умолчанию. Третий аргумент для интерполяции выступает в роли веса для двух других источников. Поскольку интерполяция — всего лишь функция комбинирования трех аргументов, чтобы задать для *Arg2* значение по умолчанию, используется константа GL_CONSTANT. На первый взгляд может показаться необычно, что значение по умолчанию для GL_OPERAND2_RGB — это GL_SRC_ALPHA. Но вес — обычно одинаковый для всех трех компонентов, так что использование одного значения не лишено смысла, и подстановка альфа-значения константы удобна. Значения по умолчанию для некоторых режимов источников и операндов представлены в табл. 9.10.

Таблица 9.10. Значения по умолчанию для некоторых режимов источников и операндов

Аргумент pname команды glTexEnv	Значение инициализации для аргумента param
GL_SOURCE0_RGB	GL_TEXTURE
GL_SOURCE1_RGB	GL_PREVIOUS
GL_SOURCE2_RGB	GL_CONSTANT
GL_OPERAND0_RGB	GL_SRC_COLOR
GL_OPERAND1_RGB	GL_SRC_COLOR
GL_OPERAND2_RGB	GL_SRC_ALPHA

Применение вторичного цвета после текстурирования

Когда текстура применяется к типичному фрагменту, с цветами ее текселов комбинируется только первичный цвет. Первичный цвет может быть результатом вычисления освещения и быть задан командой glColor*().

После текстурирования, но перед обсчетом тумана, иногда к фрагменту также применяется вторичный цвет. Приложения, использующие вторичный цвет, могут выдавать более реалистичные результаты для освещенных и текстурированных объектов.

Вторичный цвет при отключенном освещении

Если не включено ни освещение, ни режим суммирования цвета (командой `glEnable(GL_COLOR_SUM)`), тогда к цвету прошедшего текстурирование объекта добавляется вторичный цвет (устанавливаемый `glSecondaryColor*()`).

void `glSecondaryColor3`{b s i f d ub us ui}(*TYPE r, TYPE g, TYPE b*);
void `glSecondaryColor3`{b s i f d ub us ui}v(const *TYPE *values*);
Устанавливает значения красного, зеленого и синего компонентов для текущего вторичного цвета. Первый суффикс указывает тип данных параметров: байт, короткое целое, целое, вещественное, вещественное двойной точности, беззнаковый байт, беззнаковое короткое целое или беззнаковое целое число. Если присутствует второй суффикс, `v`, тогда аргумент *values* является указателем на массив значений данного типа.

Команды `glSecondaryColor*()` принимают данные тех же типов, что и команды `glColor*()`, и интерпретируют их тем же образом. (См. табл. 4.1.) Вторичные цвета могут также задаваться в массивах вершин.

Вторичный отраженный цвет при отключенном освещении

Операции текстурирования применяются после вычисления освещения, но смешивание зеркальных светлых участков с цветами текстуры обычно уменьшает эффект от освещения. Как говорилось ранее (в разделе «Выбор модели освещения» в главе 5), вы можете вычислить два цвета на вершину: первичный, который представляет все незеркальные составляющие, и вторичный, который есть сумма вкладов всех зеркальных составляющих. Если отраженный цвет отделен от прочих, вторичный (отраженный) цвет добавляется к фрагменту после вычислений текстурирования.

ПРИМЕЧАНИЕ

Если освещение включено, применяется вторичный отраженный свет, несмотря на режим GL_COLOR_SUM, а любой вторичный цвет, установленный командой glSecondaryColor*(), игнорируется.

Стек матриц текстур

Подобно тому, как координаты вашей модели преобразуются по матрице перед построением изображения, координаты текстур умножаются на матрицу 4 × 4 перед любыми наложениями текстур. По умолчанию матрица текстур инициализирована единицами, поэтому координаты текстур, назначенные вами явно или сгенерированные автоматически, остаются неизменными. Изменяя матрицу текстур при перерисовке объекта, вы можете осуществлять «скольжение» текстуры по поверхности, поворачивать текстуру, растягивать и сжимать или комбинировать

все три действия сразу. Фактически, поскольку матрица текстур простая обычная 4 × 4 матрица, можно добиться эффекта, подобного уходу в перспективу.

Матрица текстур, о которой мы говорим, находится на вершине стека, в который можно поместить как минимум две матрицы. Все стандартные команды манипулирования матрицами, такие как `glPushMatrix()`, `glPopMatrix()`, `glMultMatrix()` и `glRotate*()`, в той же мере применимы и к матрицам текстур. Чтобы модифицировать текущую матрицу текстур, вам надо установить режим матриц текстур, `GL_TEXTURE`, как это сделано ниже:

```
glMatrixMode(GL_TEXTURE); /* вход в режим матриц текстур */
glRotated(...);
/*... другие манипуляции с матрицами... */
glMatrixMode(GL_MODELVIEW); /* возврат в модельно-видовой режим */
```

Координата q

Для математиков четвертая координата текстуры, q, подобна w-координате (x, y, z, w) объекта. Когда четыре координаты текстуры (s, t, r, q) умножаются на матрицу текстур, результирующий вектор (s', t', r', q') интерпретируется как однородные координаты текстуры. Другими словами, карта текстуры индексируется значениями s'/q', t'/q' и r'/q'.

Вы можете сделать координату q используемой в случаях, где требуется более одной проекции или преобразования перспективы. Предположим, что вы хотите смоделировать прожектор, который светит не идеально ровно — ярче по центру, например, или световое пятно имеет некруглую форму, потому что линзы изменяют форму светового пучка. Вы можете смоделировать такой свет на плоской поверхности, сделав карту текстуры, соответствующую форме и распределению интенсивности света, и затем спроецировать ее на поверхность уже ближе к использованию преобразований проекции. Проецирование конуса света на поверхности сцены требует преобразований перспективы ($q \neq 1$), так лампы могут светить на поверхности, не перпендикулярные направлению лучей. Другие преобразования перспективы происходят из-за того, что наблюдатель осматривает сцену с различных (но в перспективе) точек. (См. цветной рис. 28 в качестве примера; и работу «Fast Shadows and Lighting Effects Using Texture Mapping» Марка Сегала (Mark Segal), Карла Коробкина (Carl Korobkin), Рольфа ван Виденфельта (Rolf van Widenfelt), Джима Форана (Jim Foran) и Пола Хэберли (Paul Haeberli), труды конференции SIGGRAPH 1992 [*Computer Graphics*, 26:2, July, 1992, с. 249–252] для получения более подробной информации.)

Другой пример — карта текстуры образована на основе фотографии, в которую перспектива уже заложена. Как и для прожектора, здесь конечное изображение зависит от сочетания двух преобразований перспективы.

Пособие по текстурам от Нэйта Робинса

Пособие по текстурам от Нэйта Робинса предоставляет всплывающее меню для просмотра 4 × 4 матриц текстур, внесения в них изменений и для наблюдения за оказанным эффектом.

Глубина текстур

Опробовав технику освещения поверхности (см. главу 5), вы быстро заметите, что источники света OpenGL не приводят к отбрасыванию каких-либо теней. Цвет каждой вершины вычисляется независимо от других объектов сцены. Чтобы получить тени, вам следует выяснить и зафиксировать, какие поверхности (или части поверхности) становятся препятствием на пути прямого распространения света.

Многопроходная техника использует текстуры глубины (depth textures), предоставляющие решение для воспроизведения теней. Если вы временно передвинете точку наблюдения в позицию источника света, вы обнаружите, что все освещенные им элементы сцены не дают теней в перспективу. Текстуры с учетом глубины обеспечивают механизм сохранения значений глубины для всех «неомраченных» фрагментов в *теневой карте* (shadow map). При воспроизведении своей сцены, если вы сравниваете каждый входящий фрагмент с соответствующим значением глубины в карте теней, вы можете выбрать, что рисовать, в зависимости от того, где должно быть затенение. Идея схожа с тестом на глубину, с той разницей, что делается это из точки наблюдения, в которой расположен источник света.

Вот ее концентрированное изложение:

1. Воспроизведите сцену с расположением источника света в точке наблюдения. Неважно, как вообще сцена выглядит; вас должны интересовать только значения глубины. Создайте карту теней, захватывая значения буфера глубины и сохраняя их в карте текстуры («теневой» карте).
2. Сгенерируйте координаты текстуры с координатами (s, t), привязываясь к позициям в пределах карты теней, с третьей координатой текстуры (r), как расстоянием от источника света. Затем нарисуйте сцену еще раз, сравнивая значения r с соответствующими значениями глубины с целью определения, в каких местах фрагменты находятся под светом или в тени.

Следующие разделы содержат более подробное объяснение, а примеры кода иллюстрируют каждый из представленных шагов.

Создание «теневой» карты

Первый шаг — это создание карты текстуры со значениями глубины. Вы создаете ее посредством построения сцены с условием, что точка наблюдения совпадает с позицией источника света. Код из листинга 9.17 вызывает команды `glGetLightfv()` для получения текущей позиции источника света, вычисляет нужный вектор, а затем использует его для видовых преобразований.

Листинг 9.17 начинается с установки размеров окна просмотра, согласованного с картой текстуры. Затем устанавливаются соответствующая проекция и видовые матрицы. Объекты сцены отрисовываются, и результирующее изображение глубины копируется в память текстур для использования ее в качестве карты теней. Наконец, окно просмотра сбрасывается в его оригинальные размеры и позицию.

Отметьте для себя несколько моментов:

- Матрица проекции управляет формой источников света, представляя как бы «абажур». Переменные *lightFovy* и *lightAspect*, передаваемые в `gluPerspective()`, управляют размером «абажура». Маленькое значение *lightFovy* больше подходит для прожектора, а большое значение — для софита, прибора рассеянного света (floodlight).
- Ближняя и дальняя плоскости отсечения для лампы (*lightNearPlane* и *lightFarPlane*) отвечают за точность представления значений глубины. Попробуйте найти «золотую середину» для них, с тем, чтобы точность была максимальной.
- После того как значения глубины попали в буфер глубины, вам нужно захватить их оттуда и поместить в карту текстуры в формате `GL_DEPTH_COMPONENT`. В листинге 9.17 для получения текстуры из содержимого буфера глубины используется команда `glCopyTexImage2D()`. Как и для любой текстуры, обеспечьте равенство размеров изображения текстуры степени двойки.

Листинг 9.17. Воспроизведение сцены с источником света в точке наблюдения: shadowmap.c

```
GLint viewport[4];
GLfloat lightPos[4];

glGetLightfv(GL_LIGHT0, GL_POSITION, lightPos);
glGetIntegerv(GL_VIEWPORT, viewport);

glViewport(0, 0, SHADOW_MAP_WIDTH, SHADOW_MAP_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPerspective(lightFovy, lightAspect, lightNearPlane, lightFarPlane);
glMatrixMode(GL_MODELVIEW);

glPushMatrix();
glLoadIdentity();
gluLookAt(lightPos[0], lightPos[1], lightPos[2],
  lookat[0], lookat[1], lookat[2], up[0], up[1], up[2]);
drawObjects();
glPopMatrix();

glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);

glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 0, 0,
  SHADOW_MAP_WIDTH, SHADOW_MAP_HEIGHT, 0);

glViewport(viewport[0], viewport[1], viewport[2], viewport[3]);
```

Генерация координат текстур и их отображение

Теперь используйте команду `glTexGen*()` для автоматической генерации координат текстур, чтобы вычислить расстояние от позиции источника света в координатах наблюдателя. Значение координаты *r* должно соответствовать дистанции между источником света и примитивом. Вы можете сделать это, используя те же проекцию и видовые преобразования, которые применялись при создании карты тени. В листинге 9.18 для всех матричных вычислений используется стек матриц `GL_MODELVIEW`.

Заметьте, что генерируемые координаты текстуры (*s*, *t*, *r*, *q*) и значения глубины в теневой карте масштабируются не совсем одинаково. Координаты текстуры генерируются в системе наблюдателя, поэтому они попадают в диапазон [−1, 1]. Значения глубины в текселах находятся в пределах [0, 1]. Отсюда, координаты текстур при начальных параллельном переносе и масштабировании карты будут в том же диапазоне значений, что и для теневой карты.

Листинг 9.18. Вычисление координат текстур: shadowmap.c (продолжение)

```
GLfloat tmpMatrix[16];

glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
glTranslatef(0.5, 0.5, 0.0);
glScalef(0.5, 0.5, 1.0);
gluPerspective(lightFovy, lightAspect, lightNearPlane, lightFarPlane);
gluLookAt(lightPos[0], lightPos[1], lightPos[2],
   lookat[0], lookat[1], lookat[2], up[0], up[1], up[2]);
glGetFloatv(GL_MODELVIEW_MATRIX, tmpMatrix);
glPopMatrix();

transposeMatrix(tmpMatrix);

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);

glTexGenfv(GL_S, GL_OBJECT_PLANE, &tmpMatrix[0]);
glTexGenfv(GL_T, GL_OBJECT_PLANE, &tmpMatrix[4]);
glTexGenfv(GL_R, GL_OBJECT_PLANE, &tmpMatrix[8]);
glTexGenfv(GL_Q, GL_OBJECT_PLANE, &tmpMatrix[12]);

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_GEN_Q);
```

В листинге 9.19, перед воспроизведением сцены во второй и последний раз, мы посредством константы режима сравнения текстур `GL_COMPARE_R_TO_TEXTURE` предписываем OpenGL сравнивать *r*-координату фрагмента со значением текселя.

Если расстояние *r* меньше значения тексела или равно ему (функция сравнения GL_LEQUAL), значит, между этим фрагментом и источником света нет никакой преграды, и фрагмент обрабатывается с соответствующим значением яркости. При неуспехе сравнения считаем, что между фрагментом и источником света находится другой примитив, поэтому этот фрагмент затеняется — его яркость равна нулю.

Листинг 9.19. Визуализация сцены со сравнением координаты r: shadowmap.c

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE, GL_LUMINANCE);
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_R_TO_
TEXTURE);
glEnable(GL_TEXTURE_2D);
```

Эта техника может внести некоторые неумышленные визуальные дефекты:

- Самозатенение, когда объект некорректно попадает в тень самого себя, это наиболее общая проблема.
- Ступенчатость в спроецированной текстуре, особенно в областях, более всего удаленных от источника света. Чтобы избавиться от дефекта, увеличьте разрешение карты теней насколько можно.
- Если при текстурировании используется режим GL_MODULATE, могут получиться слишком контрастные (sharp) переходы между затененной и незатененной областями.

К сожалению, нет строгих правил для преодоления этих проблем. При некотором экспериментировании все же можно получить более приятно выглядящее изображение.

10 Буфер кадров

После прочтения этой главы вы узнаете:

- Какие буферы входят в буфер экрана и как они используются
- Как очищать выбранные буферы и производить в них запись
- Как контролировать параметры применяемых к изображениям тестов на отсечение, трафарета, глубины и альфа-теста
- Как выполнять размытие и логические операции
- Как использовать буфер накопления, в частности, для сглаживания сцены

Конечной целью практически любой графической программы является вывод изображения на экран. Экран представляет собой прямоугольный массив пикселов, каждый из которых отображает очень маленький окрашенный квадрат, соответствующий точке изображения. После этапа растеризации (включающего в себя текстурирование и наложение тумана) данные еще не являются пикселами, пока они представляют собой фрагменты. Каждый фрагмент описывается координатной информацией, а также значениями цвета и глубины. Каждый такой фрагмент подвергается серии тестов и операций, наподобие описанных ранее (см. раздел «Смешивание» главы 6) и других, рассматриваемых в этой главе.

Прошедшие тесты фрагменты готовы к тому, чтобы стать пикселами. Чтобы нарисовать эти пикселы, вы должны знать их цвет, то есть информацию, сохраненную в буфере цвета. Хранилище, в которое единообразно помещаются все пикселы, называется *буфером*. Различные буферы могут содержать различное количество информации о пикселе, но внутри каждого буфера пиксел описывается одной и той же информацией. Буфер, сохраняющий один бит информации о пикселе, называется *битовой плоскостью* (bitplane).

Как показано на рис. 10.1, левый нижний пиксел окна OpenGL, (0, 0), в оконных координатах соответствует левому нижнему углу области размерами 1 × 1, занимаемой пикселом. В общем случае пиксел (x, y) заполняет собой область, ограниченную значением x слева, значением $x + 1$ справа, y — снизу и $y + 1$ — сверху.

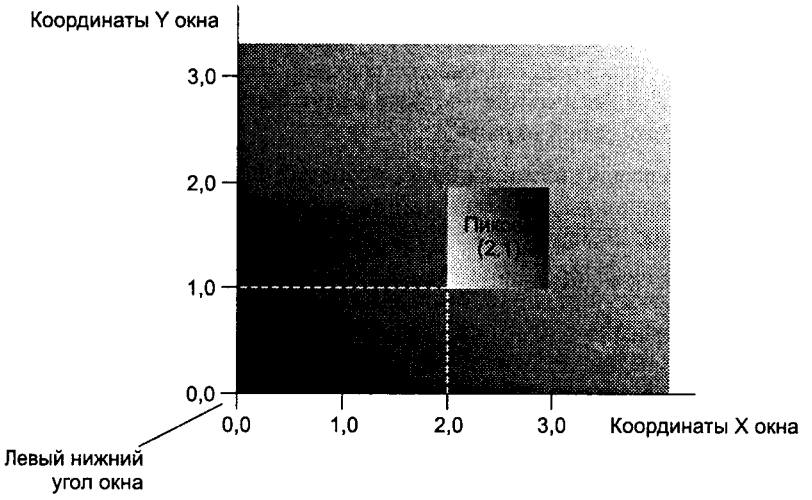


Рис. 10.1. Область, занимаемая пикселом

Давайте более пристально посмотрим на буфер, который хранит цветовую информацию о данных, отображаемых на экране. Положим, что ширина экрана составляет 1280 пикселов, а высота — 1024 пиксела, и что мы имеем дело с 24-разрядным экраном — таким, который может воспроизвести 2^{24} (16 777 216) различных цветов. Поскольку 24 бита это то же, что 3 байта (стандартный байт состоит из 8 бит), буфер цвета должен быть разбит на порции из 3 байтов для каждого из

1 310 720 (произведение 1280 и 1024) пикселов экрана. Число пикселов на физическом экране зависит от возможностей аппаратного обеспечения, но количество информации о пикселе, сохраняемой в буфере цвета, — величина одинаковая во всех случаях.

Буфер цвета — это только один из нескольких буферов, хранящих информацию о пикселах. Вспомните раздел «Удаление невидимых поверхностей» главы 5, из которого вы узнали, что буфер глубины содержит информацию о глубине пикселов. Сам буфер цвета может состоять из нескольких подбуферов. Понятие *буфера кадров* (framebuffer) относится ко всем этим буферам. За исключением буфера цвета, вы не можете видеть эти другие буферы непосредственно; они привлекаются для таких задач, как удаление невидимых поверхностей, сглаживание сцены, отображение по маске, воспроизведение плавного движения, и других операций.

В этой главе описаны все буферы, поддерживаемые реализациями OpenGL, и рассказывается о том, как их использовать. Здесь также говорится о серии тестов, применяемых к пикселам перед их помещением в буфер цвета. Наконец, объясняется назначение буфера накопления — накопление изображений, подлежащих помещению в буфер цвета. Данная глава разбита на следующие основные разделы:

- **Буферы и их применение.** Описывает существующие буферы, их назначение и управление ими — очистку и разрешение записи.
- **Тесты и операции над фрагментами.** Объясняет сущность тестов на отсечение, трафарета, глубины и альфа-теста, выполняемых после расчета положения и цвета пиксела, но до его вывода на экран. Несколько операций — смешивание, размытие и логические — также могут быть выполнены перед тем, как фрагмент обновит собой экран.
- **Буфер накопления.** Описывает более изощренные приемы, включающие в себя сглаживание сцены, размытие при движении и имитацию глубины, подобно фотографиям.

В версии OpenGL 1.4 добавлены операции вывода по шаблону `GL_INCR_WRAP` и `GL_DECR_WRAP`.

Буферы и их применение

Система OpenGL может управлять следующими буферами:

- цветовыми: передним левым, передним правым, задним левым, задним правым и любым количеством вспомогательных буферов;
- буфером глубины;
- буфером трафарета;
- буфером накопления.

Конкретная реализация OpenGL определяет, какие буферы доступны и какое количество битов на пиксел для них принято. Кроме этого поддерживаются состав-

ные, то есть «оконные» типы, подразумевающие работу с несколькими буферами. Таблица 10.1 содержит список параметров функции `glGetIntegerv()`, которая запрашивает систему OpenGL о размере памяти, отводимой пикселу какого-либо объекта визуализации.

ПРИМЕЧАНИЕ

Если вы используете систему X Window, вам гарантируется как минимум один объект визуализации с одним буфером цвета для использования в режиме RGBA со связанными буферами трафарета, глубины и накопления, значения компонентов, цвета которых отличны от нуля. Также, если реализация X Window поддерживает псевдоцвета, вам гарантируется наличие одного объекта визуализации OpenGL с буфером цвета для использования в режиме индексации цвета, с ассоциированными буферами глубины и трафарета. Для запроса об этих объектах существует команда glXGetConfig(); см. приложение В и руководство OpenGL Reference Manual для получения более подробной информации.

Таблица 10.1. Параметры запроса о размере памяти, отводимой под пиксел

Параметр	Пояснение
GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS	Количество битов на R-, G-, B- и A-компоненты цветовых буферов
GL_INDEX_BITS	Количество битов на индекс в буферах цвета
GL_DEPTH_BITS	Количество битов на пиксел в буфере глубины
GL_STENCIL_BITS	Количество битов на пиксел в буфере трафарета
GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, GL_ACCUM_ALPHA_BITS	Количество битов на R-, G-, B- и A-компоненты буфера накопления

Буферы цвета

Буферы цвета это то место, где обычно выполняется рисование. Они содержат RGB- или индексные данные и могут также содержать значения альфа-компонента. В реализациях OpenGL, поддерживающих стереоскопический просмотр, имеются левый и правый буферы цвета для стереоизображений. В противном случае используются только левые буферы. Аналогично, система двойной буферизации подразумевает рабочий и фоновый буферы, а система однократной буферизации работает только с рабочими буферами. Любая реализация OpenGL предоставляет передний левый буфер цвета.

Необязательно, но могут поддерживаться неотображаемые буферы. OpenGL не регламентирует правила их применения, поэтому вы можете использовать их, как пожелаете. Например, задействовать их для хранения многократно используемых изображений. Тогда, чтобы не перерисовывать изображение заново, вы можете просто скопировать существующее из вспомогательного буфера в нужные буферы цвета. (См. описание команды `glCopyPixels()` в разделе «Чтение, запись и копирование пикселов» главы 8.)

В вашем распоряжении параметры `GL_STEREO` и `GL_DOUBLEBUFFER` для команды `glGetBooleanv()`, позволяющие выяснить, поддерживается ли в системе стереовывод (то есть доступен ли правый буфер) или двойная буферизация (задний бу-

фер). Чтобы узнать, какие вообще вспомогательные буферы присутствуют, используйте команду `glGetIntegerv()` с параметром `GL_AUX_BUFFERS`.

Буфер глубины

Буфер глубины хранит значение глубины для каждого пиксела. Как говорилось в разделе «Удаление невидимых поверхностей» в главе 5, под глубиной понимается расстояние до наблюдателя исходя из того, что пикселы с большей глубиной записываются поверх пикселов с меньшими значениями. Это просто полезное соглашение, хотя поведение буфера глубины может быть изменено согласно описанию из раздела «Тест глубины». Буфер глубины иногда также называют *z-буфером* (z — это как бы третья координата (помимо x и y), перпендикулярная экрану).

Буфер трафарета

Одно из применений буфера трафарета это ограничение области рисования какой-то частью экрана, аналогично тому, как шаблон из картона помогает создать резкие изображения при простом распылении красителя. Например, вы можете захотеть воспроизвести картину, видимую через лобовое стекло, имеющее неправильную геометрическую форму. Тогда сохраните форму ветрового стекла в буфере трафарета, а в дальнейшем используйте его для вывода полной сцены. Буфер трафарета предотвратит отображение лишней панорамы, невидимой через переднее стекло. Таким образом, если ваше приложение моделирует автомобиль, достаточно один раз нарисовать приборную доску и другие элементы внутри автомобиля, а при имитации его движения только обновлять внешнюю сцену.

Буфер накопления

Буфер накопления хранит данные цвета RGBA, как и каждый буфер цвета в режиме RGBA. (В режиме индексированного цвета для буфера накопления результаты не определены.) Обычно накапливается серия изображений, затем сливаемая в конечное, составное изображение. Таким образом вы можете выполнять такие операции, как сглаживание, с помощью избыточной выборки (supersampling) и усреднения отсчетов, с целью последующей окраски пикселов в буфере цвета в полученное значение цвета. Рисование в буфер накопления непосредственно не производится; он просто аккумулирует действия, всегда выполняемые прямоугольными областями, поступающими в буфер цвета и исходящими из него.

Очистка буферов

В графических программах очистка экрана (или любого из буферов) — обычно одна из самых «дорогостоящих» операций. Так, на мониторе 1280 × 1024 она требует обработать более миллиона пикселов. В простых графических приложениях операция очистки может отнять больше времени, чем все прочее рисование. Если вам нужно очистить не только буфер цвета, но также и буферы глубины и трафарета, время очистки увеличится еще в три раза.

Чтобы решить эту проблему, некоторое аппаратное обеспечение очищает все буферы вместе. Команды OpenGL для очистки по возможности оптимизированы под такую архитектуру. Вначале вы указываете значения, которые следует записать в каждый буфер для его сброса. Затем дается единственная команда на выполнение очистки, самостоятельно перебирающая буферы из заданного списка. Внутри же все зависит от архитектуры, если она способна на единовременную очистку, таковая и выполняется, в противном случае буферы очищаются последовательно.

Следующие команды устанавливают значения очистки для каждого буфера:

void `glClearColor`(GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);
void `glClearIndex`(GLfloat *index*);
void `glClearDepth`(GLclampd *depth*);
void `glClearStencil`(GLint *s*);
void `glClearAccum`(GLfloat *red*, GLfloat *green*, GLfloat *blue*, GLfloat *alpha*);
Эти команды задают значения очистки для буферов цвета (в режиме RGBA или индексации цвета), глубины, трафарета и накопления соответственно. Типы GLclampf и GLclampd (ограниченные GLfloat и GLdouble) имеют диапазон значений от 0 до 1. Значение по умолчанию для глубины — 1.0; все другие значения по умолчанию равны 0. Однажды установленные значения имеют силу до тех пор, пока они не будут изменены следующим вызовом той же команды.

Непосредственно очистка буфера производится вызовом `glClear()`:

void `glClear`(GLbitfield *mask*);
Очищает указанные буферы. Маска (*mask*) — это поразрядное логическое ИЛИ из констант `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, `GL_STENCIL_BUFFER_BIT` и `GL_ACCUM_BUFFER_BIT`, идентифицирующих собой конкретный тип буфера. Очистка с параметром `GL_COLOR_BUFFER_BIT` выполняется согласно текущему режиму цвета — RGBA или индексному. При этом очищаются все буферы, доступные для записи (см. следующий раздел). Если включены тесты принадлежности (ownership), отсечения и размытия, они применяются к операциям очистки. Также здесь могут быть применены маскирующие операции, такие как `glColorMask()` и `glIndexMask()`. Альфа-тест, тесты трафарета и глубины никак не воздействуют на `glClear()`.

Выбор буферов цвета для записи и чтения

Результаты операций записи (рисования) или чтения могут затрагивать любые буферы цвета: рабочий, фоновый, левый рабочий, левый фоновый, правый рабочий, правый фоновый и любой из вспомогательных буферов. Каждый из этих буферов можно выбрать в качестве целевого как при рисовании, так и при чтении. Для рисования вы можете также установить целевыми более одного буфера одновременно. Для указания буфера записи предназначена команда `glDrawBuffer()`, а для чтения — `glReadBuffer()`. Выбранный буфер используется командами `glReadPixels()`, `glCopyPixels()`, `glCopyTexImage*()` и `glCopyTexSubImage*()`.

В случае двойной буферизации для рисования привлекается только фоновый буфер (а после рисования выполняется перестановка буферов). В иных ситуациях вы можете захотеть обращаться к окну с двойной буферизацией как к окну с однократной буферизацией, одним вызовом `glDrawBuffer()` заполняя оба буфера — рабочий и фоновый.

Команда `glDrawBuffer()` также используется для выбора буферов при воспроизведении стереоизображений (`GL*LEFT` и `GL*RIGHT`) и вывода в дополнительные буферы (`GL_AUX`*i*).

void `glDrawBuffer`(GLenum *mode*);

Указывает буферы цвета для записи или очистки и делает недоступными буферы, активированные предыдущими вызовами `glDrawBuffer()`. Одновременно могут быть активны несколько буферов. Параметр режима (*mode*) принимает одно из следующих значений:

`GL_FRONT`	`GL_FRONT_LEFT`	`GL_AUX`*i*
`GL_BACK`	`GL_FRONT_RIGHT`	`GL_FRONT_AND_BACK`
`GL_LEFT`	`GL_BACK_LEFT`	`GL_NONE`
`GL_RIGHT`	`GL_BACK_RIGHT`	

Параметры `LEFT` и `RIGHT` относятся к обоим стереобуферам — левому и правому; аналогично, параметры `FRONT` и `BACK` воздействуют на два буфера — фоновый и рабочий. Суффикс *i* в `GL_AUX`*i* — это цифра, идентифицирующая дополнительный буфер. По умолчанию режим (*mode*) устанавливается в значение `GL_FRONT` для контекстов с однократной буферизацией и в значение `GL_BACK` для двойной буферизации.

ПРИМЕЧАНИЕ

Вы можете указать для рисования несуществующие буферы, но по крайней мере один из буферов должен существовать. Если ни один из указанных буферов не существует, вы получите сообщение об ошибке.

void `glReadBuffer`(GLenum *mode*);

Выбирает буфер цвета, активируемый как источник для чтения пикселов командами `glReadPixels()`, `glCopyPixels()`, `glCopyTexImage*()`, `glCopyTexSubImage*()` и `glCopyConvolutionFilter*()`, и отключает буферы, включенные предшествующими вызовами `glReadBuffer()`. Значение режима (*mode*) может быть одним из следующих:

`GL_FRONT`	`GL_FRONT_LEFT`	`GL_AUX`*i*
`GL_BACK`	`GL_FRONT_RIGHT`	
`GL_LEFT`	`GL_BACK_LEFT`	
`GL_RIGHT`	`GL_BACK_RIGHT`	

Буферы для `glReadBuffer()` те же, что и для команды `glDrawBuffer()`. По умолчанию аргумент *mode* устанавливается в значение `GL_FRONT` для контекстов с однократной буферизацией и в значение `GL_BACK` для двойной буферизации.

ПРИМЕЧАНИЕ

Буфер для чтения должен существовать, иначе будет сгенерирована ошибка.

Маскирование буферов

Перед записью OpenGL данных в буфер цвета, глубины или трафарета к этим данным применяются операции наложения, или маскирования, как это показано в следующих командах. Для каждой маски и связанных с ней данных выполняется операция поразрядного логического И.

void `glIndexMask`(GLuint *mask*);
void `glColorMask`(GLboolean *red*, GLboolean *green*, GLboolean *blue*, GLboolean *alpha*);
void `glDepthMask`(GLboolean *flag*);
void `glStencilMask`(GLuint *mask*);
Устанавливают маски, используемые для управления записью в определенные буферы. Команда `glIndexMask()` применяется только для режима индексации цвета. Если в маске (*mask*) присутствует 1, в буфер индексов цвета записывается соответствующий бит; для нулевых битов маски запись в буфер не производится. Аналогично, команда `glColorMask()` имеет смысл только для режима RGBA. Значения *red*, *green*, *blue* и *alpha* отвечают за запись соответствующего компонента. (Запись — `GL_TRUE`.)
Если для команды `glDepthMask()` аргумент *flag* установлен в `GL_TRUE`, для записи включается буфер глубины; в противном случае он отключается. Маска для команды `glStencilMask()` используется для данных маски так же, как маска `glIndexMask()`. Значения по умолчанию для любых GLboolean-масок равны `GL_TRUE`, а для GLuint-масок — это единицы.

Маскирование в режиме индексации цвета — возможность самым хитрым образом обработать данные. Например, вы можете использовать каждый бит индекса как отдельный слой и определить взаимодействие между произвольными слоями, соответствующим образом настроив карту цвета. Вы можете создавать наложения и подложки и таким образом получать анимированные картинки. (См. главу 14, где приведены примеры маскирования цвета.) Маскирование в режиме RGBA применяется не так часто, но вы можете использовать его, например, для загрузки отдельных изображений из файлов в красную, зеленую и синюю битовые плоскости.

Мы уже знакомили вас с отключением буфера глубины в разделе «Трехмерное смешивание с использованием буфера глубины» в главе 6. Этот прием полезен и в случае, когда для серии кадров нужен одинаковый фон, а вы хотите добавить что-либо, заслоняющее фоновые участки. Например, предположим, что действие происходит на фоне леса и вам нужно несколько раз показать одни и те же деревья с перемещающимися между ними объектами. После того как деревья нарисованы и их глубины зафиксированы в буфере глубины, изображение с деревьями сохраняется, а новые элементы рисуются в условиях недоступного для записи буфера глубины. Пока новые элементы не перекрывают уже нарисованные новые элементы, картинка корректна. Чтобы нарисовать следующий кадр, восстановите изображение с деревьями и продолжите рисование. Вам не нужно восстанавливать значения из буфера глубины. Этот прием особенно полезен, если задний план достаточно сложен — настолько, что быстрее просто перекопировать изображение из буфера цвета, нежели перевычислять его геометрию.

Маскирование буфера трафарета позволяет использовать многоразрядный буфер трафарета для хранения множества шаблонов (по одному на бит). Эта техника полезна, например, для выполнения так называемого кэпинга (capping) (см. раздел «Тест трафарета» этой главы) или для реализации игры «Жизнь» (см. раздел «„Жизнь“ в буфере трафарета» главы 14).

ПРИМЕЧАНИЕ

Маска, указываемая для glStencilMask(), управляет тем, какие битовые плоскости маски участвуют в записи. Эта маска не связана с маской, задаваемой в качестве третьего аргумента glStencilFunc() и говорящей о том, какие битовые плоскости охватываются буфером трафарета.

Тесты и операции над фрагментами

Когда вы отображаете на экране геометрические фигуры, тексты или изображения, OpenGL выполняет определенные расчеты для поворота, переноса, масштабирования, определения освещения, получения проекции объекта в перспективе, вычисляет, какие пикселы следует воспроизводить в окне, и выясняет, какими цветами должны быть нарисованы эти пикселы. Об управлении этими операциями говорилось во многих из глав, предшествующих данной. Затем OpenGL решает, какие фрагменты следует генерировать и с какими цветами. Часть этапов оставшейся обработки зависит от местонахождения соответствующего фрагменту пиксела в буфере кадров. Например, если он расположен снаружи прямоугольной области или если находится дальше от точки наблюдения, чем пиксел, уже помещенный в буфер кадров, фрагмент не рисуется. На других стадиях цвета фрагментов смешиваются с цветом пиксела, присутствующего в буфере кадров.

Этот раздел описывает оба полных набора тестов: выполняемых до поступления фрагмента в буфер кадров и заключительных действий. Тесты и операции выполняются в следующем порядке (если фрагмент исключается из рассмотрения ранними тестами, к нему не применяются дальнейшие тесты и операции):

1) тест на отсечение;
2) альфа-тест;
3) тест трафарета;
4) тест глубины;
5) смешивание;
6) размытие;
7) логические операции.

Все перечисленные тесты и операции подробно рассматриваются в следующих подразделах.

Тест на отсечение

Вы можете определить прямоугольную область окна и ограничить рисование только ее пределами, используя команду `glScissor()`. Если фрагмент лежит внутри прямоугольника, он считается успешно прошедшим тест на отсечение.

void `glScissor`(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);
Устанавливает положение и размеры прямоугольника отсечения (scissor box). Эти параметры определяют левый нижний угол (x, y) и ширину и высоту прямоугольника. Пикселы, лежащие внутри прямоугольника, успешно проходят тест. Отсечение включается и выключается указанием константы `GL_SCISSOR_TEST` в вызовах `glEnable()` и `glDisable()`. По умолчанию размеры прямоугольника приравниваются размерам окна и отсечение выключено.

Тест на отсечение — это просто частный случай теста трафарета, где в качестве области отсечения используется прямоугольная область. Выделение его в особый случай объясняется тем, что на аппаратном уровне прямоугольное отсечение реализуется намного проще. Если же отсечение происходит медленно, значит оно реализовано через вывод по шаблону программно.

В расширенном варианте отсечение выполняется посредством нелинейного проецирования. Сначала окно делится регулярной сеткой на подобласти, указываются область просмотра и параметры отсечения, которые ограничивают единовременное воспроизведение областей. Далее полная сцена проецируется на каждый регион с использованием его собственной матрицы проекции.

Чтобы узнать, активно ли отсечение, и запросить параметры прямоугольника отсечения, используйте константу `GL_SCISSOR_TEST` для `glIsEnabled()` и `GL_SCISSOR_BOX` для `glGetIntegerv()`.

Альфа-тест

В режиме RGBA альфа-тест позволяет вам принять или отклонить фрагмент, основываясь на значении альфа-компонента. Альфа-тест включается и выключается заданием константы `GL_ALPHA_TEST` для вызовов `glEnable()` и `glDisable()`. Чтобы определить, активен ли этот тест, используйте вызов `glIsEnabled()` с параметром `GL_ALPHA_TEST`.

При работе теста сравнивается входящее значение альфа-компонента с его образцовым значением. Успех прохождения фрагмента через тест зависит от результата сравнения. Как эталонная величина, так и функция сравнения устанавливаются командой `glAlphaFunc()`. По умолчанию в качестве образца принимается 0, функция сравнения приравнивается `GL_ALWAYS`, а альфа-тестирование не действует. Чтобы получить информацию о параметрах тестирования, используйте константы `GL_ALPHA_TEST_FUNC` и `GL_ALPHA_TEST_REF` при вызове `glGetIntegerv()`.

void `glAlphaFunc`(GLenum *func*, GLclampf *ref*);
Устанавливает эталонное значение и функцию сравнения для альфа-теста. Эталонное значение (*ref*) ограничивается интервалом от 0 до 1. Допустимые значения аргумента *func* и их толкование представлены в табл. 10.2. Обозначение alpha_f соответствует фрагменту, а обозначение alpha_r — эталонному значению.

Таблица 10.2. Значения параметров glAlphaFunc()

Параметр	Пояснение
GL_NEVER	Никогда не принимать фрагменты
GL_ALWAYS	Всегда принимать фрагменты
GL_LESS	Принять фрагмент, если $alpha_f < alpha_r$
GL_LEQUAL	Принять фрагмент, если $alpha_f \leq alpha_r$
GL_EQUAL	Принять фрагмент, если $alpha_f = alpha_r$
GL_GEQUAL	Принять фрагмент, если $alpha_f \geq alpha_r$
GL_GREATER	Принять фрагмент, если $alpha_f > alpha_r$
GL_NOTEQUAL	Принять фрагмент, если $alpha_f \neq alpha_r$

Одно из применений альфа-теста — реализация прозрачности. Для этого сцена воспроизводится дважды: сначала с фрагментами, у которых значение альфа-компонента равно 1, а затем с фрагментами, где это значение не равно единице. На обоих проходах следует включить буфер глубины, но на втором нужно также выключить запись в буфер глубины.

Другой вариант применения заключается в указании тех мест изображения, которые не следует отображать, с помощью карт текстур. Благодаря этому вы можете видеть через какие-то части изображения. Для тех участков, которые должны быть видимы, установите значения альфа-компонента в шаблоне равным 0.0, и в 1.0 в противном случае задайте эталонное значение равным 0.5 (или любым другим от 0 до 1) и задайте функцию сравнения GL_GREATER. Такая «переводная картинка» (decal) имеет просвечивающие части, и значения глубины не будут в отношении их эффективны. Эта техника, названная нами обшивкой (*billboarding*, буквально — реклама на стендах) описана в разделе «Пример смешивания» главы 6.

Тест трафарета

Тест трафарета имеет место только при наличии буфера трафарета. (Иначе он всегда пропускается.) При выводе по шаблону эталонное значение для пиксела сравнивается со значением, хранимым в буфере трафарета. В зависимости от результатов сравнения, значение в буфере изменяется. Вы вправе выбрать функцию сравнения и образцовое значение, а также модифицировать их посредством команд `glStencilFunc()` и `glStencilOp()`.

void `glStencilFunc`(GLenum *func*, GLint *ref*, GLuint *mask*);

Устанавливает функцию сравнения (*func*), эталонное значение (*ref*) и маску (*mask*) для их использования в тесте трафарета. Эталонное значение сравнивается со значением из буфера маски заданной функцией, но делается это только для тех битов, для которых биты маски выставлены в 1. Функция может быть одной из следующих: GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER или GL_NOTEQUAL. Например, если это GL_LESS, фрагмент считается успешно прошедшим тест, когда значение *ref* меньше соответствующей величины в буфере трафарета. Если буфер трафарета

содержит *s* битовых плоскостей, к младшим *s* битам маски, значениям из буфера трафарета и эталонному значению применяется поразрядное И до того, как будет выполнено сравнение. Все маскируемые значения интерпретируются как неотрицательные числа. Тест трафарета включается и выключается указанием константы `GL_STENCIL_TEST` в вызовах `glEnable()` и `glDisable()`. По умолчанию функция сравнения (*func*) — `GL_ALWAYS`, *ref* — 0, *mask* — единицы, а обработка по шаблону не активна.

void `glStencilOp`(GLenum *fail*, GLenum *zfail*, GLenum *zpass*);

Задает, как изменяются данные в буфере трафарета по результату теста трафарета. В качестве аргументов выступают три функции — *fail*, *zfail* и *zpass*, которые могут быть указаны константами `GL_KEEP`, `GL_ZERO`, `GL_REPLACE`, `GL_INCR`, `GL_INCR_WRAP`, `GL_DECR`, `GL_DECR_WRAP` или `GL_INVERT`. Они соответствуют сохранению текущего значения, замене его нулем, замене на эталонное значение, инкременту с насыщением, инкременту без насыщения, декременту с насыщением и без него и побитовому инвертированию. Результат инкрементирования и декрементирования ограничивается диапазоном от нуля до максимально допустимого беззнакового целого числа ($2^s - 1$ для буфера трафарета с *s* битовыми плоскостями).

Функция *fail* применяется к фрагментам, потерпевшим неудачу в ходе выполнения теста трафарета; для выдержавших испытание фрагментов срабатывает функция *zfail* в случае, если они не прошли тест глубины, и *zpass* — если они преодолели тест глубины или если он не выполнялся. (См. раздел «Тест глубины».) По умолчанию всем трем аргументам присваивается операция `GL_KEEP`.

«С насыщением» означает, что значение маски ограничивается неким экстремальным уровнем. Если вы попытаетесь уменьшать нуль с насыщением, он так и останется нулем. «Без насыщения» следует понимать так, что при выходе за предел допустимого диапазона значения берутся с другого его конца. Так, при попытке уменьшать нуль без насыщения, значение маски станет максимально возможным (!) беззнаковым целым числом.

Запросы значений теста трафарета

Вы можете узнать значения для всех шести связанных с шаблоном параметров, обратившись к функции запроса `glGetIntegerv()` и подставив в нее одно из значений, перечисленных в табл. 10.3. Вы можете также определить, где будет активен тест трафарета, указав константу `GL_STENCIL_TEST` для вызова `glIsEnabled()`.

Таблица 10.3. Значения запросов для теста трафарета

Значение запроса	Пояснение
GL_STENCIL_FUNC	Функция трафарета
GL_STENCIL_REF	Эталонное значение
GL_STENCIL_VALUE_MASK	Маска трафарета
GL_STENCIL_FAIL	Действие при неуспехе теста трафарета
GL_STENCIL_PASS_DEPTH_FAIL	Действие при неуспехе буфера глубины, если тест трафарета пройден
GL_STENCIL_PASS_DEPTH_PASS	Действие при успехе буфера глубины, если тест трафарета пройден

Примеры применения теста трафарета

Вероятно, самое типичное применение теста трафарета — это маскирование области экрана неправильной формы для предотвращения рисования за пределами этой области (как в примере с ветровым стеклом из раздела «Буферы и их применение»). В этом случае маска трафарета заполняется нулями, а требуемая форма инициируется единицами в буфере трафарета. Вы можете определить геометрическую фигуру непосредственно в буфере трафарета, но тот же результат дает и рисование в буфер цвета вместе с выбором подходящего значения для функции *zpass* (такого как `GL_REPLACE`). (Для рисования пикселов напрямую в буфер трафарета используется команда `glDrawPixels()`.) Во время рисования значение также записывается в буфер маски (в нашем случае эталонное значение). Чтобы рисование в буфер трафарета не отражалось на содержимом буфера цвета, сбросьте маску цвета в 0 (или `GL_FALSE`). Вы вправе отключить также запись в буфер глубины.

После определения области трафарета установите эталонное значение равным 1 и задайте функцию сравнения таким образом, чтобы фрагмент считался успешно прошедшим тест, если значение из битовой плоскости трафарета равно эталонному. Во время рисования не модифицируйте содержимое плоскостей трафарета.

Листинг 10.1 демонстрирует такой способ использования теста трафарета. Рисуются два тора с ромбовидным контуром в центре сцены. В пределах этого контура размещается сфера. Здесь отображение в буфер трафарета имеет место только в момент перерисовки окна, поэтому буфер цвета очищается после создания трафарета.

Листинг 10.1. Использование теста трафарета: stencil.c

```
#define YELLOWMAT 1
#define BLUEMAT 2
void init(void)
{
   GLfloat yellow_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
   GLfloat yellow_specular[] = { 1.0, 1.0, 1.0, 1.0 };

   GLfloat blue_diffuse[] = { 0.1, 0.1, 0.7, 1.0 };
   GLfloat blue_specular[] = { 0.1, 1.0, 1.0, 1.0 };

   GLfloat position_one[] = { 1.0, 1.0, 1.0, 0.0 };

   glNewList(YELLOWMAT, GL_COMPILE);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, yellow_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, yellow_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
   glEndList();

   glNewList(BLUEMAT, GL_COMPILE);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, blue_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, blue_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 45.0);
   glEndList();
```

```
   glLightfv(GL_LIGHT0, GL_POSITION, position_one);
   glEnable(GL_LIGHT0);
   glEnable(GL_LIGHTING);
   glEnable(GL_DEPTH_TEST);

   glClearStencil(0x0);
   glEnable(GL_STENCIL_TEST);
}

/* Рисование сферы в ромбовидный контур,
 * в центре окна с двумя торами. */

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/* Там, где единицы, рисуется синяя сфера */
   glStencilFunc(GL_EQUAL, 0x1, 0x1);
   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
   glCallList(BLUEMAT);
   glutSolidSphere(0.5, 15, 15);

/* Если не единицы, рисуются торы */
   glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);
   glPushMatrix();
      glRotatef(45.0, 0.0, 0.0, 1.0);
      glRotate(45.0, 0.0, 1.0, 0.0);
      glCallList(YELLOWMAT);
      glutSolidTorus(0.275, 0.85, 15, 15);
      glPushMatrix();
         glRotatef(90.0, 1.0, 0.0, 0.0);
         glutSolidTorus(0.275, 0.85, 15, 15);
      glPopMatrix();
   glPopMatrix();
}

/* Каждый раз при изменении геометрии окна переопределяется
 * система координат и перерисовывается область трафарета */
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);

/* Создание области трафарета в виде ромба */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      gluOrtho2D(-3.0, 3.0, -3.0*(GLfloat)h/(GLfloat)w,
                 3.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(-3.0*(GLfloat)w/(GLfloat)h,
                 3.0*(GLfloat)w/(GLfloat)h, -3.0, 3.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
```

продолжение ⇩

Листинг 10.1 (*продолжение*)

```
   glClear(GL_STENCIL_BUFFER_BIT);
   glStencilFunc(GL_ALWAYS, 0x1, 0x1);
   glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
   glBegin(GL_QUADS);
      glVertex2f(-1.0, 0.0);
      glVertex2f(0.0, 1.0);
      glVertex2f(1.0, 0.0);
      glVertex2f(0.0, -1.0);
   glEnd();

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 3.0, 7.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.0, -5.0);
}
/* Главный цикл
 * Убедитесь, что биты шаблона запрашиваются */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH | GLUT_STENCIL);
   glutInitWindowSize(400, 400);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Следующие примеры иллюстрируют другие варианты теста трафарета. (См. главу 14 для получения дополнительной информации.)

- **Кэпинг** или **«наложение шапочки» (capping)** — предположим, что вы рисуете замкнутый выпуклый объект (или несколько таких, не содержащих общих областей объектов), собранный из нескольких многоугольников, и есть плоскость отсечения, которая может отрезать от него кусок, а может и не отрезать. Допустим, что, если плоскость пересекает объект, вы хотите покрыть объект монотонно окрашенной поверхностью, как будто это его внутренний цвет. Для этого заполните буфер трафарета нулями и начните рисование с активным шаблоном и функцией сравнения, которая всегда признает фрагменты успешными. Инвертируйте значения битовых плоскостей трафарета каждый раз при признании фрагмента. После того как все объекты будут нарисованы, не требующие покрытия области экрана будут иметь нули в плоскостях трафарета, а остальные станут ненулевыми. Сбросьте функцию трафарета так, чтобы рисование выполнялось только для ненулевых значений маски, и нарисуйте большой многоугольник цвета накладываемой поверхности на полный экран.

- **Перекрытие (overlapping) полупрозрачных многоугольников** — пусть вы имеете полупрозрачную поверхность, составленную из частично перекрывающихся многоугольников. Если вы просто используете альфа-смешивание, части нижележащих объектов, покрытые несколькими прозрачными поверхностями, будут выглядеть некорректно. Используйте плоскости трафарета, чтобы быть уверенными в том, что каждый фрагмент покрыт не более чем одной прозрачной поверхностью. Делайте это, сбрасывая плоскости в нулевые значения и выполняя рисование только для таких нулевых плоскостей, причем инкрементируя значения в плоскостях трафарета во время вывода.
- **Штриховка (stippling)** — для вывода изображения с шаблоном штриховки. (См. раздел «Отображение точек, линий и многоугольников».) Вы можете сделать это, нарисовав шаблон в буфере трафарета, и тогда рисование будет выполнено согласно этому шаблону. Для того чтобы объект был изображен согласно шаблону, нужно, чтобы буфер трафарета с готовым шаблоном не изменялся во время рисования.

Тест глубины

Буфер глубины хранит для каждого пиксела экрана расстояние между точкой наблюдения и соответствующим местом объекта, ассоциированным с этим пикселом. При рисовании, если тесты глубины успешно пройдены, обрабатываемое значение глубины замещает собой уже находящееся там значение.

В основном буфер глубины используется для исключения невидимых поверхностей. Если новый цвет претендует на пиксел, требования этого объекта удовлетворяются лишь тогда, когда он находится ближе своих предшественников к точке наблюдения. В этом случае после воспроизведения всей сцены на ней остаются только те объекты, которые не заслонены другими. Изначально буфер глубины инициируется значением, соответствующим максимальному удалению от наблюдателя, соответственно, глубина любого объекта заведомо меньше этого значения. Чтобы использовать буфер глубины, вам нужно просто разрешить его применение указанием константы `GL_DEPTH_TEST` в вызове `glEnable()` и помнить, что очистку следует делать перед каждым кадром. (См. раздел «Очистка буферов» этой главы). Вы можете также выбрать другую функцию сравнения для теста глубины с помощью команды `glDepthFunc()`.

> void `glDepthFunc`(GLenum *func*);
> Устанавливает функцию сравнения для теста глубины. Значение аргумента выбирается из списка: `GL_LESS`, `GL_LEQUAL`, `GL_EQUAL`, `GL_GEQUAL`, `GL_GREATER` и `GL_NOTEQUAL`. Все поступающие на тест фрагменты проходят его, если их *z*-значение удовлетворяет заданному соотношению со значением, уже помещенным в буфер глубины. По умолчанию используется функция `GL_LESS`, то есть входящий фрагмент признается, когда его *z*-координата меньше ранее занесенной в буфер. В этом случае *z*-значения представляют собой расстояние от объекта до точки наблюдения, и меньшие величины соответствуют ближе расположенным объектам.

Смешивание, размытие и логические операции

Поступивший на обработку фрагмент, успешно преодолевший все тесты, описанные в разделе «Тесты и операции над фрагментами», допускается к объединению с текущим содержимым буфера цвета одним из нескольких способов. Простейший, и он же по умолчанию, — это перезапись имеющихся значений. Альтернативно, если используется режим RGBA и вы хотите сгладить фрагмент или сделать его прозрачным, вы можете усреднить оба значения — новое и находящееся в буфере, то есть выполнить смешивание (blending). Для систем с небольшим количеством доступных цветов можно «размыть» (dither) значения цвета, увеличив число доступных полутонов, жертвуя разрешением. На финальной стадии в вашем распоряжении поразрядные логические операции, позволяющие произвольно комбинировать входящий фрагмент и уже записанные в буфер пикселы.

Смешивание

При смешивании объединяются R-, G-, B- и альфа-компоненты поступивших на обработку фрагментов с ранее обработанными пикселами. Предусмотрены различные операции смешивания, все они зависят от значений альфа-компонентов входящего фрагмента (если он есть) и сохраненных пикселов. (См. раздел «Смешивание» главы 6 с более подробной информацией по теме.)

Размытие

В системах с небольшим количеством битовых плоскостей цвета вы можете принести разрешение в жертву передаче полутонов (dithering, также размытие, клиширование, смешивание цветов) изображения. Размытие напоминает растрирование (halftoning) в полиграфии. Хотя «Нью-Йорк Таймс» печатается только в двух цветах — белом и черном, — фотографии на страницах этого издания представлены градациями серого как результат комбинирования черных и белых точек. При сравнении газетной фотографии (не в оттенках серого) с ее оригиналом (в полутонах) наблюдается потеря пространственного (spatial) разрешения. Аналогично, системы с поддержкой малого числа битовых плоскостей могут воспользоваться красными, зелеными и синими цветами соседних пикселов для имитации более широкого диапазона цветов.

Операция размытия является аппаратно-зависимой; все реализации OpenGL могут ее включить и выключить. Фактически, на иных компьютерах она вообще ни к чему не приводит, поскольку обычно ощущается на машинах с очень точным воспроизведением цвета. Чтобы включить и выключить размытие, задайте параметр `GL_DITHER` для команд `glEnable()` и `glDisable()`. По умолчанию функция размытия не действует.

Размытие применяется как в RGBA-режиме, так и в режиме индексации цвета. Цвета и цветовые индексы чередуются зависимым от аппаратной части способом, путем выбора из двух ближайших. Например, при индексации цвета, если размы-

тие активно и индекс равен 4,4, из всех пикселов 60 % могут быть нарисованы с индексом 4 и 40 % — с индексом 5. (Есть много алгоритмов передачи полутонов, но в любом случае результирующее значение должно зависеть только от поступающих на обработку значений и координат (x и y) фрагмента.) В режиме RGBA размытие производится раздельно для каждого компонента (включая альфа-канал). Чтобы использовать размытие в режиме индексации цвета, вам нужно упорядочить цвета в цветовой карте по линейному закону; во избежание некорректного результата.

Логические операции

Последними над фрагментом выполняются *логические операции*, как то: ИЛИ (OR), исключающее ИЛИ (XOR) или INVERT. Они применимы к значениям поступившего на обработку фрагмента (источнику) и/или к текущим значениям в буфере цвета (приемник). Такие операции особенно полезны на компьютерах, оперирующих блоками битов, где первичной графической операцией является копирование прямоугольника с данными из одного места окна в другое, или из окна в оперативную память, или из памяти в окно. Обычно данные при копировании не записываются непосредственно в память, что позволяет выполнить над ними и уже имеющимися данными произвольные логические операции; при этом в приемник попадает результат операции.

Поскольку этот процесс может быть реализован достаточно «дешево» на аппаратном уровне, во многие машины логические операции заложены в «железо». Так, XOR может быть использована для стирания-восстановления изображения; повторное применение XOR в том же месте приводит к восстановлению изначального экрана. Как другой пример можно привести интерпретацию индексов цвета в качестве битовых шаблонов. Тогда составное изображение можно комбинировать из слоев, используя ограничивающие маски записи применительно к различным битовым плоскостям, и модифицировать данные слоев посредством логических операций.

Включается и выключается режим логических операций заданием константы `GL_INDEX_LOGIC_OP` или `GL_COLOR_LOGIC_OP` при вызове команд `glEnable()` и `glDisable()` для режимов индексации цвета или RGBA соответственно. В вашем распоряжении выбор из 16 логических операций с помощью `glLogicOp()` или просто применение значения по умолчанию — `GL_COPY`. (В целях обратной совместимости с версией 1.0, `glEnable(GL_LOGIC_OP)` также разрешает применение логических операций в режиме индексации цвета.)

void `glLogicOp`(GLenum *opcode*);
Выбирает логическую операцию, для применения ее к входящему фрагменту (источнику) и пикселу, ранее сохраненному в буфере цвета (приемнику). В табл. 10.4 представлены допустимые значения для аргумента кода операции (*opcode*) и соответствующие объяснения (здесь s — источник и d — приемник). Значение по умолчанию — `GL_COPY`.

Таблица 10.4. Логические операции

Параметр	Операция	Параметр	Операция
GL_CLEAR	0	GL_AND	s ∧ d
GL_COPY	s	GL_OR	s ∨ d
GL_NOOP	d	GL_NAND	¬(s ∧ d)
GL_SET	1	GL_NOR	¬(s ∨ d)
GL_COPY_INVERTED	¬s	GL_XOR	s XOR d
GL_INVERT	¬d	GL_EQUIV	¬(s XOR d)
GL_AND_REVERSE	s ∧ ¬d	GL_AND_INVERTED	¬s ∧ d
GL_OR_REVERSE	s ∨ ¬d	GL_OR_INVERTED	¬s ∨ d

Буфер накопления

Буфер накопления может быть использован для таких целей, как сглаживание сцены, размытие при движении, имитация фотографической глубины резкости и вычисление результирующих теней от множества источников света. Есть и другие применения, причем в сочетании с другими буферами. (См. *The Accumulation Buffer: Hardware Support for High-Quality Rendering* Пауля Хэберли (Paul Haeberli) и Курта Акели (Kurt Akeley (SIGGRAPH, 1990, с. 309–318) для получения большей информации по поводу буфера накопления.)

Операции рисования OpenGL не записывают данные напрямую в буфер накопления. Обычно генерируется серия изображений в одном из стандартных буферов цвета, и эти изображения передаются вместе, за один заход, в буфер накопления. Когда накапливать больше нечего, результат копируется назад в буфер цвета для просмотра. С целью уменьшить ошибку округления, буфер накопления может иметь более высокую точность (больше битов на цвет), чем стандартные буферы цвета. Воспроизведение сцены за несколько проходов длиннее одноразового вывода по времени, но дает более качественные результаты. Решение, что предпочесть в конкретной программе — качество или скорость, — во власти разработчика.

Вы можете использовать буфер накопления подобно тому, как фотограф накладывает одну экспозицию на другую, не переводя кадр. При этом движущийся объект получается размытым. Не удивительно, что компьютер способен на большее, чем фотограф со своей камерой. Например, компьютер обладает всеми возможностями контроля над точкой наблюдения, а фотограф не застрахован от случайной встряски камеры. (См. раздел «Очистка буферов» по вопросу об очистке буфера накопления; используйте команду `glAccum()` для управления им.)

void `glAccum`(GLenum *op*, GLfloat *value*);
Управляет буфером накопления. Аргумент *op* предназначен для выбора операции, а *value* — просто используемое во время операции число. Допустимые операции — это `GL_ACCUM`, `GL_LOAD`, `GL_RETURN`, `GL_ADD` и `GL_MULT`:

GL_ACCUM читает каждый пиксел из буфера, выбранного для чтения командой glReadBuffer(), умножает значения R-, G-, B- и альфа-компонентов на *value* и добавляет к результату значения из буфера накопления.

GL_LOAD действует аналогично GL_ACCUM, за исключением того, что значения замещают содержимое буфера накопления, а не добавляются к ним.

GL_RETURN выбирает значения из буфера накопления, умножает их на *value*, и помещает результат в буфере (буферах) цвета, в который разрешена запись.

GL_ADD добавляет *value* к значению каждого пиксела из буфера накопления, а GL_MULT — умножает это значение на *value*; результат заносится в буфер накопления. Для GL_MULT множитель (*value*) ограничен диапазоном [–1,0, 1,0]. Для GL_ADD ограничения нет.

Сглаживание сцены

Выполняя сглаживание сцены, сначала очистите буфер накопления и разрешите применение переднего буфера для чтения и записи. После этого несколько раз (скажем, *n* раз) повторите код, чуть-чуть смещающий (*jittering* — дрожание, перемещение изображения в близко расположенную позицию) изображение и отрисовывающий его, накапливая данные командой

```
glAccum(GL_ACCUM, 1.0/n);
```

и последней вызывая

```
glAccum(GL_RETURN, 1.0);
```

Заметьте, что этот метод несколько ускорится, если на первом проходе использовать параметр GL_LOAD и опускать очистку буфера накопления. (См. табл. 10.5 в конце главы о возможных значениях «дрожания» (jittering)). Здесь изображение рисуется *n* раз до момента его последнего вывода. Если вы хотите избежать показа пользователю промежуточных изображений, рисуйте в невидимый буфер цвета, накапливайте из него и используйте константу GL_RETURN для рисования в отображаемый буфер (или в фоновый буфер для последующего обмена его содержимого с передним).

Другой вариант — «обернуть» это в пользовательский интерфейс, показывающий, насколько улучшается изображение после каждого момента накопления, и позволяющий пользователю прервать процесс при достижении изображением необходимого качества. Для этого в цикле, последовательно рисующем изображение, вызывайте glAccum() с параметром GL_RETURN после каждого акта накопления, задавая числа 16,0/1,0, 16,0/2,0, 16,0/3,0 и т. д. в качестве второго аргумента. Здесь после первого прохода показывается 1/16 конечного изображения, после двух — 2/16 и т. д. После GL_RETURN должен проверить, нет ли у пользователя желания прервать процесс. Этот интерфейс немного медленнее, поскольку результирующее изображение должно быть скопировано в видимый буфер после каждого прохода.

Выбирая *n*, вам следует найти компромисс между скоростью (чем большее количество раз рисуется сцена, тем длиннее становится весь процесс) и качеством (чем большее число раз рисуется сцена, тем более удовлетворительней она вы-

глядит, пока не выработана разрешающая способность буфера). Описанный процесс иллюстрируется также цветными рисунками 22 и 23.

В листинг 10.2 включены две процедуры для «дрожания» и сглаживания сцены, которые могут оказаться вам полезным: `accPerspective()` и `accFrustum()`. Процедура `accPerspective()` используется вместо команды `gluPerspective()`, первые четыре аргумента обеих процедур одинаковые. Чтобы заставить «дрожать» конус угла зрения, укажите x и y (меньше одного пиксела) в качестве значений пятого и шестого аргументов `accPerspective()`. Также присвойте значение 0.0 седьмому и восьмому аргументам `accPerspective()` и ненулевое значение девятому (с целью упреждения деления на нуль в `accPerspective()`). Эти последние три параметра используются для получения эффекта резко очерченного пространства (глубины резкости), описанного ниже в этой главе.

Листинг 10.2. «Дрожание» объектов сцены: accpersp.c

```
#define PI_ 3.14159265358979323846
void accFrustum(GLdouble left, GLdouble right, GLdouble bottom,
   GLdouble top, GLdouble zNear, GLdouble zFar, GLdouble pixdx,
   GLdouble pixdy, GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
   GLdouble xwsize, ywsize;
   GLdouble dx, dy;
   GLint viewport[4];

   glGetIntegerv(GL_VIEWPORT, viewport);
   xwsize = right - left;
   ywsize = top - bottom;
   dx = -(pixdx*xwsize/(GLdouble) viewport[2] + eyedx*zNear/focus);
   dy = -(pixdy*ywsize/(GLdouble) viewport[3] + eyedy*zNear/focus);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glFrustum(left + dx, right + dx, bottom + dy, top + dy, zNear, zFar);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(-eyedx, -eyedy, 0.0);
}

void accPerspective(GLdouble fovy, GLdouble aspect,
   GLdouble zNear, GLdouble zFar, GLdouble pixdx,
   GLdouble pixdy, GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
   GLdouble fov2,left,right,bottom,top;
   fov2 = ((fovy*PI_) / 180.0) / 2.0;

   top = zNear / (fcos(fov2) / fsin(fov2));
   bottom = -top;
   right = top * aspect;
   left = -right;

   accFrustum(left, right, bottom, top, zNear, zFar,
              pixdx, pixdy, eyedx, eyedy, focus);
}
```

В листинге 10.3 обе процедуры листинга 10.2 задействованы для сглаживания сцены.

Листинг 10.3. Сглаживание сцены: accpersp.c

```
void init(void)
{
  GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
  GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_FLAT);

  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClearAccum(0.0, 0.0, 0.0, 0.0);
}
void displayObjects(void)
{
  GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
  GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
  GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
  GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };

  glPushMatrix();
  glTranslatef(0.0, 0.0, -5.0);
  glRotatef(30.0, 1.0, 0.0, 0.0);

  glPushMatrix();
  glTranslatef(-0.80, 0.35, 0.0);
  glRotatef(100.0, 1.0, 0.0, 0.0);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
  glutSolidTorus(0.275, 0.85, 16, 16);
  glPopMatrix();

  glPushMatrix();
  glTranslatef(-0.75, -0.50, 0.0);
  glRotatef(45.0, 0.0, 0.0, 1.0);
  glRotatef(45.0, 1.0, 0.0, 0.0);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
  glutSolidCube(1.5);
  glPopMatrix();

  glPushMatrix();
  glTranslatef(0.75, 0.60, 0.0);
```

продолжение ⇙

Листинг 10.3 (*продолжение*)

```
   glRotatef(30.0, 1.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
   glutSolidSphere(1.0, 16, 16);
   glPopMatrix();

   glPushMatrix();
   glTranslatef(0.70, -0.90, 0.25);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
   glutSolidOctahedron();
   glPopMatrix();

   glPopMatrix();
}

#define ACSIZE 8
void display(void)
{
   GLint viewport[4];
   int jitter;

   glGetIntegerv(GL_VIEWPORT, viewport);

   glClear(GL_ACCUM_BUFFER_BIT);
   for (jitter = 0; jitter < ACSIZE; jitter++) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      accPerspective(50.0, (GLdouble) viewport[2]/(GLdouble) viewport[3],
         1.0, 15.0, j8[jitter].x, j8[jitter].y, 0.0, 0.0, 1.0);
      displayObjects();
      glAccum(GL_ACCUM, 1.0/ACSIZE);
   }
   glAccum(GL_RETURN, 1.0);
   glFlush();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}
/* Главный цикл
 * Убедитесь в наличии запроса буфера накопления. */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_ACCUM | GLUT_DEPTH);
   glutInitWindowSize(250, 250);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Вы не должны использовать перспективную проекцию для сглаживания сцены. Ступенчатость сцены можно устранять с ортогональной проекцией, просто вызывая `glTranslate*()` для получения «дрожания». Имейте в виду, что команда `glTranslate*()` оперирует мировыми (world) координатами, а вас интересуют координаты экранные. Таким образом, вы должны самостоятельно произвести соответствующее преобразование, разделив высоту и ширину в мировых координатах на размеры области просмотра. Затем умножьте значение в мировых координатах на интервал дрожания, чтобы «предсказать» максимальную величину сдвига — меньше 1 пиксела. Листинг 10.4 показывает, как выглядели бы процедуры `display()` и `reshape()`, если бы ширина и высота в мировых координатах были равны 4.5.

Листинг 10.4. «Дрожание» с помощью ортогональной проекции: accanti.c

```
#define ACSIZE 8

void display(void)
{
  GLint viewport[4];
  int jitter;

  glGetIntegerv(GL_VIEWPORT, viewport);
  glClear(GL_ACCUM_BUFFER_BIT);
  for (jitter = 0; jitter < ACSIZE; jitter++) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
/* Заметьте, что 4,5 — это расстояние в мировых координатах
 * между левой и правой сторонами и между верхней и нижней.
 * Эта формула преобразует движение пиксела от дробного значения
 * к мировым координатам. */
    glTranslatef(j8[jitter].x*4.5/viewport[2],
      j8[jitter].y*4.5/viewport[3], 0.0);
    displayObjects();
    glPopMatrix();
    glAccum(GL_ACCUM, 1.0/ACSIZE);
  }
  glAccum(GL_RETURN, 1.0);
  glFlush();
}

void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
    glOrtho(-2.25, 2.25, -2.25*h/w, 2.25*h/w, -10.0, 10.0);
  else
    glOrtho(-2.25*w/h, 2.25*w/h, -2.25, 2.25, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
```

Размытие при движении

Схожие методы могут использоваться для имитации размытия при движении, как показано на цветном рисунке 7 и на рис. 10.2. Пусть на вашей сцене присутствуют статические и несколько движущихся объектов, и вы хотите получить изображение с размытыми краями, как бы снятое за некоторый период времени. Настройте буфер накопления так же, как в предыдущем примере, но вместо «дрожания» изображения в пространстве заставьте его флуктуировать во времени. Полная сцена может быть получена последовательным «затуханием» при вызове

```
glAccum(GL_MULT, decayFactor);
```

Здесь сцена рисуется в буфер накопления, а *decayFactor* — число в интервале от 0.0 до 1.0. Небольшие числа «периода полураспада» (*decayFactor*) заставляют объект «двигаться» быстрее. Вы можете передать полную сцену, имитируя «испарение» с предшествующей позиции в буфере накопления, в стандартный буфер цвета с помощью вызова

```
glAccum(GL_RETURN, 1.0);
```

Изображение выглядит приемлемо, даже если его элементы «движутся» с различными скоростями или если некоторые из них ускоряются. Как и раньше, чем больше точек «дрожания» (в этом случае во времени) задано, тем лучше заключительное изображение, по крайней мере, до места, где начинается потеря разрешения по причине ограниченной точности буфера накопления. Вы можете объединить размытие при движении со сглаживанием посредством «дрожания» в обоих составляющих пространственно-временного континуума, но платой за более высокое качество будет увеличение времени воспроизведения сцены.

Рис. 10.2. Размытие объекта при движении

Глубина резкости

На фотографии, полученной с помощью фотоаппарата, в фокус попадают только элементы, принадлежащие плоскости, удаленной на определенное расстояние от пленки. С удалением элемента от этой плоскости его четкость снижается. *Глубиной резкости* (depth of field) в терминах фотографии называется область рядом с фокальной плоскостью, где потерей четкости можно пренебречь.

В нормальных условиях все, что рисует OpenGL, всегда находится «в фокусе» (если у вас хороший монитор, вы могли это заметить). Буфер накопления может быть использован для приближения к картинке, которую вы увидели бы, будь

изображение сформировано фотоаппаратом, где элементы тем больше размыты, чем дальше они находятся от фокальной плоскости. Это не точное моделирование съемки фотоаппаратом, а приближение к нему.

Чтобы достигнуть этого результата, повторите многократно рисование сцены, вызывая `glFrustum()` с различными значениями аргументов. Подберите аргументы так, чтобы позиция точки наблюдения слегка отклонялась от ее истинной позиции и чтобы каждый конус угла зрения проецировался в общий прямоугольник, принадлежащий фокальной плоскости, как показано на рис. 10.3. Результаты по всем вариантам воспроизведения сцены нужно усреднить обычным образом через буфер накопления.

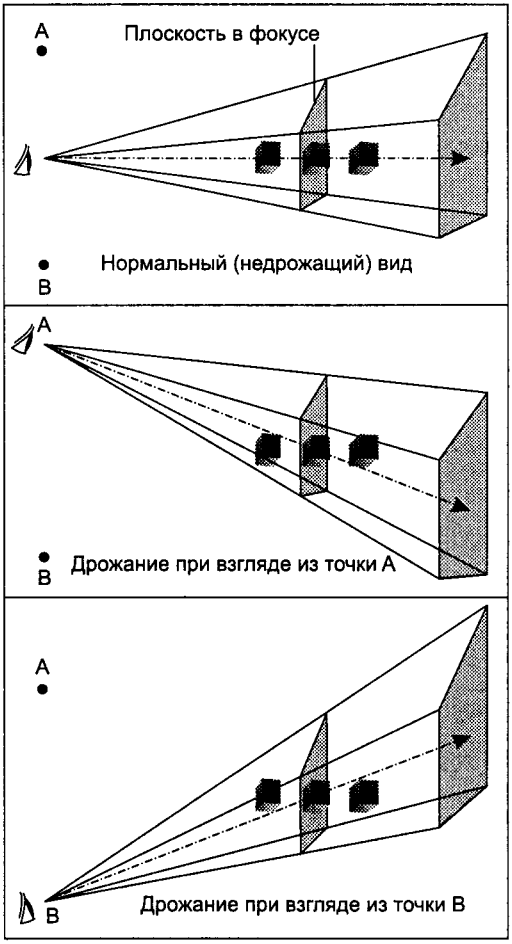


Рис. 10.3. «Дрожание» видимого объема с целью получения эффекта глубины резкости

На цветном рисунке 10 показано изображение пяти чайников, сформированное с имитацией глубины резкости. Чайник золотого цвета (второй слева) находится

в фокусе, а другие теряют четкость при удалении от фокальной плоскости (золотой чайник). Код, посредством которого было получено изображение, приведен в листинге 10.5 (считаем, что `accPerspective()` определены `accFrustum()`, как в листинге 10.2). Сцена рисуется восемь раз, каждый раз с небольшим отклонением, вызовом `accPerspective()`. Как вы помните из примера со сглаживанием сцены, пятый и шестой параметры «дрожания» смещают сцену относительно осей координат x и y. Однако, чтобы получить эффект глубины резкости, положение фокальной плоскости z должно быть неизменным. Оно задается девятым (последним) аргументом `accPerspective()`, и в нашем случае равно 5.0. Уровень размытия определяется константой — произведением смещений «дрожания» x и y (седьмой и восьмой параметры `accPerspective()`). Ее значение не выводится теоретически, а подбирается экспериментально на основании значений глубин резкости изображаемого пространства. (Заметьте, что в листинге 10.5 пятый и шестой аргументы `accPerspective()` равны 0, что означает отсутствие сглаживания сцены.)

Листинг 10.5. Имитация глубины резкости: dof.c

```
void init(void)
{
   GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };

   GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
   GLfloat local_view[] = { 0.0 };

   glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
   glLightfv(GL_LIGHT0, GL_POSITION, position);

   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
   glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

   glFrontFace(GL_CW);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);
   glEnable(GL_DEPTH_TEST);

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClearAccum(0.0, 0.0, 0.0, 0.0);
/* список отображения для чайника */
teapotList = glGenLists(1);
   glNewList(teapotList, GL_COMPILE);
   glutSolidTeapot(0.5);
   glEndList();
}

void renderTeapot(GLfloat x, GLfloat y, GLfloat z,
```

```
  GLfloat ambr, GLfloat ambg, GLfloat ambb,
  GLfloat difr, GLfloat difg, GLfloat difb,
  GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
{
  GLfloat mat[4];

  glPushMatrix();
  glTranslatef(x, y, z);
  mat[0] = ambr; mat[1] = ambg; mat[2] = ambb; mat[3] = 1.0;
  glMaterialfv(GL_FRONT, GL_AMBIENT, mat);
  mat[0] = difr; mat[1] = difg; mat[2] = difb;
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat);
  mat[0] = specr; mat[1] = specg; mat[2] = specb;
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat);
  glMaterialf(GL_FRONT, GL_SHININESS, shine*128.0);
  glCallList(teapotList);
  glPopMatrix();
}

void display(void)
{
  int jitter;
  GLint viewport[4];

  glGetIntegerv(GL_VIEWPORT, viewport);
  glClear(GL_ACCUM_BUFFER_BIT);

  for (jitter = 0; jitter < 8; jitter++) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    accPerspective(45.0, (GLdouble) viewport[2]/(GLdouble) viewport[3],
                   1.0, 15.0, 0.0, 0.0, 0.33*j8[jitter].x,
0.33*j8[jitter].y, 5.0);
/* Рубиновый, золотой, серебряный, изумрудный и цианový чайники */
    renderTeapot(-1.1, -0.5, -4.5, 0.1745, 0.01175,
                 0.01175, 0.61424, 0.04136, 0.04136,
                 0.727811, 0.626959, 0.626959, 0.6);
    renderTeapot(-0.5, -0.5, -5.0, 0.24725, 0.1995,
                 0.0745, 0.75164, 0.60648, 0.22648,
                 0.628281, 0.555802, 0.366065, 0.4);
    renderTeapot(0.2, -0.5, -5.5, 0.19225, 0.19225,
                 0.19225, 0.50754, 0.50754, 0.50754,
                 0.508273, 0.508273, 0.508273, 0.4);
    renderTeapot(1.0, -0.5, -6.0, 0.0215, 0.1745, 0.0215,
                 0.07568, 0.61424, 0.07568, 0.633,
                 0.727811, 0.633, 0.6);
    renderTeapot(1.8, -0.5, -6.5, 0.0, 0.1, 0.06, 0.0,
                 0.50980392, 0.50980392, 0.50196078,
                 0.50196078, 0.50196078,.25);
    glAccum(GL_ACCUM, 0.125);
  }
  glAccum(GL_RETURN, 1.0);
  glFlush();
}
```

продолжение ⇩

Листинг 10.5 (*продолжение*)

```
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}
/* Главный цикл
 * Убедитесь в наличии запроса буфера накопления. */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_ACCUM | GLUT_DEPTH);
   glutInitWindowSize(400, 400);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Плавные тени

Чтобы получить суммарную тень от множества источников цвета, нужно воспроизвести тени от каждого источника по отдельности и свести их вместе. Это вычисление может быть объединено во времени с пространственным «дрожанием» (сглаживанием сцены). (См. раздел «Тени» для получения подробной информации об отрисовке теней.)

«Дрожание»

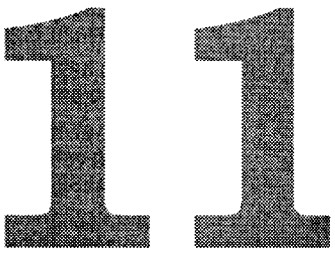
Если вам нужно 9 или 16 экземпляров изображения для устранения его ступенчатости, вы можете подумать, что лучшим выбором точек будет равномерная сетка с шагом в пиксел. Удивительно, что такой подход неверен. В действительности лучшей мыслью будет брать точки, принадлежащие смежным пикселам. В предположении, что вы можете захотеть получить равномерное или нормализованное распределение, со средним значением поближе к центру пикселов, мы подготовили табл. 10.5, в которой представлено несколько наборов подходящих значений для «дрожания» в соответствии с количеством экземпляров изображения. Большинство наборов в этой таблице равномерно распределены в пикселе, и все лежат в пределах пиксела.

Таблица 10.5. Значения для «дрожания»

Количество	Значения
2	{0.25, 0.75}, {0.75, 0.25}
3	{0.5033922635, 0.8317967229}, {0.7806016275, 0.2504380877}, {0.2261828938, 0.4131553612}
4	{0.375, 0.25}, {0.125, 0.75}, {0.875, 0.25}, {0.625, 0.75}

Количество	Значения
5	{0.5, 0.5}, {0.3, 0.1}, {0.7, 0.9}, {0.9, 0.3}, {0.1, 0.7}
6	{0.4646464646, 0.4646464646}, {0.1313131313, 0.7979797979}, {0.5353535353, 0.8686868686}, {0.8686868686, 0.5353535353}, {0.7979797979, 0.1313131313}, {0.2020202020, 0.2020202020}
8	{0.5625, 0.4375}, {0.0625, 0.9375}, {0.3125, 0.6875}, {0.6875, 0.8125}, {0.8125, 0.1875}, {0.9375, 0.5625}, {0.4375, 0.0625}, {0.1875, 0.3125}
9	{0.5, 0.5}, {0.1666666666, 0.9444444444}, {0.5, 0.1666666666}, {0.5, 0.8333333333}, {0.1666666666, 0.2777777777}, {0.8333333333, 0.3888888888}, {0.1666666666, 0.6111111111}, {0.8333333333, 0.7222222222}, {0.8333333333, 0.0555555555}
12	{0.4166666666, 0.625}, {0.9166666666, 0.875}, {0.25, 0.375}, {0.4166666666, 0.125}, {0.75, 0.125}, {0.0833333333, 0.125}, {0.75, 0.625}, {0.25, 0.875}, {0.5833333333, 0.375}, {0.9166666666, 0.375}, {0.0833333333, 0.625}, {0.583333333, 0.875}
16	{0.375, 0.4375}, {0.625, 0.0625}, {0.875, 0.1875}, {0.125, 0.0625}, {0.375, 0.6875}, {0.875, 0.4375}, {0.625, 0.5625}, {0.375, 0.9375}, {0.625, 0.3125}, {0.125, 0.5625}, {0.125, 0.8125}, {0.375, 0.1875}, {0.875, 0.9375}, {0.875, 0.6875}, {0.125, 0.3125}, {0.625, 0.8125}

Мозаичное представление и поверхности второго порядка



Прочитав эту главу, вы научитесь:

- Отображать вогнутые закрашенные многоугольники, предварительно разбитые на выпуклые многоугольники, которые могут быть отображены стандартными средствами OpenGL
- Использовать библиотеку инструментов OpenGL для создания объектов второго порядка — моделировать и отображать поверхности сфер и цилиндров, дисков и неполных дисков

Библиотека OpenGL Library (GL) разработана для низкоуровневых операций, оптимально совместимых с аппаратными видеоускорителями. Библиотека инструментов OpenGL (OpenGL Utility Library, GLU) служит дополнением к библиотеке OpenGL и поддерживает операции более высокого уровня. Некоторые операции GLU рассмотрены в других главах. MIP-преобразование (`gluBuild*DMipmaps()`) и масштабирование изображений (`gluScaleImage()`) обсуждались вместе с другими аспектами наложения текстур в главе 9. Несколько GLU-процедур матричных преобразований (`gluOrtho2D()`, `gluPerspective()`, `gluLookAt()`, `gluProject()`, `gluUnProject()` и `gluUnProject4()`) описаны в главе 3. Об использовании `gluPickMatrix()` говорится в главе 13. Средства GLU NURBS, надстроенные поверх вычислителей OpenGL, представлены в главе 12. Нам осталось затронуть только две темы, связанные с GLU: мозаичное представление многоугольников и квадратичные (второго порядка) поверхности; эти темы обсуждаются в данной главе.

С целью достижения лучшей производительности основные средства OpenGL воспроизводят только выпуклые многоугольники, но GLU содержит подпрограммы для превращения выпуклых многоугольников в вогнутые. В то время как основные средства OpenGL оперируют простыми примитивами, такими как точки, линии и закрашенные многоугольники, GLU способна создавать объекты более высокого уровня, такие как сферические поверхности, цилиндры и конусы.

Эта глава включает в себя следующие основные разделы:

- **Мозаичное представление многоугольников.** Объясняет, как вогнутые многоугольники преобразуются в легко отображаемые выпуклые многоугольники.
- **Поверхности второго порядка: сферы, цилиндры и диски.** Описывает, как генерировать сферы, цилиндры, окружности и дуги, включая такие данные, как нормали к поверхностям и координаты текстур.

Мозаичное представление многоугольников

В разделе «Описание точек, линий и многоугольников» главы 2 уже говорилось, что OpenGL в состоянии непосредственно отображать только простые выпуклые многоугольники. Многоугольник является простым, если его ребра пересекаются только в вершинах, в нем отсутствуют дубликаты вершин и из любой вершины всегда исходят два ребра. Если ваша программа требует отображения вогнутых многоугольников, или содержащих отверстия, или с пересекающимися ребрами, эти многоугольники должны быть до своего отображения разбиты на простые выпуклые многоугольники. Такое представление называется *мозаичным* (tessellation), и библиотека GLU предоставляет соответствующий набор подпрограмм для процесса *тесселяции* — получения мозаичного представления[1]. Эти подпрограммы принимают на вход контуры произвольной формы, которые описывают

[1] Тесселяция (tesselation) — считается, что слово происходит от названия кубиков из смальты — tessera, из которых художники собирают мозаику. Термин еще не совсем прижился в отечественной технической литературе. — *Примеч. перев.*

трудновоспроизводимые многоугольники, и возвращают комбинацию из треугольников, сеток треугольников, «веерров» треугольников и линий.

На рис. 11.1 показаны несколько контуров фигур, требующих разбиения: вогнутый многоугольник, многоугольник с отверстием и самопересекающийся многоугольник.

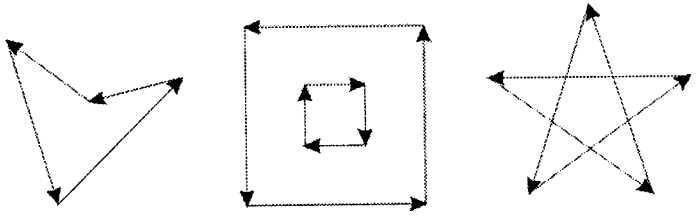


Рис. 11.1. Контуры, требующие мозаичного разбиения

Если вы считаете, что многоугольник нуждается в тесселяции, выполните следующие действия:

1. Создайте новый объект разбиения с помощью команды `gluNewTess()`.
2. Используйте несколько раз `gluTessCallback()`, чтобы зарегистрировать функции обратной связи, вызываемые во время разбиения. Самый сложный случай обратной связи — это когда алгоритм тесселяции обнаруживает пересечение и должен вызвать функцию, зарегистрированную для параметра `GLU_TESS_COMBINE`.
3. Задайте свойства разбиения командой `gluTessProperty()`. Наиболее важное из них — правило «намотки» (winding rule), определяющее области, подлежащие заполнению или остающиеся контурными.
4. Создайте и отобразите мозаику из многоугольников, указав контуры одного или более замкнутых многоугольников. Если данные объекта статичны, инкапсулируйте мозаику в список отображения. (Если вам не требуется повторное вычисление мозаики, списки отображения дадут большую эффективность.)
5. При необходимости распространить разбиение на другие многоугольники вы можете использовать тот же объект мозаичного представления. Если последний вам уже не нужен, удалите его командой `gluDeleteTess()`.

ПРИМЕЧАНИЕ

Представленные рекомендации соответствуют версии GLU 1.2. Если ваша версия более старая, вам следует обратиться к процедурам, описанным в разделе «Сообщения об ошибках GLU». Чтобы узнать версию своей библиотеки GLU, используйте команду gluGetString(GLU_VERSION), которая возвращает требуемое значение в виде строки. А если и gluGetString() нет в вашей GLU, значит — это GLU 1.0.

Создание объекта разбиения

Сложный многоугольник, представленный в виде мозаики, связан с такими данными, как вершины, ребра и функции обратной связи. Чтобы выполнить тесселяцию, ваша программа сначала должна создать объект разбиения с помощью команды `gluNewTess()`.

GLUtesselator* gluNewTess(void);
Создает новый объект разбиения и возвращает указатель на него. При неудаче операции возвращается нулевой указатель.

Один объект разбиения может быть использован повторно. Он необходим только для того, чтобы библиотечные процедуры могли выполнять собственные разбиения, независимые от любых других, которые делает программа. Бывает полезно иметь несколько объектов разбиения, различающихся функциями обратной связи. Но, как правило, типичной программе достаточно выделить единственный объект разбиения и использовать его во всех случаях. Причем нет необходимости освобождать его, поскольку требования объекта разбиения к памяти минимальны. С другой стороны, аккуратность никогда не мешает.

Процедуры обратной связи

После создания объекта разбиения вы должны подготовить несколько процедур обратной связи (callback), вызываемых по ходу тесселяции. Затем с помощью процедур GLU вы описываете контуры одного или более многоугольников. Теперь средство выполнения разбиения будет обращаться к вашим процедурам обратной связи по необходимости.

Те функции, которые опущены, просто не вызываются при тесселяции, и вы не получите информацию, которую они могли бы передать вашей программе. Для указания процедур обратной связи предназначена команда gluTessCallback().

void gluTessCallback(GLUtesselator **tessobj*, GLenum *type*, void (**fn*)());
Ассоциирует функцию обратной связи (*fn*) с объектом разбиения (*tessobj*). Назначение функции определяется аргументом *type*, который может принимать следующие значения: GLU_TESS_BEGIN, GLU_TESS_BEGIN_DATA, GLU_TESS_EDGE_FLAG, GLU_TESS_EDGE_FLAG_DATA, GLU_TESS_VERTEX, GLU_TESS_VERTEX_DATA, GLU_TESS_END, GLU_TESS_END_DATA, GLU_TESS_COMBINE, GLU_TESS_COMBINE_DATA, GLU_TESS_ERROR или GLU_TESS_ERROR_DATA. Двенадцать возможных функций обратной связи имеют следующие прототипы:

GLU_TESS_BEGIN	void begin(GLenum *type*)
GLU_TESS_BEGIN_DATA	void begin(GLenum *type*, void **user_data*)
GLU_TESS_EDGE_FLAG	void edgeFlag(GLboolean *flag*)
GLU_TESS_EDGE_FLAG_DATA	void edgeFlag(GLboolean *flag*, void **user_data*)
GLU_TESS_VERTEX	void vertex(void **vertex_data*)
GLU_TESS_VERTEX_DATA	void vertex(void **vertex_data*, void **user_data*)
GLU_TESS_END	void end(void)
GLU_TESS_END_DATA	void end(void **user_data*)
GLU_TESS_COMBINE	void combine(GLdouble *coords*[3], void **vertex_data*[4], GLfloat *weight*[4], void ***outData*)
GLU_TESS_COMBINE_DATA	void combine(GLdouble *coords*[3], void **vertex_data*[4], GLfloat *weight*[4], void ***outData*, void **user_data*)
GLU_TESS_ERROR	void error(GLenum *errno*)
GLU_TESS_ERROR_DATA	void error(GLenum *errno*, void **user_data*)

Для переназначения процедуры обратной связи просто вызывают команду `gluTessCallback()` с другим аргументом. Чтобы бесследно удалить такую процедуру, в команде `gluTessCallback()` задается нулевой указатель.

Подпрограммы обратной связи вызываются подобно командам OpenGL `glBegin()`, `glEdgeFlag*()`, `glVertex*()` и `glEnd()`. (См. раздел «Маркирование граничных ребер многоугольников» главы 2 для получения большей информации относительно `glEdgeFlag*()`.) Вызовы `GLU_TESS_COMBINE*` используются для создания новой вершины в месте пересечения ребер. К функциям ошибки (`GLU_TESS_ERROR*`) обращение производится, если во время тесселяции что-то оказалось сделано неправильно.

Для каждого объекта разбиения функция для параметра `GLU_TESS_BEGIN` вызывается с одним из четырех возможных значений: `GL_TRIANGLE_FAN`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLES` или `GL_LINE_LOOP`. Во время декомпозиции многоугольников алгоритм разбиения выясняет, какой тип примитива треугольника подходит больше других. (Если включено свойство `GLU_TESS_BOUNDARY_ONLY`, используется `GL_LINE_LOOP`).

Поскольку флаги границы не имеют смысла в «веере» (fan) треугольников или полосе (strip) треугольников, то при задании процедуры, ассоциированной с параметром `GLU_TESS_EDGE_FLAG`, которая включает флаг границ, процедура для `GLU_TESS_BEGIN` вызывается только со значением `GL_TRIANGLES`. Вызов `GLU_TESS_EDGE_FLAG` работает аналогично команде OpenGL `glEdgeFlag*()`.

Между вызовами для `GLU_TESS_BEGIN` и `GLU_TESS_END` производятся обращения к процедурам для параметров `GLU_TESS_EDGE_FLAG` и `GLU_TESS_VERTEX` (обычно с помощью команды `gluTessVertex()`, описанной ниже). Ассоциированные с вершинами и ребрами признаки интерпретируются так же, как если бы вызовы происходили между `glBegin()` и `glEnd()`.

Если что-то пошло неправильно, в процедуру обработки ошибки передается номер ошибки библиотеки GLU. Строка символов, описывающая ошибку, извлекается командой `gluErrorString()`. (См. раздел «Сообщения об ошибках GLU».)

Листинг 11.1 содержит часть программы-примера `tess.c` и демонстрирует создание объекта разбиения и регистрацию нескольких функций обратной связи.

Листинг 11.1. Регистрация функций обратной связи для мозаичного разбиения: tess.c

```
#ifndef CALLBACK
#define CALLBACK
#endif

/* Часть блока инициализации init() */
tobj = gluNewTess();
gluTessCallback(tobj, GLU_TESS_VERTEX, glVertex3dv);
gluTessCallback(tobj, GLU_TESS_BEGIN, beginCallback);
gluTessCallback(tobj, GLU_TESS_END, endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR, errorCallback);

/* Регистрация процедур обратной связи gluTessCallback() */
void CALLBACK beginCallback(GLenum which)
```

```
{
   glBegin(which);
}

void CALLBACK endCallback(void)
{
   glEnd();
}

void CALLBACK errorCallback(GLenum errorCode)
{
   const GLubyte *estring;

   estring = gluErrorString(errorCode);
   fprintf(stderr, "Ошибка разбиения: %s\n", estring);
   exit(0);
}
```

ПРИМЕЧАНИЕ

Приведение типов (type casting) функций обратной связи — дело непростое, особенно если вы хотите получить код, выполняемый и в Microsoft Windows, и в UNIX. Чтобы выполняться под Windows, программы, объявляющие функции обратного вызова, наподобие tess.c, требуют указания обозначения CALLBACK при объявлении функций. Если указать пустое определение CALLBACK (как ниже), это позволит коду одинаково хорошо запускаться как в Microsoft Windows, так и в UNIX:

```
#ifndef CALLBACK
#define CALLBACK
#endif
void CALLBACK callbackFunction(...) {
....
}
```

В листинге 11.1 для параметра обратной связи GLU_TESS_VERTEX просто зарегистрирована команда glVertex3dv(), и передаются только координаты каждой вершины. Если вы хотите задать больше информации для вершин, например цвет, нормаль к поверхности или координаты текстуры, вам требуется более сложная процедура обратной связи. Листинг 11.2 включает в себя начало другого объекта разбиения, полное описание находится в примере tess.c. Здесь зарегистрированная функция vertexCallback() ожидает поступления аргумента, являющегося указателем на шесть значений с плавающей запятой двойной точности: x-, y- и z-координаты и значения красного, зеленого и синего компонент цвета вершины.

Листинг 11.2. Функции обратной связи для вершин и объединения (combine): tess.c

```
/* Другая часть блока инициализации init() */
gluTessCallback(tobj, GLU_TESS_VERTEX, vertexCallback);
gluTessCallback(tobj, GLU_TESS_BEGIN, beginCallback);
gluTessCallback(tobj, GLU_TESS_END, endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR, errorCallback);
gluTessCallback(tobj, GLU_TESS_COMBINE, combineCallback);

/* Регистрация функций обратной связи */
void CALLBACK vertexCallback(GLvoid *vertex)
```

продолжение

Листинг 11.2 (*продолжение*)

```
{
   const GLdouble *pointer;

   pointer = (GLdouble *) vertex;
   glColor3dv(pointer+3);
   glVertex3dv(vertex);
}

void CALLBACK combineCallback(GLdouble coords[3],
   GLdouble *vertex_data[4], GLfloat weight[4], GLdouble **dataOut )
{
   GLdouble *vertex;
   int i;

   vertex = (GLdouble *) malloc(6 * sizeof(GLdouble));
   vertex[0] = coords[0];
   vertex[1] = coords[1];
   vertex[2] = coords[2];
   for (i = 3; i < 6; i++)
      vertex[i] = weight[0] * vertex_data[0][i]
                + weight[1] * vertex_data[1][i]
                + weight[2] * vertex_data[2][i]
                + weight[3] * vertex_data[3][i];
   *dataOut = vertex;
}
```

Листинг 11.2 также демонстрирует использование функции обратной связи GLU_TESS_COMBINE. К этой функции происходит обращение всякий раз, когда алгоритм тесселяции, анализируя входные контуры, обнаруживает пересечение и принимает решение о создании вершины. Она же вызывается, когда тесселятор решает объединить две вершины с практически одинаковыми координатами. В листинге 11.2 новая вершина является линейной комбинацией четырех существующих — *vertex_data*[0..3]. Коэффициенты комбинирования находятся в массиве весов *weight*[0..3]; сумма этих весов равна 1.0. В массив *coords* помещаются координаты новой вершины.

Зарегистрированные процедуры обратной связи должны выделить память для другой вершины, выполнить взвешенную интерполяцию данных на основе данных *vertex_data* и *weight* и вернуть указатель на новую вершину в переменной *dataOut*. Функция `combineCallback()` из листинга 11.2 интерполирует RGB-значения цвета. В начало массива из шести элементов помещаются x-, y- и z-координаты, а взвешенные средние RGB-компонентов заносятся в последние три элемента.

Данные, определяемые пользователем

Зарегистрированы могут быть функции обратной связи шести различных типов. Поскольку для каждой функции имеется две версии, всего получается 12 вызовов. Для первой версии данные определяются пользователем. Эти данные передаются приложением в функцию `gluTessBeginPolygon()` и далее проходят, оставаясь неизменными, каждую процедуру обратной связи `*DATA`. С GLU_TESS_

BEGIN_DATA заданные пользователем данные могут быть применены для отдельных многоугольников. Если указаны обе версии процедуры обратной связи, используется вызов *user_data*, а другой игнорируется. Таким образом, хотя существует 12 вариантов, одновременно активны могут быть не более шести из них.

Так, в листинге 11.2 используется плавное затенение, и процедура `vertexCallback()` задает RGB-цвет для каждой вершины. Если вам нужно реализовать освещение и плавные переходы при выводе теней, для функции обратной связи следует указывать нормаль к поверхности для каждой вершины. Вместе с тем при необходимости в освещении и «плоском» (однотонном) затенении достаточно указать только одну нормаль на многоугольник, но не для всех вершин. В этом случае вам следует выбрать функцию GLU_TESS_BEGIN_DATA и передавать координаты вершины и нормаль к поверхности по указателю *user_data*.

Свойства мозаичного разбиения

Перед тем как выполнить тесселяцию и воспроизведение, вы можете использовать `gluTessProperty()` для установки свойств, имеющих отношение к алгоритму тесселяции. Самое важное и сложное из этих свойств — правило «намотки» (winding rule), определяющее, что следует считать находящимся «внутри» многоугольника, а что — «снаружи».

void `gluTessProperty`(GLUtesselator **tessobj*, GLenum *property*, GLdouble *value*);

Устанавливает значение (*value*) свойства (*property*) для объекта разбиения (*tessobj*). Свойство указывается константами GLU_TESS_BOUNDARY_ONLY, GLU_TESS_TOLERANCE или GLU_TESS_WINDING_RULE.

Если указано свойство GLU_TESS_BOUNDARY_ONLY, его значением может быть GL_TRUE или GL_FALSE. В первом случае многоугольники не разбиваются на многоугольники с заполнением; рисуются лишь контуры, разграничивающие внутреннюю и внешнюю части многоугольника. Значение по умолчанию — GL_FALSE. (См. `gluTessNormal()` для получения более подробной информации.)

Если указано свойство GLU_TESS_TOLERANCE, его значением является расстояние между двумя вершинами, удовлетворяющими требованию близости расположения. Такие вершины могут быть объединены функцией обратной связи GLU_TESS_COMBINE. Чтобы выяснить максимальное расстояние между вершинами, нужно умножить значение допуска на наибольшую разницу координат для поступивших на обработку вершин. Возможность объединения вершин поддержана не во всех реализациях, поэтому значение допуска следует расценивать как рекомендованное. Допуск по умолчанию равен нулю.

Свойство GLU_TESS_WINDING_RULE определяет, какая часть многоугольника является «внутренней», а какая — «наружной» и, соответственно, не подлежащей заполнению. Аргумент *value* здесь может быть одним из следующих значений: GLU_TESS_WINDING_ODD (по умолчанию), GLU_TESS_WINDING_NONZERO, GLU_TESS_WINDING_POSITIVE, GLU_TESS_WINDING_NEGATIVE или GLU_TESS_WINDING_ABS_GEQ_TWO.

Число «витков» и правило «намотки»

Для одиночного контура число «витков» (winding number) точки — это знаковое число оборотов вокруг нее при однократном обходе контура (обходу против часовой стрелки соответствуют положительные значения, по часовой стрелке — отрицательные). В случае нескольких контуров отдельные числа витков складываются. Эта процедура связывает целое знаковое число с каждой точкой плоскости. Заметьте, что число витков одно и то же для всех точек отдельно взятой области.

На рис. 11.2 показаны три набора контуров и их числа витков для точек внутри этих контуров. В левом наборе контуры обходятся против часовой стрелки, соответственно, каждая вложенная область добавляет единицу к числу витков. На среднем рисунке два внутренних контура обходятся по часовой стрелке, и их число витков декрементируется и становится отрицательным.

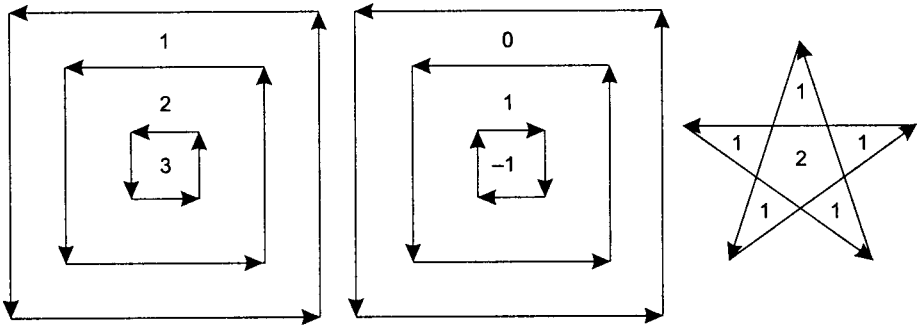


Рис. 11.2. Примеры контуров для демонстрации понятия числа витков

Правило намотки классифицирует область как внутреннюю, если число витков соответствует категории: четное, не нуль, положительное, отрицательное или «абсолютное значение, большее или равное 2». Обычно четные и ненулевые числа свидетельствуют о «внутреннем» характере области. Остальные варианты имеют ограниченное применение для CSG- (constructive solid geometry — конструктивная блочная геометрия) операций над многоугольниками.

Программа-пример `tesswind.c` из состава установочного пакета OpenGL демонстрирует использование правила намотки. На рис. 11.3 отображены четыре набора контуров. Темные участки на рисунке представляют внутренние области. Обратите внимание на то, как влияет на интерпретацию контура направление его обхода.

Правила намотки в конструктивной блочной геометрии

Больше других бывают востребованы параметры `GLU_TESS_WINDING_ODD` и `GLU_TESS_WINDING_NONZERO`. Они находят применение в наиболее типичных случаях затенения.

Правила намотки также применяются в конструктивной блочной геометрии (CSG), где они облегчают поиск объединений, разностей или пересечений (булевы операции) нескольких контуров.

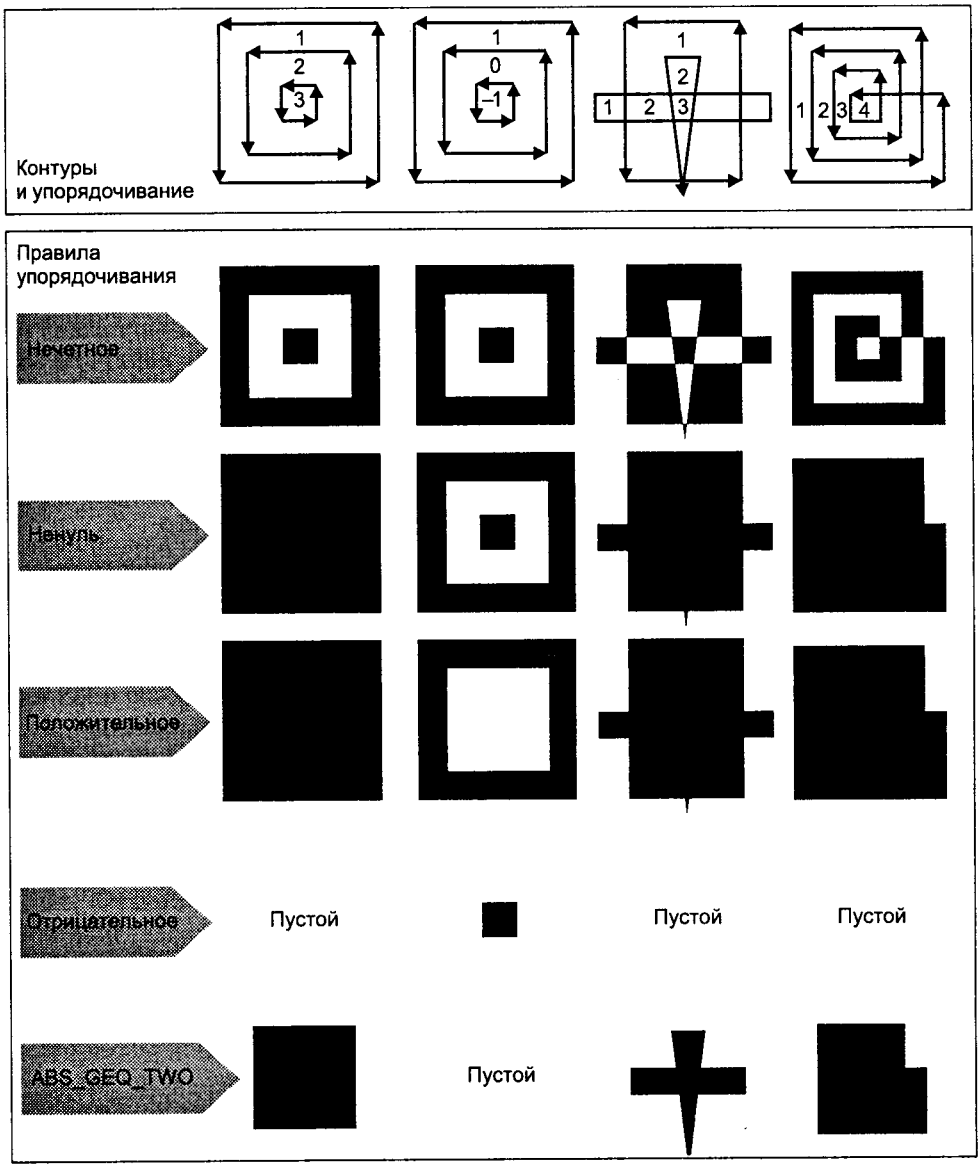


Рис. 11.3. Как правила намотки определяют внутренние области

Для начала сделаем допущение, что для каждого контура число витков равно 0 для любой внутренней области и равно 1 для всех внешних областей. (Контур не должен быть самопересекающимся.) Согласно этой схеме контуры «против часовой стрелки» соответствуют внешним границам многоугольника, а контуры «по часовой стрелке» — дырам. Контуры могут быть вложенными, причем вложенный контур должен быть ориентирован противоположно наружному.

Если оригинальные многоугольники не удовлетворяют этим допущениям, они могут быть предварительно преобразованы к этой форме посредством тесселяции со включенным свойством GLU_TESS_BOUNDARY_ONLY. При этом возвращается список «правильных» контуров. Если вы создадите два объекта разбиения, то с помощью обратной связи можете осуществлять между ними обмен данными.

К многоугольникам представленной формы применимы следующие CSG-операции:

- **Объединение (union).** Для вычисления объединения нескольких контуров нарисуйте все входные контуры как отдельно взятый многоугольник. Число витков в каждой результирующей области — это количество ее оригинальных многоугольников. Объединение может быть получено с помощью правил намотки GLU_TESS_WINDING_NONZERO или GLU_TESS_WINDING_POSITIVE. Заметьте, что в первом случае вы получите тот же результат, как если бы изменили ориентацию всех контуров на обратную.
- **Пересечение (intersection).** Действует при условии одновременной обработки двух контуров. По результатам рисуется один многоугольник. Для извлечения результата используйте GLU_TESS_WINDING_ABS_GEQ_TWO.
- **Разность (difference).** Допустим, что нужно вычислить разницу A diff (B union C union D). Нарисуйте многоугольник, составленный из немодифицированных контуров A, с сопутствующими контурами B, C и D с обратным порядком вершин. Для получения результата используйте правило GLU_TESS_WINDING_POSITIVE. Если B, C и D выводятся из правила GLU_TESS_BOUNDARY_ONLY, другой способ изменения ориентации — функция gluTessNormal() для изменения знака нормали.

Другие функции задания свойств тесселяции

Команда gluGetTessProperty() возвращает текущие значения свойств мозаичного разбиения. В дополнение к ней предусмотрены вспомогательные подпрограммы. Если алгоритм тесселяции применяется для генерации каркасного (wireframe) представления вместо закрашенных многоугольников, gluTessNormal() может быть использована для указания направления «намотки» обрабатываемых многоугольников.

void gluGetTessProperty(GLUtesselator **tessobj*, GLenum *property*, GLdouble **value*);

Возвращает значение (*value*) свойства (*property*) для объекта разбиения (*tessobj*). Значения для аргументов *property* и *value* те же, что и для gluTessProperty().

void gluTessNormal(GLUtesselator **tessobj*, GLdouble *x*, GLdouble *y*, GLdouble *z*);

Определяет вектор нормали для объекта разбиения (*tessobj*), управляющий направлением «намотки» генерируемых многоугольников. Перед разбиением все входные данные проецируются на плоскость, перпендикулярную нормали. Далее все выходные треугольники ориентируются против часовой стрелки относительно этой нормали. Ориентировать их по часовой стрелке можно, если изменить знак нормали. Нормаль по умолчанию — (0, 0, 0).

При наличии представления о расположении и ориентации входных данных используйте команду gluTessNormal(), чтобы увеличить скорость тесселяции. На-

пример, если вы знаете, что все многоугольники принадлежат плоскости *xy*, вам подойдет вызов `gluTessNormal(`*tessobj*, 0, 0, 1).

Повторим, что нормаль по умолчанию — это (0, 0, 0). Эффект от ее применения совсем не очевиден. В таком случае ожидается, что входные данные лежат в плоскости «приблизительно», и плоскость соответствует вершинам невзирая на то, как они соединены в действительности. Знак нормали выбирается так, чтобы сумма знаковых областей всех входных контуров была неотрицательной (где контур «против часовой стрелки» имеет «положительную» область). Заметьте, что, если положение входных данных расходится с допущением о «приблизительности» их к плоскости, проекция, перпендикулярная к вычисленной нормали, может существенно изменить геометрию объектов.

Описание многоугольника

После задания свойств разбиения и регистрации функций обратной связи самое время описать вершины, принадлежащие входным контурам, и произвести собственно процесс тесселяции.

void `gluTessBeginPolygon`(GLUtesselator **tessobj*, void **user_data*);
void `gluTessEndPolygon`(GLUtesselator **tessobj*);
Начинают и заканчивают спецификацию обрабатываемого многоугольника и связывают с ним объект разбиения (*tessobj*). Аргумент *user_data* указывает на определенные пользователем данные, передаваемые всем функциям обратной связи `GLU_TESS_*_DATA`.

Вызовы `gluTessBeginPolygon()` и `gluTessEndPolygon()` обрамляют собой описание одного или нескольких контуров. Когда вызывается `gluTessEndPolygon()`, происходит разбиение, то есть генерация мозаики из треугольников и их отображение. Функции обратной связи и свойства тесселяции устанавливаются командами `gluTessCallback()` и `gluTessProperty()`.

void `gluTessBeginContour`(GLUtesselator **tessobj*);
void `gluTessEndContour`(GLUtesselator **tessobj*);
Начинают и заканчивают спецификацию замкнутых контуров, являющихся частью многоугольника. Описание замкнутого контура включает в себя нуль и более вызовов `gluTessVertex()`, определяющих вершины. Последняя вершина каждого контура автоматически замыкается на первую.

Если говорить практически, для определения контура достаточно трех вершин.

void `gluTessVertex`(GLUtesselator **tessobj*, GLdouble *coords*[3], void **vertex_data*);
Задает вершину текущего контура объекта разбиения. Массив *coords* содержит координаты вершины в трехмерном пространстве, а *vertex_data* — это указатель, посылаемый функциям обратной связи для параметров `GLU_TESS_VERTEX` или `GLU_TESS_VERTEX_DATA`. Обычно массив *vertex_data* содержит координатную и цветовую информацию для вершины и текстур, нормали к поверхности или любую другую, требуемую программе.

В примере `tess.c` та часть, которая вошла в листинг 11.3, определяет два многоугольника. Первый — это прямоугольный контур с треугольным отверстием, а другой — равномерно окрашенная пятиконечная звезда, то есть самопересекающийся контур. С целью увеличения быстродействия оба многоугольника сохранены в списках отображения. Первый многоугольник состоит из двух контуров; внешний обходится против часовой стрелки, а «дыра» — по часовой стрелке. Массив *star* содержит координатные и цветовые данные второго многоугольника. Назначенные для него функции обратной связи `vertexCallback()` при разбиении обращаются к этим данным.

Важно, что каждая вершина расположена в памяти отдельно, поскольку ее данные не копируются `gluTessVertex()`; сохраняется только указатель на *vertex_data*. Если программа станет записывать данные для разных вершин в одно и то же место, результат может оказаться далеким от желаемого.

ПРИМЕЧАНИЕ

Может показаться, что двойное определение координат для gluTessVertex() — сначала в coords и затем в vertex_data — избыточно; однако нужны оба параметра. Дублируются только координаты вершины, а кроме них массив vertex_data содержит другую полезную информацию.

Листинг 11.3. Определение многоугольника: tess.c

```
GLdouble rect[4][3] = {50.0, 50.0, 0.0,
                       200.0, 50.0, 0.0,
                       200.0, 200.0, 0.0,
                       50.0, 200.0, 0.0};
GLdouble tri[3][3] = {75.0, 75.0, 0.0,
                      125.0, 175.0, 0.0,
                       175.0, 75.0, 0.0};
GLdouble star[5][6] = {250.0, 50.0, 0.0, 1.0, 0.0, 1.0,
                       325.0, 200.0, 0.0, 1.0, 1.0, 0.0,
                       400.0, 50.0, 0.0, 0.0, 1.0, 1.0,
                       250.0, 150.0, 0.0, 1.0, 0.0, 0.0,
                       400.0, 150.0, 0.0, 0.0, 1.0, 0.0};

startList = glGenLists(2);
tobj = gluNewTess();
gluTessCallback(tobj, GLU_TESS_VERTEX, glVertex3dv);
gluTessCallback(tobj, GLU_TESS_BEGIN, beginCallback);
gluTessCallback(tobj, GLU_TESS_END, endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR, errorCallback);

glNewList(startList, GL_COMPILE);
glShadeModel(GL_FLAT);
gluTessBeginPolygon(tobj, NULL);
   gluTessBeginContour(tobj);
      gluTessVertex(tobj, rect[0], rect[0]);
      gluTessVertex(tobj, rect[1], rect[1]);
      gluTessVertex(tobj, rect[2], rect[2]);
      gluTessVertex(tobj, rect[3], rect[3]);
   gluTessEndContour(tobj);
   gluTessBeginContour(tobj);
```

```
      gluTessVertex(tobj, tri[0], tri[0]);
      gluTessVertex(tobj, tri[1], tri[1]);
      gluTessVertex(tobj, tri[2], tri[2]);
   gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList();

gluTessCallback(tobj, GLU_TESS_VERTEX, vertexCallback);
gluTessCallback(tobj, GLU_TESS_BEGIN, beginCallback);
gluTessCallback(tobj, GLU_TESS_END, endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR, errorCallback);
gluTessCallback(tobj, GLU_TESS_COMBINE, combineCallback);

glNewList(startList + 1, GL_COMPILE);
glShadeModel(GL_SMOOTH);
gluTessProperty(tobj, GLU_TESS_WINDING_RULE, GLU_TESS_WINDING_POSITIVE);
gluTessBeginPolygon(tobj, NULL);
   gluTessBeginContour(tobj);
      gluTessVertex(tobj, star[0], star[0]);
      gluTessVertex(tobj, star[1], star[1]);
      gluTessVertex(tobj, star[2], star[2]);
      gluTessVertex(tobj, star[3], star[3]);
      gluTessVertex(tobj, star[4], star[4]);
   gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList();
```

Удаление объекта разбиения

Если вы больше не нуждаетесь в объекте разбиения, вы можете удалить его и освободить занимаемую им память посредством команды `gluDeleteTess()`.

void `gluDeleteTess`(GLUtesselator **tessobj*);
Удаляет указанный объект разбиения (*tessobj*) и высвобождает отведенную под него память.

Советы по увеличению производительности разбиения

Чтобы ваше разбиение работало быстро, придерживайтесь следующих правил:

- Кэшируйте вывод алгоритма тесселяции в списке отображения или в пользовательской структуре. Получайте координаты вершин после разбиения, то есть не забывайте пользоваться обратной связью. (См. раздел «Обратная связь» главы 13.)
- Используйте `gluTessNormal()` для задания нормали многоугольнику.
- Используйте один и тот же объект разбиения для разных многоугольников, это быстрее, чем выделение персонального объекта разбиения каждому многоугольнику. (Преимущества последнего проявляются в многозадачной, многопроцессорной среде.)

Сообщения об ошибках GLU

Библиотека GLU предоставляет вызов, возвращающий описательную строку для кода ошибки. Эта подпрограмма не ограничивается тесселяцией, а также действует для ошибок, связанных с NURBS и квадратичными поверхностями, подобно основной библиотеке GL. (См. раздел «Обработка ошибок» главы 14.)

Обратная совместимость

Если вы используете версию 1.0 или 1.1 библиотеки GLU, не удивляйтесь, что тесселятор работает медленно. В этих версиях обрабатываются только простые невыпуклые многоугольники и такие же простые многоугольники с отверстиями. Также в них не принимаются во внимание пересекающиеся контуры (нет обратной связи для параметра `COMBINE`). Тесселятор версии 1.0/1.1 все еще работает в GLU 1.2 и 1.3, но его использование не рекомендуется.

Тесселятор 1.0/1.1 имеет много общего с его развитым потомком. Функции `gluNewTess()` и `gluDeleteTess()` используются и там, и там. Главной процедурой для спецификации вершины осталась `gluTessVertex()`. Механизм обратной связи управляется `gluTessCallback()`, хотя регистрировать можно только пять функций из нынешних двенадцати.

Ниже представлены прототипы вызовов тесселятора версий 1.0/1.1:

void `gluBeginPolygon`(GLUtriangulatorObj **tessobj*);
void `gluNextContour`(GLUtriangulatorObj **tessobj*, GLenum *type*);
void `gluEndPolygon`(GLUtriangulatorObj **tessobj*);

Самый дальний от центра контур специфицируется первым, при этом не обязательно проводить инициализацию `gluNextContour()`. Для многоугольников без отверстий определяется только один контур, а `gluNextContour()`, соответственно, не используется. Если многоугольник состоит из множества контуров (то есть отверстий и дыр внутри отверстий), контуры описываются один за другим и каждому предшествует вызов `gluNextContour()`. Процедура `gluTessVertex()` вызывается для каждой вершины контура.

Для `gluNextContour()` аргумент *type* может быть равным `GLU_EXTERIOR`, `GLU_INTERIOR`, `GLU_CCW`, `GLU_CW` или `GLU_UNKNOWN`. Эти параметры несут только рекомендательный характер. Если вы зададите правильные значения, процесс разбиения ускорится. Неправильные значения будут проигнорированы, но разбиение будет продолжено. Для многоугольников с отверстиями один контур считается внешним, а другой — внутренним. Первый контур подразумевает тип `GLU_EXTERIOR`. Ориентация (по часовой стрелке или против) выбирается произвольно в трех измерениях; однако для любой плоскости возможны две различные ориентации, поэтому типы `GLU_CCW` и `GLU_CW` должны использоваться непротиворечиво. Используйте `GLU_UNKNOWN`, если не знаете что делать.

Строго рекомендуется преобразовывать код GLU 1.0/1.1 к новому интерфейсу тесселяции GLU 1.2 в следующем порядке:

1. Измените тип ссылки на главную структуру данных с GLUtriangulatorObj на GLUtesselator. В GLU 1.2, GLUtriangulatorObj и GLUtesselator являются одним и тем же типом.
2. Разделите вызов `gluBeginPolygon()` на две команды: `gluTessBeginPolygon()` и `gluTessBeginContour()`. Все контуры, включая первый, должны быть явно начаты.
3. Распределите обязанности `gluNextContour()` между `gluTessEndContour()` и `gluTessBeginContour()`. Вы должны закончить предыдущий контур перед началом описания следующего.
4. Преобразуйте `gluEndPolygon()` в `gluTessEndContour()` и `gluTessEndPolygon()`. Конечный контур должен быть замкнутым.
5. Переопределите применение параметров в `gluTessCallback()`. В GLU 1.2 `GLU_BEGIN`, `GLU_VERTEX`, `GLU_END`, `GLU_ERROR` и `GLU_EDGE_FLAG` считаются синонимами `GLU_TESS_BEGIN`, `GLU_TESS_VERTEX`, `GLU_TESS_END`, `GLU_TESS_ERROR` и `GLU_TESS_EDGE_FLAG`.

Поверхности второго порядка: сферы, цилиндры и диски

Основная библиотека OpenGL предоставляет поддержку для моделирования и отображения простых точек, линий и выпуклых закрашенных многоугольников. Ни 3D-объекты, ни наиболее часто используемые 2D-объекты, такие как окружности, непосредственно недоступны.

На протяжении всей книги трехмерные объекты создавались с помощью библиотеки GLUT. Библиотека GLU также предоставляет средства для моделирования и мозаичного разбиения, аппроксимации многоугольниками разнообразных 2D- и 3D-фигур (сфер, цилиндров, дисков и неполных дисков), которые могут быть обсчитаны уравнениями второго порядка. К таким средствам относятся процедуры для рисования поверхностей второго порядка (квадратичных) с различными стилями и ориентацией. Квадратичные поверхности удовлетворяют следующему общему уравнению:

$$a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5yx + a_6xz + a_7x + a_8y + a_9z + a_{10} = 0.$$

(См. *Procedural Elements for Computer Graphics* Дэвида Роджерса (David Rogers), Нью-Йорк, издательство McGraw-Hill, 1985.) Создание и отображение поверхностей второго порядка напоминает тесселяцию. Порядок использования квадратичных объектов следующий:

1. Чтобы создать квадратичный объект, используйте команду `gluNewQuadric()`.
2. Задайте атрибуты воспроизведения объекта (если вас не устраивают значения по умолчанию):
   - используйте `gluQuadricOrientation()` для управления направлением «намотки» и для разграничения внутренней и внешней областей;

- используйте `gluQuadricDrawStyle()` для выбора между воспроизведением объекта в виде точек, линий или закрашенных многоугольников;
- для освещенных квадратичных объектов используйте `gluQuadricNormals()` для указания одной нормали на вершину или одной нормали на грань. По умолчанию нормали не генерируются;
- для текстурных квадратичных объектов используйте `gluQuadricTexture()`, если хотите генерировать координаты текстуры.

3. Подготовьтесь к потенциальным проблемам — зарегистрируйте обработчик ошибок с помощью `gluQuadricCallback()`. Тогда при возникновении ошибки в процессе воспроизведения будет вызвана указанная вами процедура.
4. Затем вызовите процедуру для воспроизведения, соответствующую типу квадратичного объекта: `gluSphere()`, `gluCylinder()`, `gluDisk()` или `gluPartialDisk()`. Чтобы увеличить производительность для статичных данных, инкапсулируйте квадратичный объект в список отображения.
5. По окончании работы с квадратичным объектом уничтожьте его с помощью вызова `gluDeleteQuadric()`. Если вы намерены обрабатывать другие фигуры второго порядка, лучше повторно использовать уже имеющийся квадратичный объект.

Управление квадратичными объектами

Квадратичный объект определяется значениями аргументов, атрибутов функциями обратной связи. Все эти данные сохраняются в структуре типа GLUquadricObj. Квадратичный объект может генерировать вершины, нормали, координаты текстуры и другие данные, которые все могут быть использованы сразу же или быть сохранены в списке отображения для последующего применения. Представленные ниже процедуры отвечают за создание, уничтожение квадратичного объекта и за сообщения об ошибках в процессе воспроизведения.

GLUquadricObj* `gluNewQuadric`(void);

Создает новый квадратичный объект и возвращает указатель на него. При неудаче возвращается нулевой указатель.

void `gluDeleteQuadric`(GLUquadricObj **qobj*);

Разрушает квадратичный объект (*qobj*) и освобождает занимаемую им память.

void `gluQuadricCallback`(GLUquadricObj **qobj*, GLenum *which*, void (**fn*)());

Определяет функцию обратной связи (*fn*), вызываемую при особых обстоятельствах. Константа `GLU_ERROR` — это единственное возможное значение для аргумента *which*, поэтому *fn* вызывается в момент возникновения ошибки. Если значение *fn* равно `NULL`, из обработки исключается существующая функция обратной связи.

Для `GLU_ERROR` функция *fn* вызывается с одним аргументом — кодом ошибки. Для преобразования кода ошибки в ASCII-строку можно использовать команду `gluErrorString()`.

Управление атрибутами квадратичных объектов

Следующие процедуры имеют отношение к данным, генерируемым процедурами квадратичных объектов. Используйте их до того, как будете задавать примитивы. Листинг 11.4, программа `quadric.c`, демонстрирует изменения стиля рисования и вида генерируемых нормалей, а также создание квадратичных объектов, обработку ошибок и рисование примитивов.

void `gluQuadricDrawStyle`(GLUquadricObj **qobj*, GLenum *drawStyle*);
Управляет стилем (*drawStyle*) воспроизведения квадратичного объекта (*qobj*). Допустимые значения стиля: `GLU_POINT`, `GLU_LINE`, `GLU_SILHOUETTE` и `GLU_FILL`.

Константы `GLU_POINT` и `GLU_LINE` указывают, что примитивы должны быть отображены как точка в каждой вершине или как линия, соединяющая пару вершин.

Константа `GLU_SILHOUETTE` указывает, что примитивы отображаются как линии, за исключением ребер, разделяющих соседние грани, которые не отображаются. Данный вариант часто используется командами `gluDisk()` и `gluPartialDisk()`.

Константа `GLU_FILL` указывает, что отображение выполняется закрашенными многоугольниками, которые рисуются против часовой стрелки с учетом своих нормалей. Это может быть эффективно для `gluQuadricOrientation()`.

void `gluQuadricOrientation`(GLUquadricObj **qobj*, GLenum *orientation*);
Для квадратичного объекта (*qobj*) аргументу *orientation* может быть присвоено значение `GLU_OUTSIDE` (наружу — по умолчанию) или `GLU_INSIDE` (внутрь); эти константы управляют направлением нормали.

Для `gluSphere()` (сферы) и `gluCylinder()` (цилиндра) понятия «наружу» и «вовнутрь» очевидны. Для `gluDisk()` и `gluPartialDisk()` (диска) сторона с положительным значением *z* является наружной.

void `gluQuadricNormals`(GLUquadricObj **qobj*, GLenum *normals*);
Для квадратичного объекта *qobj* аргумент «нормали» (*normals*) — `GLU_NONE` (по умолчанию), `GLU_FLAT` или `GLU_SMOOTH`.

Функция `gluQuadricNormals()` используется для указания того, где генерируются векторы нормалей. Константа `GLU_NONE` означает, что нормали не генерируются, и нацелена на использование без освещения. Константа `GLU_FLAT` соответствует одной нормали на поверхность — это часто предпочтительно для имитации освещения с «плоским» затенением. При задании `GLU_SMOOTH` генерируется одна нормаль на каждую вершину квадратичного объекта; такой вариант подходит для освещения с плавным затенением.

void `gluQuadricTexture`(GLUquadricObj **qobj*, GLboolean *textureCoords*);
Для квадратичного объекта *qobj* аргумент *textureCoords* задается равным `GL_FALSE` (по умолчанию) или `GL_TRUE`. В последнем случае для квадратичного объекта генерируются координаты текстуры. Манера, в которой это делается, зависит от типа квадратичного объекта.

Квадратичные примитивы

Следующие подпрограммы непосредственно генерируют вершины и другие данные, составляющие квадратичный объект. Во всех случаях аргумент *qobj* ссылается на квадратичный объект, созданный вызовом `gluNewQuadric()`.

void `gluSphere`(GLUquadricObj **qobj*, GLdouble *radius*, GLint *slices*, GLint *stacks*);

Рисует сферу с заданным радиусом (*radius*), с центром в начале координат (0, 0, 0). Вокруг оси *z* сфера разбивается на элементы *slices* (подобные географической долготе), а вдоль оси *z* — откладывается количество *stacks* (аналог географической широты).

Если координаты текстуры генерируются средствами квадратичного объекта, *t*-координата принадлежит диапазону от 0.0 при $z = -radius$ до 1.0 при $z = radius$, и *t* наращивается линейно вдоль продольных линий. При этом значение *s* лежит в диапазоне от 0.0 для оси $+y$, до 0.25 — для оси $+x$, до 0.5 — для оси $-y$, до 0.75 —для оси $-x$ и назад до 1.0 — для оси $+y$.

void `gluCylinder`(GLUquadricObj **qobj*, GLdouble *baseRadius*, GLdouble *topRadius*, GLdouble *height*, GLint *slices*, GLint *stacks*);

Рисует цилиндр, ориентированный вдоль оси *z*, с основанием $z = 0$ и верхней гранью в $z = height$. Подобно сфере, цилиндр разбивается на *slices* вокруг оси *z* и вдоль этой оси откладывается количество *stacks*. Аргумент *baseRadius* — это радиус цилиндра в основании ($z = 0$). Аргумент *topRadius* — радиус цилиндра в его верхней точке ($z = height$). Если значение *topRadius* равно 0, генерируется конус.

Если координаты текстуры генерируются средствами квадратичного объекта, *t*-координата изменяется линейно от 0.0 при $z = 0$ до 1.0 при $z = height$. Координата *s* текстуры генерируется так же, как для сферы.

ПРИМЕЧАНИЕ

Цилиндр не «закрывается» ни сверху, ни снизу, то есть диски в основании и в верхней точке не рисуются.

void `gluDisk`(GLUquadricObj **qobj*, GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *slices*, GLint *rings*);

Рисует диск в плоскости $z = 0$ с радиусом *outerRadius* и концентрическим круговым отверстием радиусом *innerRadius*. Если *innerRadius* равен 0, отверстие не создается. Диск разбивается вокруг оси *z* на количество «ломтиков» *slices* (подобно ломтикам пиццы), а вдоль оси *z* — на концентрические кольца (*rings*).

С учетом стандартной ориентации сторона $+z$ диска считается наружной, то есть любые генерируемые нормали указывают в направлении $+z$. Иначе нормали направлены в направлении $-z$.

Если координаты текстуры генерируются средствами квадратичного объекта, они изменяются линейно таким образом, что R = *outerRadius*, значения координат *s* и *t* при (R, 0, 0) равны (1, 0.5), при (0, R, 0) — (0.5, 1), при (−R, 0, 0) — (0, 0.5) и при (0, −R, 0) — (0.5, 0).

void `gluPartialDisk`(GLUquadricObj **qobj*, GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *slices*, GLint *rings*, GLdouble *startAngle*, GLdouble *sweepAngle*);

Рисует неполный диск в плоскости $z = 0$. К неполному диску применимы те же термины, что и для полного диска: *outerRadius*, *innerRadius*, *slices* и *rings*. Разница только в том, что неполный диск рисуется от начального угла *startAngle* до *startAngle* + *sweepAngle* (*startAngle* и *sweepAngle* измеряются в градусах, где 0° это ось $+y$, 90° соответствует оси $+x$, 180° — ось $-y$ и 270° — ось $-x$).

Ориентация и координаты текстуры для неполного диска определяются аналогично полному диску.

ПРИМЕЧАНИЕ

Для всех квадратичных объектов лучше использовать *Radius, height и подобные параметры для масштабирования, чем команду glScale*(), чтобы сгенерированные единичные нормали не приходилось нормализовать снова. Задавайте значения аргументов rings и stacks отличными от 1 с целью лучшей гранулярности при расчете освещения, особенно если материал обладает высоким альбедо.

Листинг 11.4 показывает рисование каждого из квадратичных примитивов, а также влияние различных стилей.

Листинг 11.4. Квадратичные объекты: quadric.c

```
#ifndef CALLBACK
#define CALLBACK
#endif

GLuint startList;
void CALLBACK errorCallback(GLenum errorCode)
{
   const GLubyte *estring;

   estring = gluErrorString(errorCode);
   fprintf(stderr, "Ошибка %s\n", estring);
   exit(0);
}

void init(void)
{
   GLUquadricObj *qobj;
   GLfloat mat_ambient[] = { 0.5, 0.5, 0.5, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 50.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
   GLfloat model_ambient[] = { 0.5, 0.5, 0.5, 1.0 };

   glClearColor(0.0, 0.0, 0.0, 0.0);

   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, model_ambient);
```

продолжение ⇩

Листинг 11.4 (*продолжение*)

```
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

/* Создает 4 списка отображения по числу квадратичных объектов.
 * Демонстрируются различные стили рисования и описания
 * нормалей к поверхности */
   startList = glGenLists(4);
   qobj = gluNewQuadric();
   gluQuadricCallback(qobj, GLU_ERROR, errorCallback);

   gluQuadricDrawStyle(qobj, GLU_FILL); /* плавное затенение */
   gluQuadricNormals(qobj, GLU_SMOOTH);
   glNewList(startList, GL_COMPILE);
      gluSphere(qobj, 0.75, 15, 10);
   glEndList();

   gluQuadricDrawStyle(qobj, GLU_FILL); /* плоское затенение */
   gluQuadricNormals(qobj, GLU_FLAT);
   glNewList(startList+1, GL_COMPILE);
      gluCylinder(qobj, 0.5, 0.3, 1.0, 15, 5);
   glEndList();

   gluQuadricDrawStyle(qobj, GLU_LINE); /* каркас */
   gluQuadricNormals(qobj, GLU_NONE);
   glNewList(startList+2, GL_COMPILE);
      gluDisk(qobj, 0.25, 1.0, 20, 4);
   glEndList();

   gluQuadricDrawStyle(qobj, GLU_SILHOUETTE);
   gluQuadricNormals(qobj, GLU_NONE);
   glNewList(startList+3, GL_COMPILE);
      gluPartialDisk(qobj, 0.0, 1.0, 20, 4, 0.0, 225.0);
   glEndList();
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glPushMatrix();

   glEnable(GL_LIGHTING);
   glShadeModel(GL_SMOOTH);
   glTranslatef(-1.0, -1.0, 0.0);
   glCallList(startList);
   glShadeModel(GL_FLAT);
   glTranslatef(0.0, 2.0, 0.0);
   glPushMatrix();
   glRotatef(300.0, 1.0, 0.0, 0.0);
   glCallList(startList+1);
   glPopMatrix();
```

```
   glDisable(GL_LIGHTING);
   glColor3f(0.0, 1.0, 1.0);
   glTranslatef(2.0, -2.0, 0.0);
   glCallList(startList+2);

   glColor3f(1.0, 1.0, 0.0);
   glTranslatef(0.0, 2.0, 0.0);
   glCallList(startList+3);

   glPopMatrix();
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-2.5, 2.5, -2.5*(GLfloat)h/(GLfloat)w,
              2.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho(-2.5*(GLfloat)w/(GLfloat)h,
              2.5*(GLfloat)w/(GLfloat)h, -2.5, 2.5, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
         exit(0);
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Вычислители и NURBS

12

Прочитав эту главу, вы научитесь:

- Использовать команды вычислителя OpenGL для рисования основных кривых и поверхностей
- Использовать средства NURBS высокого уровня библиотеки GLU для рисования более сложных кривых и поверхностей, возвращать данные вершин геометрических объектов, получаемых при мозаичном разбиении кривых и поверхностей

На самом низком уровне аппаратные средства рисуют точки, отрезки линий и многоугольники — обычно треугольники и четырехугольники. Гладкие кривые и поверхности рисуются аппроксимированными большим числом маленьких отрезков или многоугольников. Тем не менее многие полезные кривые и поверхности могут быть описаны математически небольшим числом параметров, таких как *контрольные точки*. Сохранение 16 контрольных точек для поверхности требует значительно меньше памяти, чем сохранение 1000 треугольников вместе с информацией о векторах нормали для каждой вершины. Кроме того, 1000 треугольников лишь аппроксимируют истинную поверхность, а контрольные точки точно описывают.

Вычислители обеспечивают способ указания точек кривой или поверхности (или ее части) с помощью только контрольных точек. Кривая или поверхность может быть воспроизведена с любой точностью. Дополнительно автоматически вычисляются векторы нормалей к поверхностям. Вы можете использовать данные, сгенерированные вычислителем, различными способами: нарисовать точки в месте нахождения поверхности, построить проволочный каркас поверхности или отобразить ее полную версию — с освещением, затенением и текстурами.

Вычислители позволяют описать любые полиномиальные или рациональные (rational) полиномиальные сплайны или поверхности любого порядка. Это практически все сплайны и сплайновые поверхности, применяемые сегодня, в том числе B-сплайны, NURBS- (Non-Uniform Rational B-Spline — неоднородный рациональный B-сплайн) поверхности, кривые и поверхности Безье и сплайны Эрмита (Hermite). Поскольку вычислители предоставляют только низкоуровневое описание точек кривой или поверхности, они обычно используются «вышестоящими» библиотеками инструментов, обеспечивающими интерфейс программирования более высокого уровня. Средства NURBS библиотеки GLU — как раз такой интерфейс. Процедуры NURBS инкапсулируют в себе большой объем сложного кода. Конечное отображение обычно выполняется вычислителями, но в некоторых ситуациях, например при вырезании (trimming) кривых, NURBS-процедуры используются для воспроизведения плоских многоугольников.

Эта глава содержит следующие основные разделы:

- **Необходимые знания.** Здесь говорится, какие знания вы уже должны иметь для чтения этой главы. Здесь также приводятся несколько ссылок на дополнительные источники информации.
- **Вычислители.** Рассказывает, как работают вычислители и как ими управлять с помощью команд OpenGL.
- **Интерфейс NURBS библиотеки GLU.** Описывает процедуры GLU для создания NURBS-кривых и поверхностей, их воспроизведения и возврата информации для вершин, полученных при мозаичном разбиении.

Необходимые знания

Вычислители рассчитывают сплайны и поверхности, основанные на кривых Безье (или на полиномах Бернштейна).

В этой главе приводятся определяющие уравнения, но обсуждение не отклоняется от основной линии и даже не включает в себя перечисление всех интересных свойств этих объектов. Если вы хотите использовать вычислители для рисования кривых и поверхностей с помощью других базисов, вы должны уметь выполнять их преобразование к базису Безье. Кроме этого, когда вы воспроизводите поверхность Безье или ее часть с помощью этих вычислителей, вам нужно определиться с детализацией разбиения. Ваше решение должно быть направлено в сторону компромисса между качеством разбиения изображений и быстродействием. Выработка соответствующей стратегии разбиения может оказаться весьма непростым делом, даже чересчур сложным для обсуждения в рамках книги.

Соответственно, подробное обсуждение NURBS также выходит за «область видимости» этой книги. Выдержки из документации к интерфейсу GLU NURBS и примеры программ, представленные здесь, ориентированы на читателей, уже знакомых с предметом. То есть вы уже должны иметь представление о контрольных точках NURBS, последовательностях узлов (knots) и вырезании (trimming) кривых.

Если вы еще не имеете таких знаний, обратитесь за помощью к следующим руководствам:

- Фарин Джеральд (Farin Gerald E.), *Curves and Surfaces for Computer-Aided Geometric Design*, Fourth Edition. San Diego, CA: Academic Press, 1996;
- Фарин Джеральд (Farin, Gerald E.), *NURB Curves and Surfaces: from Projective Geometry to Practical Use*. Wellesley, MA: A. K. Peters Ltd., 1995;
- Фарин Джеральд (Farin, Gerald E., editor), *NURBS for Curve and Surface Design*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991;
- Хосчек Джозеф (Hoschek, Josef) и Дитер Лессер (Dieter Lasser), *Fundamentals of Computer Aided Geometric Design*. Wellesley, MA: A. K. Peters Ltd., 1993;
- Пигл Лес (Piegl, Les) и Уэйн Тиллер (Wayne Tiller), *The NURBS Book*. New York, NY: Springer–Verlag, 1995.

ПРИМЕЧАНИЕ

Некоторые термины, употребляемые в этой главе, могут выступать в несколько ином значении относительно другой литературы по сплайнам, кривым и поверхностям, поскольку полное согласие среди практиков в этой области не достигнуто. Вообще говоря, терминология здесь отталкивается от возможностей OpenGL. Например, вычислители OpenGL всегда опираются на кривые Безье; в целом же вычислитель вправе использовать произвольный базис.

Вычислители

Кривая Безье — это векторнозначная (vector-valued) функция одной переменной:

$$\mathbf{C}(u) = [\mathbf{X}(u)\ \mathbf{Y}(u)\ \mathbf{Z}(u)],$$

где u изменяется в некоторых пределах (скажем, в интервале [0, 1]). Лоскут (patch) поверхности Безье — это векторнозначная функция двух переменных:

$$\mathbf{S}(u,v) = [\mathbf{X}(u,v)\ \mathbf{Y}(u,v)\ \mathbf{Z}(u,v)].$$

где u и v варьируются в некоторых пределах. Область не обязательно должна быть трехмерной, как показано здесь. Вы вправе применить двухмерный вывод для кривых на плоскости или координаты текстуры, или же четырехмерный вывод для передачи RGBA-информации. Даже одномерный вывод может иметь смысл для градаций серого.

Для каждого значения u (или u и v, в случае поверхности) формула для `C()` (или `S()`) вычисляет положение точки кривой (или поверхности). В этом случае вычислители сначала определяют функцию `C()` или `S()`, включают ее, и затем уже обращаются к команде `glEvalCoord1()` или `glEvalCoord2()` вместо `glVertex*()`. Здесь вершины кривой или поверхности могут использоваться, как и любые другие вершины, например, для формирования точек или линий. Кроме того, другие команды автоматически генерируют серию вершин, образующих однородную периодическую сетку по u (или по u и v). Одномерные и двухмерные вычислители действуют аналогично. А поскольку одномерный случай проще для понимания, обсуждение мы начнем именно с него.

Одномерные вычислители

В этом подразделе представлен пример использования одномерного вычислителя для рисования кривой. Пример включает в себя команды и уравнения, управляющие вычислителем.

Пример: простая кривая Безье

Программа, приведенная в листинге 12.1, рисует кубическую кривую Безье по четырем контрольным точкам, как показано на рис. 12.1.

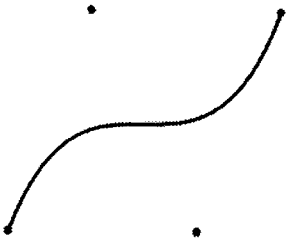


Рис. 12.1. Кривая Безье

Листинг 12.1. Кривая Безье для четырех контрольных точек: bezcurve.c

```
GLfloat ctrlpoints[4][3] = {
                { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
                {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};
void init(void)

  glClearColor(0.0, 0.0, 0.0, 0.0);
```

продолжение ⇙

Листинг 12.1 (*продолжение*)

```
   glShadeModel(GL_FLAT);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}

void display(void)
{
   int i;

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glBegin(GL_LINE_STRIP);
   for (i = 0; i <= 30; i++)
      glEvalCoord1f((GLfloat) i/30.0);
   glEnd();

/* Код, отображающий контрольные точки */
   glPointSize(5.0);
   glColor3f(1.0, 1.0, 0.0);
   glBegin(GL_POINTS);
   for (i = 0; i < 4; i++)
      glVertex3fv(&ctrlpoints[i][0]);
   glEnd();
   glFlush();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
              5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
   else
      glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
              5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

Кубическая кривая Безье описывается четырьмя контрольными точками, в нашем примере — массивом `ctrlpoints[][]`. Этот массив подается на вход процедуре `glMap1f()`. Другие ее аргументы следующие:

`GL_MAP1_VERTEX_3`	Устанавливает, что используются трехмерные контрольные точки и что должны быть образованы трехмерные вершины
0.0	Нижнее значение параметра *u*
1.0	Верхнее значение параметра *u*
3	Количество значений с плавающей точкой в данных между последовательными контрольными точками
4	Порядок сплайна, то есть порядок кривой плюс 1: в нашем случае степень кривой равна 3 (кубическая)
&ctrlpoints[0][0]	Указатель на данные первой контрольной точки

Заметьте, что второй и третий аргументы управляют параметризацией кривой — диапазон переменной *u* (от 0,0 до 1,0) соответствует всей кривой от ее начала до конца. Вызов `glEnable()` разрешает применение одномерного вычислителя для трехмерных вершин.

Кривая рисуется процедурой `display()` — ее частью между командами `glBegin()` и `glEnd()`. Поскольку вычислитель включен, команда `glEvalCoord1f()` просто влечет вызов `glVertex()` с координатами вершин кривой, соответствующих входному параметру *u*.

Определение одномерного вычислителя и вычисления для него

Многочлен Бернштейна степени n (или порядка $n + 1$) выглядит следующим образом

$$B_i^n(u) = \binom{n}{i} u^i (1 - u)^{n-i}.$$

Если P_i — это набор контрольных точек (одномерных, двух-, трех- и даже четырехмерных), уравнение

$$C(u) = \sum_{i=0}^{n} B_i^n(u) P_i$$

представляет кривую Безье, где u изменяется от 0.0 до 1.0. Чтобы представить эту же кривую, но с диапазоном для u от u_1 до u_2, вычислим

$$C\left(\frac{u - u_1}{u_2 - u_1}\right).$$

Команда `glMap1()` определяет одномерный вычислитель, используемый для этих вычислений.

void `glMap1`{fd}(GLenum *target*, *TYPE u1*, *TYPE u2*, GLint *stride*, GLint *order*, *const TYPE *points);*

Определяет одномерный вычислитель. Аргумент *target* задает, как представляются контрольные точки, как показано в табл. 12.1, и тем самым указывает, сколько значений соответствует массиву *points*. Эти точки могут быть пред-

ставлены вершинами, цветовыми данными RGBA, векторами нормалей или координатами текстур. Например, с помощью GL_MAP1_COLOR_4 вычислитель генерирует цветовые данные кривой в четырехмерном (RGBA) цветовом пространстве. Вы можете также использовать значения параметров из табл. 12.1 для того, чтобы разрешить применение вычислителей перед их вызовом. Передача соответствующего значения в glEnable() или glDisable() включает или выключает вычислитель.

Аргументы *u1* и *u2* указывают диапазон значений переменной *u*. Переменная шага по индексу (*stride*) — это количество значений с одинарной или двойной точностью в каждом блоке памяти. Таким образом, она задает смещение между данными двух последовательных точек.

Порядок *order* — это степень многочлена плюс 1, он должен соответствовать количеству контрольных точек. Указатель *points* ссылается на первую координату первой контрольной точки. Для нашего примера нужно использовать следующую структуру *points*: (GLfloat *)(&ctlpoints[0].x).

Таблица 12.1. Типы контрольных точек для glMap1*()

Параметр	Пояснение
GL_MAP1_VERTEX_3	x-, y-, z-координаты вершин
GL_MAP1_VERTEX_4	x-, y-, z- и w-координаты вершин
GL_MAP1_INDEX	Индекс цвета
GL_MAP1_COLOR_4	R, G, B, A
GL_MAP1_NORMAL	Координаты нормали
GL_MAP1_TEXTURE_COORD_1	s-координата текстуры
GL_MAP1_TEXTURE_COORD_2	s- и t-координаты текстуры
GL_MAP1_TEXTURE_COORD_3	s-, t- и r-координаты текстуры
c	s-, t-, r- и q-координаты текстуры

Одновременно могут использоваться несколько вычислителей. Так, если определены и активны вычислители GL_MAP1_VERTEX_3 и GL_MAP1_COLOR_4, вызовы glEvalCoord1() генерируют сразу позицию и цвет. Но одномоментно допустимо задействовать только один вычислитель для вершин, хотя определить можно оба. Аналогично, в текущий момент активен может быть лишь один вычислитель текстуры. Вместе с тем вычислители могут быть задействованы для генерирования любого сочетания вершин, нормалей, цвета и координатных данных текстур. Если вы определите и включите более одного вычислителя одинакового типа, будет использоваться предполагающий большую размерность.

Для обсчета уже определенного и активного одномерного отображения используйте команду glEvalCoord1*().

void glEvalCoord1{fd}(*TYPE u*);
void glEvalCoord1{fd}v(*TYPE *u*);
Запускают вычисления ранее определенных одномерных отображений. Аргумент *u* — это значение (или указатель на значение в векторной версии команды) координаты области.

В процессе вычисления для обрабатываемых вершин генерируются значения цвета, индекса цвета, векторов нормалей и координат текстур. Вызовы `glEvalCoord*()` не используются для текущих значений всех упомянутых данных. Команда `glEvalCoord*()` не изменяет текущие значения.

Определение равноотстоящих координат в одном измерении

Вы можете использовать `glEvalCoord1()` с любыми значениями *u*, но на практике чаще применяются равноотстоящие (эквидистантные), как в листинге 12.1. Чтобы получить такие значения, определите одномерную сетку с помощью `glMapGrid1*()` и тогда уже применяйте к ней `glEvalMesh1()`.

void `glMapGrid1`{fd}(GLint *n*, *TYPE u1*, *TYPE u2*);
Определяет сетку между значениями *u1* и *u2* с шагом *n*, то есть эквидистантную.

void `glEvalMesh1`(GLenum *mode*, GLint *p1*, GLint *p2*);
Применяет текущую определенную сетку ко всем включенным вычислителям. Режим *mode* выбирается из двух: `GL_POINT` или `GL_LINE`, в зависимости от того, как вы хотите нарисовать кривую — точками или соединительными линиями. Этот вызов действует так же, как `glEvalCoord1()` для каждого шага от *p1* до *p2* включительно, где $0 \le p1, p2 \le n$. Программно это эквивалентно коду

```
glBegin(GL_POINTS); /* или glBegin(GL_LINE_STRIP); */
for (i = p1; i <= p2; i++)
glEvalCoord1(u1 + i*(u2-u1)/n);
glEnd();
```

с той лишь разницей, что, если $i = 0$ или $i = n$, команда `glEvalCoord1()` вызывается со значениями *u1* или *u2*.

Двухмерные вычислители

Действия в двух измерениях подобны одномерному случаю, за исключением того, что все команды должны принимать два аргумента — *u* и *v*.

И, конечно, точки, цвета, нормали или координаты текстур должны относиться к поверхности, а не к кривой. Математически фрагмент поверхности Безье представляется так:

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) P_{ij},$$

где P_{ij} — это каждая из $m \times n$ контрольных точек, а функции B — те же самые многочлены Бернштейна для одного измерения. Как и раньше, значения P_{ij} могут представлять вершины, нормали, цвета или координаты текстуры.

Процедура использования двухмерных вычислителей подобна одномерному случаю.

1. Определите вычислитель(-и) с помощью `glMap2*()`.
2. Включите их передачей соответствующего значения в `glEnable()`.
3. Вызовите их посредством `glEvalCoord2()` (между `glBegin()` и `glEnd()`) или указанием и последующим применением сетки с помощью `glMapGrid2()` и `glEvalMesh2()`.

Определения двухмерного вычислителя и вычисления для него

Используйте команды `glMap2*()` и `glEvalCoord2*()` для определения и вызова двухмерных вычислителей.

void `glMap2`{fd}(GLenum *target*, *TYPE u1*, *TYPE u2*, GLint *ustride*, GLint *uorder*, *TYPE v1*, *TYPE v2*, GLint *vstride*, GLint *vorder*, *TYPE points*);

Аргумент *target* может принимать значения из табл. 12.1, с той разницей, что подстрока `MAP1` замещается на `MAP2`. Как и прежде, эти значения также используются с `glEnable()` для включения соответствующего вычислителя. Минимальное и максимальное значения *u* и *v* задаются как *u1*, *u2*, *v1* и *v2*. Независимые аргументы *ustride* и *vstride* задают количество значений с одинарной и двойной точностью между значениями переменных, позволяя пользователю выбрать прямоугольник контрольных точек из большего массива. Например, если данные представлены в виде `GLfloat ctlpoints[100][100][3]` и вы хотите использовать поднабор 4 × 4, начинающийся с `ctlpoints[20][30]`, установите *ustride* равным `100*3` и *vstride* равным 3. Начальная точка, *points*, должна быть задана как `&ctlpoints[20][30][0]`. Наконец, порядки *uorder* и *vorder* могут быть различны; тем самым поверхность можно «сшивать» одновременно, например, из кубических и квадратичных лоскутов.

void `glEvalCoord2`{fd}(*TYPE u*, *TYPE v*);
void `glEvalCoord2`{fd}v(*TYPE *values*);

Запускают вычисления для определенных ранее двухмерных отображений. Аргументы *u* и *v* — это значения (или указатели на значения *u* и *v* в векторной версии команды) координат области. Если включены вычислители для вершин (`GL_MAP2_VERTEX_3` или `GL_MAP2_VERTEX_4`), нормаль к поверхности вычисляется аналитически. Эта нормаль связывается со сгенерированными вершинами, когда активна автоматическая генерация нормалей, включаемая передачей константы `GL_AUTO_NORMAL` в `glEnable()`. В противном случае для порождения нормали используется соответствующее доступное отображение. Если такого отображения нет, используется текущая нормаль.

Пример: поверхность Безье

Программа из листинга 12.2 рисует каркас поверхности Безье с помощью вычислителей, выглядящий так, как на рис. 12.2. В этом примере поверхность образуется девятью кривыми линиями в каждом направлении. Каждая кривая рисуется 30 отрезками. Для получения законченной программы добавьте в пример процедуры `reshape()` и `main()` из листинга 12.1.

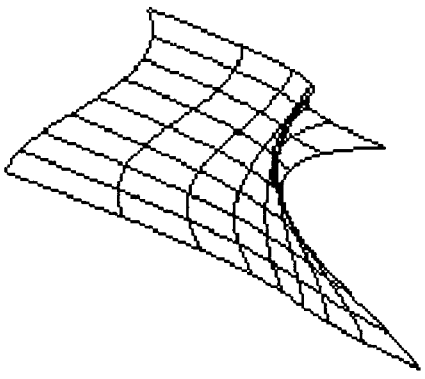


Рис. 12.2. Поверхность Безье

Листинг 12.2. Поверхность Безье: bezsurf.c

```
GLfloat ctrlpoints[4][4][3] = {
                  {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
                   {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
                  {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
                   {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
                  {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
                   {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
                  {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
                   {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void display(void)
{
   int i, j;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glPushMatrix();
   glRotatef(85.0, 1.0, 1.0, 1.0);
   for (j = 0; j <= 8; j++) {
      glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
         glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
      glEnd();
      glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
         glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
      glEnd();
   }
   glPopMatrix();
   glFlush();
}

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, &ctrlpoints[0][0][0]);
   glEnable(GL_MAP2_VERTEX_3);
   glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
}
```

Определение равноотстоящих координат в двух измерениях

«Двухмерные» команды `glMapGrid2*()` и `glEvalMesh2()` подобны «одномерным» версиям, с той разницей, что первые оперируют двумя переменными — *u* и *v*.

void `glMapGrid2`{fd}(GLint *nu*, *TYPE u1*, *TYPE u2*, GLint *nv*, *TYPE v1*, *TYPE v2*);
void `glEvalMesh2`(GLenum *mode*, GLint *i1*, GLint *i2*, GLint *j1*, GLint *j2*);
Определяет двухмерную карту сетки, от *u1* до *u2* с шагом *nu*, и от *v1* до *v2* с шагом *nv* (`glMapGrid2*()`), а затем эта сетка применяется ко всем включен-

ным вычислителям (`glEvalMesh2()`). Единственное существенное отличие от одномерного варианта в том, что для `glEvalMesh2()` аргументом *mode* можно задать константу `GL_FILL`, а не только `GL_POINT` или `GL_LINE`. Эта константа вызывает генерирование закрашенных полигонов с помощью примитивов четырехугольной сетки. Если говорить точнее, `glEvalMesh2()` — ближайший эквивалент одного из следующих трех фрагментов кода. (По той причине, что, когда i равна nu или j — nv, параметр точно равен $u2$ или $v2$, но не $u1 + nu \times (u - u1)/nu$, которое может несколько отличаться вследствие ошибки округления.)

```
glBegin(GL_POINTS); /* режим == GL_POINT */
for (i = nu1; i <= nu2; i++)
   for (j = nv1; j <= nv2; j++)
   glEvalCoord2(u1 + i*(u2 - u1)/nu, v1 + j*(v2 - v1)/nv);
glEnd();
```

или

```
for (i = nu1; i <= nu2; i++) { /* режим == GL_LINE */
glBegin(GL_LINES);
for (j = nv1; j <= nv2; j++)
   glEvalCoord2(u1 + i*(u2 - u1)/nu, v1 + j*(v2 - v1)/nv);
glEnd();
}
for (j = nv1; j <= nv2; j++) {
   glBegin(GL_LINES);
   for (i = nu1; i <= nu2; i++)
      glEvalCoord2(u1 + i*(u2 - u1)/nu, v1 + j*(v2 - v1)/nv);
   glEnd();
}
```

или

```
for (i = nu1; i < nu2; i++) { /* режим == GL_FILL */
   glBegin(GL_QUAD_STRIP);
   for (j = nv1; j <= nv2; j++) {
      glEvalCoord2(u1 + i*(u2 - u1)/nu, v1 + j*(v2 - v1)/nv);
      glEvalCoord2(u1 + (i+1)*(u2-u1)/nu, v1+j*(v2 - v1)/nv);
   glEnd();
}
```

Листинг 12.3 показывает различия, которые нужно внести в листинг 12.2, чтобы нарисовать поверхность Безье, используя `glMapGrid2()` и `glEvalMesh2()` для разбиения квадратной области на равномерную 8 × 8 сетку. В эту программу также добавлены освещение и затенение, а результат их действия продемонстрирован на рис. 12.3.

Листинг 12.3. Освещенные и затененные поверхности Безье с использованием сетки: bezmesh.c

```
void initlights(void)
{
   GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
   GLfloat position[] = {0.0, 0.0, 2.0, 1.0};
   GLfloat mat_diffuse[] = {0.6, 0.6, 0.6, 1.0};
```

```
  GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
  GLfloat mat_shininess[] = {50.0};

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);

  glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  glLightfv(GL_LIGHT0, GL_POSITION, position);

  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glRotatef(85.0, 1.0, 1.0, 1.0);
  glEvalMesh2(GL_FILL, 0, 20, 0, 20);
  glPopMatrix();
  glFlush();
}

void init(void)
{
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glEnable(GL_DEPTH_TEST);
  glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, &ctrlpoints[0][0][0]);
  glEnable(GL_MAP2_VERTEX_3);
  glEnable(GL_AUTO_NORMAL);
  glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
  initlights();
}
```

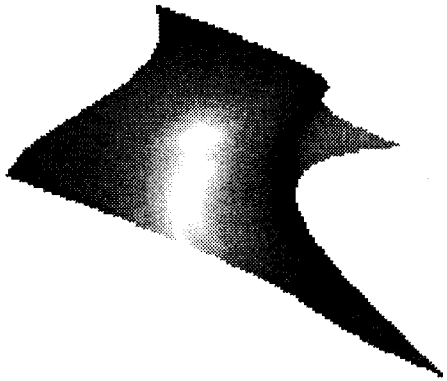


Рис. 12.3. Освещенная и тонированная поверхности Безье, построенные по сетке

Применение вычислителей для текстур

Программа из листинга 12.4 включает в себя два вычислителя, работающие одновременно: первый генерирует трехмерные точки поверхности Безье аналогично листингу 12.3, а второй вырабатывает координаты текстуры. Здесь координаты текстуры те же, что и u- и v-координаты поверхности, но это требует создания специального плоского лоскута.

Плоский фрагмент определяется на квадратной области с углами (0, 0), (0, 1), (1, 0) и (1, 1); значения (0, 0) генерируются в углу (0, 0), (0, 1) — в углу (0, 1) и т. д. Поскольку его порядок равен 2 («линейная» степень плюс 1), при вычислении текстуры в точке (u, v) генерируются координаты (s, t). Это делается одновременно с вычислением вершин, поэтому их можно наблюдать в процессе рисования поверхности (см. цветной рисунок 19). Если вы хотите размножить текстуру по три раза в каждом направлении, измените значение 1.0 в массиве `texpts[][][]` на 3.0. В этом примере поверхность строится из девяти копий карты текстуры.

Листинг 12.4. Применение вычислителей для текстур: texturesurf.c

```
GLfloat ctrlpoints[4][4][3] = {
                  {{ -1.5, -1.5, 4.0}, { -0.5, -1.5, 2.0},
                   {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
                  {{ -1.5, -0.5, 1.0}, { -0.5, -0.5, 3.0},
                   {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
                  {{ -1.5, 0.5, 4.0}, { -0.5, 0.5, 0.0},
                   {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
                  {{ -1.5, 1.5, -2.0}, { -0.5, 1.5, -2.0},
                   {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};
GLfloat texpts[2][2][2] = {{{0.0, 0.0}, {0.0, 1.0}}, {{1.0, 0.0}, {1.0, 1.0}}};

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glEvalMesh2(GL_FILL, 0, 20, 0, 20);
   glFlush();
}
#define imageWidth 64
#define imageHeight 64
GLubyte image[3*imageWidth*imageHeight];

void makeImage(void)
{
   int i, j;
   float ti, tj;

   for (i = 0; i < imageWidth; i++) {
      ti = 2.0*3.14159265*i/imageWidth;
      for (j = 0; j < imageHeight; j++) {
         tj = 2.0*3.14159265*j/imageHeight;
```

```
            image[3*(imageHeight*i+j)] =
               (GLubyte) 127*(1.0+sin(ti));
            image[3*(imageHeight*i+j)+1] =
               (GLubyte) 127*(1.0+cos(2*tj));
            image[3*(imageHeight*i+j)+2] =
               (GLubyte) 127*(1.0+cos(ti+tj));
         }
      }
}

void init(void)
{
   glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, &ctrlpoints[0][0][0]);
   glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2, 0, 1, 4, 2, &texpts[0][0][0]);
   glEnable(GL_MAP2_TEXTURE_COORD_2);
   glEnable(GL_MAP2_VERTEX_3);
   glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
   makeImage();
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, 3, imageWidth, imageHeight, 0,
      GL_RGB, GL_UNSIGNED_BYTE, image);
   glEnable(GL_TEXTURE_2D);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
              4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
   else
      glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
              4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glRotatef(85.0, 1.0, 1.0, 1.0);
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
```

продолжение ⇩

Листинг 12.4 (*продолжение*)

```
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

Интерфейс NURBS библиотеки GLU

Несмотря на то что вычислители — это единственные доступные для непосредственного рисования кривых и поверхностей примитивы OpenGL, это не недостаток, так как они очень эффективно реализованы на аппаратном уровне, а приложения чаще всего обращаются к ним с помощью библиотек более высокого уровня. Одна из них, GLU, предоставляет интерфейс NURBS, являющийся надстройкой над командами вычислителей OpenGL.

Пример простого NURBS-объекта

Если вам не нужно расшифровывать, что такое NURBS, написать для вас код OpenGL, манипулирующий NURBS-кривыми и поверхностями, не составит труда, даже с учетом освещения и наложения текстур.

Во время рисования NURBS-кривых или неусеченных (untrimmed) NURBS-поверхностей (см. раздел «Усечение NURBS-поверхностей») следуйте описанному ниже порядку действий.

1. Если вы намереваетесь осветить NURBS-поверхность, вызовите `glEnable()` с параметром `GL_AUTO_NORMAL` для автоматического генерирования нормалей к поверхности. (Или вычисляйте их самостоятельно.)
2. Когда создаете NURBS-кривую или поверхность, используйте команду `gluNewNurbsRenderer()` для создания указателя на NURBS-объект.
3. При необходимости вызовите `gluNurbsProperty()` для задания параметров построения, таких как максимальный размер линий многоугольников, которыми воспроизводится ваш NURBS-объект. Для команды `gluNurbsProperty()` можно также включить режим, в котором геометрические данные тесселяции могут быть получены с помощью интерфейса обратной связи.
4. Вызовите `gluNurbsCallback()`, если хотите быть уведомлены о случаях возникновения ошибок. (Проверка ошибок способна отрицательно повлиять на производительность, но она настоятельно рекомендуется.) Команда `gluNurbsCallback()` может также использоваться для регистрации функций, возвращающих геометрические данные тесселяции.
5. Начните построение кривой или поверхности с вызова `gluBeginCurve()` или `gluBeginSurface()`.
6. Сгенерируйте и воспроизведите свою кривую или поверхность. Хотя бы раз вызовите `gluNurbsCurve()` или `gluNurbsSurface()` с контрольными точками

(рациональными или нерациональными), последовательностью узлов (knots) и порядком базисного полинома для своего NURBS-объекта. Вы можете делать эти вызовы как дополнительные для задания нормалей к поверхности и/или координат текстур.

7. Вызовите `gluEndCurve()` или `gluEndSurface()`, чтобы закончить обработку кривой или поверхности.

В программе из листинга 12.5 строится NURBS-поверхность в форме симметричной возвышенности, контрольные точки которой принадлежат диапазону от –3.0 до 3.0. В качестве базовой функции взят кубический B-сплайн, но последовательность узлов неоднородна, со значением кратности, равным 4, в конечных точках. Соответственно, базовая функция ведет себя как кривая Безье в каждом направлении. Поверхность освещена, ей приданы темно-серое диффузное (diffuse) отражение и белые зеркальные блики. На рис. 12.4 результирующая поверхность показана также в виде освещенного каркаса.

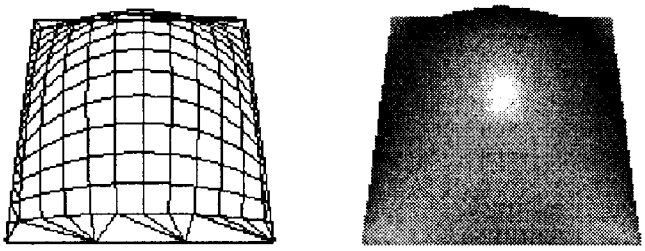


Рис. 12.4. NURBS-поверхность

Листинг 12.5. NURBS-поверхность: surface.c

```
#ifndef CALLBACK
#define CALLBACK
#endif

GLfloat ctlpoints[4][4][3];
int showPoints = 0;

GLUnurbsObj *theNurb;

void init_surface(void)
{
  int u, v;

  for (u = 0; u < 4; u++) {
     for (v = 0; v < 4; v++) {
        ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
        ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);

        if ( (u == 1 || u == 2) && (v == 1 || v == 2))
           ctlpoints[u][v][2] = 3.0;                    продолжение ⇙
```

Листинг 12.5 (*продолжение*)

```
         else
            ctlpoints[u][v][2] = -3.0;
      }
   }
}

void CALLBACK nurbsError(GLenum errorCode)
{
   const GLubyte *estring;

   estring = gluErrorString(errorCode);
   fprintf(stderr, "Ошибка NURBS: %s\n", estring);
   exit(0);
}

void init(void)
{
   GLfloat mat_diffuse[] = { 0.7, 0.7, 0.7, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 100.0 };

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);

   init_surface();

   theNurb = gluNewNurbsRenderer();
   gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
   gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
   gluNurbsCallback(theNurb, GLU_ERROR, nurbsError);
}

void display(void)
{
   GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   int i, j;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix();
   glRotatef(330.0, 1., 0., 0.);
   glScalef(0.5, 0.5, 0.5);
   gluBeginSurface(theNurb);
   gluNurbsSurface(theNurb, 8, knots, 8, knots,
                   4 * 3, 3, &ctlpoints[0][0][0], 4, 4, GL_MAP2_VERTEX_3);
```

```
   gluEndSurface(theNurb);

   if (showPoints) {
      glPointSize(5.0);
      glDisable(GL_LIGHTING);
      glColor3f(1.0, 1.0, 0.0);
      glBegin(GL_POINTS);
      for (i = 0; i < 4; i++) {
         for (j = 0; j < 4; j++) {
            glVertex3f(ctlpoints[i][j][0], ctlpoints[i][j][1],
              ctlpoints[i][j][2]);
         }
      }
      glEnd();
      glEnable(GL_LIGHTING);
   }
   glPopMatrix();
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.0, -5.0);
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 'c':
      case 'C':
         showPoints = !showPoints;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
```

продолжение ⇩

Листинг 12.5 (*продолжение*)

```
    glutCreateWindow(argv[0]);
    init();
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Управление NURBS-объектом

Как видно из листинга 12.5, команда `gluNewNurbsRenderer()` возвращает новый NURBS-объект, с типом указателя на структуру `GLUnurbsObj`. Вы должны создать этот объект перед использованием любой другой процедуры NURBS. Когда NURBS-объект станет вам не нужен, удалите его командой `gluDeleteNurbsRenderer()` для освобождения занимаемой им памяти.

GLUnurbsObj* `gluNewNurbsRenderer`(void);
Создает NURBS-объект, *nobj*, и возвращает указатель на него или нуль в случае, если OpenGL оказывается не в силах выделить память объекту.

void `gluDeleteNurbsRenderer`(GLUnurbsObj **nobj*);
Удаляет NURBS-объект *nobj*.

Управление свойствами NURBS-объекта

Управление тем, как будет строиться и отображаться NURBS-объект, производится через набор свойств, связанных с NURBS-объектом. Эти свойства отвечают за способ растеризации поверхности (например, закрашивание или каркас), за отображение или возврат вершин мозаичного разбиения, за детализацию этого разбиения.

void `gluNurbsProperty`(GLUnurbsObj **nobj*, GLenum *property*, GLfloat *value*);
Управляет атрибутами NURBS-объекта, *nobj*. Аргумент *property* указывает одно из следующих свойств: `GLU_DISPLAY_MODE`, `GLU_NURBS_MODE`, `GLU_CULLING`, `GLU_SAMPLING_METHOD`, `GLU_SAMPLING_TOLERANCE`, `GLU_PARAMETRIC_TOLERANCE`, `GLU_U_STEP`, `GLU_V_STEP` или `GLU_AUTO_LOAD_MATRIX`. Аргумент *value* задает конкретное значение для свойства.

Для свойства `GLU_DISPLAY_MODE` значение по умолчанию — `GLU_FILL`, при этом поверхность воспроизводится многоугольниками. Если совместно с ним используется свойство `GLU_OUTLINE_POLYGON`, создаются только контуры многоугольников. Атрибут `GLU_OUTLINE_PATCH` предписывает применение контуров для лоскутов и кривых усечения. (См. раздел «Создание NURBS-кривой или поверхности».)

Свойство `GLU_NURBS_MODE` отвечает за то, будут ли вершины мозаики из многоугольников просто отображены (*value* равно `GLU_NURBS_RENDERER`, это значение по умолчанию) или задействован интерфейс обратной связи (если *value* — `GL_NURBS_TESSELLATOR`). За подробностями обращайтесь к разделу «Получение примитивов с выхода тесселятора NURBS».

Константа GLU_CULLING может ускорить процесс, так как в этом случае тесселяция не выполняется для NURBS-объектов, целиком расположенными вне видимого объема; установите это свойство в GL_TRUE для включения режима отбраковки (culling) (значение по умолчанию — GL_FALSE).

Поскольку NURBS-объект воспроизводится как примитив с различными значениями параметров (u и v), он «разламывается» на небольшие отрезки прямых или многоугольники. Если значение аргумента *property* равно GLU_SAMPLING_METHOD, значение *value* устанавливается в GLU_PATH_LENGTH (по умолчанию), GLU_PARAMETRIC_ERROR, GLU_DOMAIN_DISTANCE, GLU_OBJECT_PATH_LENGTH или GLU_OBJECT_PARAMETRIC_ERROR. Эти константы задают характер мозаичного разбиения NURBS-кривой или поверхности. Когда *value* равно GLU_PATH_LENGTH, поверхность строится так, чтобы максимальная длина ребра, в пикселах, выходных многоугольников не превышала заданную значением GLU_SAMPLING_TOLERANCE. Если *value* равно GLU_PARAMETRIC_ERROR, значением GLU_PARAMETRIC_TOLERANCE является максимальное расстояние в пикселах между выходными многоугольниками и аппроксимируемой ими поверхностью.

Значение GLU_OBJECT_PATH_LENGTH для аргумента *value* схоже с GLU_PATH_LENGTH, с той разницей, что максимальная длина (величина GLU_SAMPLING_TOLERANCE) примитивов тесселяции измеряется относительно пространства координат объекта, но не в пикселах. Аналогичным образом соотносятся GLU_OBJECT_PARAMETRIC_ERROR и GLU_PARAMETRIC_ERROR, здесь GLU_OBJECT_PARAMETRIC_TOLERANCE определяет предельное расстояние между примитивами тесселяции и поверхностью NURBS-объекта в системе координат последнего.

Если значение *value* задано как GLU_DOMAIN_DISTANCE, приложение задает, в параметрических координатах, количество точек на единицу длины в u- и v-измерениях — GLU_U_STEP и GLU_V_STEP соответственно.

Если задано свойство (*property*) GLU_SAMPLING_TOLERANCE и метод выборки (sampling) GLU_PATH_LENGTH или GLU_OBJECT_PATH_LENGTH, значение *value* соответствует максимальной длине, в пикселах или в единицах измерения объекта (согласно свойству), используемой для выходных многоугольников. Например, значение по умолчанию 50.0 ограничивает длину отрезка или ребра многоугольника 50 пикселами или 50.0 единицами измерения в собственной системе отсчета объекта. Если значение *property* равно GLU_PARAMETRIC_TOLERANCE и метод выборки — GLU_PARAMETRIC_ERROR или GLU_OBJECT_PARAMETRIC_ERROR, величина *value* управляет максимальным расстоянием, в пикселах или в системе координат объекта соответственно, между мозаичными многоугольниками и аппроксимируемой ими поверхностью. Вне зависимости от метода выборки, значение по умолчанию для GLU_PARAMETRIC_TOLERANCE равно 0.5. Для GLU_PARAMETRIC_ERROR это значение создает мозаичные многоугольники в пределах половины пиксела аппроксимируемой поверхности. (Для GLU_OBJECT_PARAMETRIC_ERROR действие значения по умолчанию 0.5 не настолько очевидно.)

Если метод выборки задан как GLU_DOMAIN_DISTANCE и для *property* указано одно из двух значений: GLU_U_STEP или GLU_V_STEP, *value* является количеством выбираемых точек на единицу длины в u- или v-измерениях соответственно, в параметрических координатах. Значение по умолчанию для параметров GLU_U_STEP и GLU_V_STEP равно 100.

Свойство GLU_AUTO_LOAD_MATRIX определяет, загружаются ли матрицы проекции, модельно-видовая и окна просмотра с сервера OpenGL (по умолчанию GL_TRUE), или приложение должно само задать их с помощью gluLoadSamplingMatrices() (GL_FALSE).

ПРИМЕЧАНИЕ

Несколько свойств NURBS (GLU_NURBS_MODE и его значение GLU_NURBS_TESSELLATOR (не по умолчанию), GLU_OBJECT_PATH_LENGTH и GLU_OBJECT_PARAMETRIC_ERROR) были представлены в версии GLU 1.3. До GLU 1.3 способ их реализации зависел от конкретного расширения, если они поддерживались. Посмотрите на номер версии GLU перед тем, как попытаетесь использовать эти свойства.

void gluLoadSamplingMatrices(GLUnurbsObj **nobj*, const GLfloat *modelMatrix*[16], const GLfloat *projMatrix*[16], const GLint *viewport*[4]);

При включенном параметре GLU_AUTO_LOAD_MATRIX модельно-видовая матрица и матрица проекции, а также окно просмотра указываются gluLoadSamplingMatrices() и используются для вычисления матриц выбора и отбраковки, применяемых к каждой кривой или поверхности NURBS.

Если вам нужно запросить текущее значение свойства NURBS, используйте gluGetNurbsProperty().

void gluGetNurbsProperty(GLUnurbsObj **nobj*, GLenum *property*, GLfloat **value*);

Запрашивает значение свойства *property* NURBS-объекта *nobj*, возвращая его по ссылке на *value*.

Обработка ошибок NURBS

Поскольку при выполнении функций NURBS вероятно возникновение 37 различных ошибок, неплохо зарегистрировать функцию обратной связи на случай их проявления при выполнении кода. В программе из листинга 12.5 это делается вызовом

```
gluNurbsCallback(theNurb, GLU_ERROR, (GLvoid (*)()) nurbsError);
```

void gluNurbsCallback(GLUnurbsObj **nobj*, GLenum *which*,
void (**fn*)(GLenum *errorCode*));

Аргумент *which* — это тип обратной связи; для включения обработки ошибок он должен быть равен GLU_ERROR. Когда функция NURBS обнаруживает ошибку, вызывается функция *fn*, которой передается код ошибки в качестве единственного аргумента. Значение для *errorCode* — это одна из 37 ошибок, именованных от GLU_NURBS_ERROR1 до GLU_NURBS_ERROR37. Используйте gluErrorString() для получения содержательного описания кода ошибки.

В листинге 12.5 для обработки всех возможных ошибок зарегистрирована процедура nurbsError():

```
void CALLBACK nurbsError(GLenum errorCode)
{
const GLubyte *estring;
estring = gluErrorString(errorCode);
fprintf(stderr, "Ошибка NURBS: %s\n", estring);
exit(0);
}
```

В версию GLU 1.3 включены дополнительные функции обратной связи NURBS, возвращающие программе значения, полученные при мозаичном разбиении (вместо воспроизведения этих значений). За подробной информацией о новых вызовах обращайтесь к разделу «Получение примитивов с выхода тесселятора NURBS».

Создание NURBS-кривой или поверхности

При построении NURBS-поверхности функция `gluNurbsSurface()` вызывается между командами-скобками `gluBeginSurface()` и `gluEndSurface()`. Обрамляющие подпрограммы сохраняют и восстанавливают состояние вычислителя.

void `gluBeginSurface`(GLUnurbsObj **nobj*);
void `gluEndSurface`(GLUnurbsObj **nobj*);

После вызова `gluBeginSurface()` делается один или более вызовов `gluNurbsSurface()`, описывающих атрибуты поверхности. Среди них должны присутствовать команда с указанием типа поверхности (`GL_MAP2_VERTEX_3`) и команда для генерации вершин (`GL_MAP2_VERTEX_4`). Команда `gluEndSurface()` завершает описание поверхности. Усечение NURBS-поверхностей также производится в пределах между `gluBeginSurface()` и `gluEndSurface()`. (См. раздел «Усечение NURBS-поверхностей».)

void `gluNurbsSurface`(GLUnurbsObj **nobj*, GLint *uknot_count*, GLfloat **uknot*, GLint *vknot_count*, GLfloat **vknot*, GLint *u_stride*, GLint *v_stride*, GLfloat **ctlarray*, GLint *uorder*, GLint *vorder*, GLenum *type*);

Описывает вершины (или нормали к поверхности или координаты текстуры) NURBS-поверхности *nobj*. Некоторые значения должны быть заданы для обоих параметрических направлений: *u* и *v*, такие как последовательности узлов (*uknot* и *vknot*), количество узлов (*uknot_count* и *vknot_count*) и порядок многочлена (*uorder* и *vorder*). Заметьте, что количество контрольных точек в целом не указывается. Оно выводится для каждого параметра как число узлов минус порядок. Соответственно, для полной поверхности число контрольных точек равно произведению числа контрольных точек в каждом параметрическом направлении друг на друга. Аргумент *ctlarray* указывает на этот массив контрольных точек.

Последний аргумент, *type*, — это один из типов двухмерных вычислителей. В общем случае вы можете использовать `GL_MAP2_VERTEX_3` для рациональных и `GL_MAP2_VERTEX_4` для нерациональных контрольных точек. В вашем распоряжении и другие типы, такие как `GL_MAP2_TEXTURE_COORD_*` или `GL_MAP2_NORMAL`, для вычисления и назначения координат текстур или нормалей к поверхности. Например, чтобы создать освещенную (то есть с нормалями) и текстурированную NURBS-поверхность, вам нужно сделать следующую последовательность вызовов:

```
gluBeginSurface(nobj);
gluNurbsSurface(nobj,..., GL_MAP2_TEXTURE_COORD_2);
gluNurbsSurface(nobj,..., GL_MAP2_NORMAL);
gluNurbsSurface(nobj,..., GL_MAP2_VERTEX_3);
gluEndSurface(nobj);
```

Аргументы *u_stride* и *v_stride* представляют количество значений с плавающей точкой между контрольными точками в каждом из параметрических направлений. В отношении этих параметров эффективен тип вычислителя, а также его порядок. В программе из листинга 12.5 значение *u_stride* равно 12 (4 × 3), поскольку имеется три координаты для каждой вершины (заданные `GL_MAP2_VERTEX_3`) и четыре контрольные точки для параметра *v*; *v_stride* равно 3, так как каждая вершина описывается тремя координатами, а контрольные точки для параметра *v* в памяти являются смежными.

Рисование NURBS-кривой аналогично рисованию поверхности, с той разницей, что все вычисления относятся к одному параметру — *u*, а не к двум. Как и для поверхностей, используются подпрограммы-скобки `gluBeginCurve()` и `gluEndCurve()`.

void `gluBeginCurve`(GLUnurbsObj **nobj*);

void `gluEndCurve`(GLUnurbsObj **nobj*);

После вызова `gluBeginCurve()` делается один или более вызовов `gluNurbsCurve()`, определяющих атрибуты поверхности. Они должны включать в себя указание типа поверхности (`GL_MAP1_VERTEX_3`) или генерацию вершин (`GL_MAP1_VERTEX_4`). Описание завершается командой `gluEndCurve()`.

void `gluNurbsCurve`(GLUnurbsObj **nobj*, GLint *uknot_count*, GLfloat **uknot*, GLint *u_stride*, GLfloat **ctlarray*, GLint *uorder*, GLenum *type*);

Определяет NURBS-кривую для объекта *nobj*. Аргументы задаются аналогично `gluNurbsSurface()`. Заметьте, что здесь процедура требует указания только одной последовательности узлов и объявления одного NURBS-объекта. Если кривая определена в пределах пары `gluBeginCurve()/gluEndCurve()`, тип вычислителя может быть одним из допустимых для одномерных вычислителей (таким как `GL_MAP1_VERTEX_3` или `GL_MAP1_VERTEX_4`).

Получение примитивов с выхода тесселятора NURBS

По умолчанию тесселятор NURBS разбивает NURBS-объект на линии и многоугольники, а затем их отображает. В версии GLU 1.3 появились новые функции обратной связи, которые позволяют вместо отображения вернуть рассчитанные значения в программу.

Чтобы воспользоваться этой возможностью, вызовите `gluNurbsProperty()`, установив значение `GLU_NURBS_MODE` для свойства *property* и для *value* — `GLU_NURBS_TESSELLATOR`. Второй шаг: несколько вызовов `gluNurbsCallback()` для регистрации функций обратной связи.

void `gluNurbsCallback`(GLUnurbsObj **nobj*, GLenum *which*, void (**fn*)());

Аргумент *nobj* — это мозаично разбиваемый NURBS-объект. Аргумент *which* представляет именованный идентификатор функции обратной связи. Если значение свойства *(property)* `GLU_NURBS_MODE` установлено в `GLU_NURBS_TESSELLATOR`, допустимы 12 таких функций (в дополнение к `GLU_ERROR`). (Если свойство `GLU_NURBS_MODE` установлено в значение по умолчанию `GLU_NURBS_RENDERER`, допустим только режим `GLU_ERROR`.) Двенадцать функций обратной связи имеют следующие прототипы:

GLU_NURBS_BEGIN	void begin(GLenum *type*)
GLU_NURBS_BEGIN_DATA	void begin(GLenum *type*, void **userData*)
GLU_NURBS_TEXTURE_COORD	void texCoord(GLfloat **tCrd*)
GLU_NURBS_TEXTURE_COORD_DATA	void texCoord(GLfloat **tCrd*, void **userData*)
GLU_NURBS_COLOR	void color(GLfloat **color*)
GLU_NURBS_COLOR_DATA	void color(GLfloat **color*, void **userData*)
GLU_NURBS_NORMAL	void normal(GLfloat **nml*)
GLU_NURBS_NORMAL_DATA	void normal(GLfloat **nml*, void **userData*)
GLU_NURBS_VERTEX	void vertex(GLfloat **vertex*)
GLU_NURBS_VERTEX_DATA	void vertex(GLfloat **vertex*, void **userData*)
GLU_NURBS_END	void end(void)
GLU_NURBS_END_DATA	void end(void **userData*)

Чтобы изменить функцию обратной связи, вызовите `gluNurbsCallback()` с указанием другой процедуры. Чтобы удалить функцию обратной связи, передайте в `gluNurbsCallback()` нулевой указатель для соответствующей функции.

Шесть функций обратной связи позволяют передавать в себя некоторые другие данные. Для задания пользовательских данных вам нужно сделать вызов `gluNurbsCallbackData()`.

void `gluNurbsCallbackData`(GLUnurbsObj **nobj*, void **userData*);
Аргумент *nobj* — это мозаично разбиваемый NURBS-объект. Структура *userData* представляет данные, передаваемые в функцию обратной связи.

Во время процесса тесселяции процедуры обратного вызова вызываются аналогично командам OpenGL `glBegin()`, `glTexCoord*()`, `glColor*()`, `glNormal*()`, `glVertex*()` и `glEnd()`. В режиме `GLU_NURBS_TESSELLATOR` кривая или поверхность не отображаются сразу, а их данные вершин перехватываются и передаются в аргументах функций обратной связи.

Демонстрируют применение функций обратной связи листинги 12.6 и 12.7, являющиеся кусками программы `surfpoints.c`, представляющей собой модифицированный вариант программы `surface.c`. Листинг 12.6 включает в себя инициализацию NURBS-объекта — процедуру `init()`. Для подготовки нужного режима устанавливается также параметр `GLU_NURBS_MODE`, и регистрируются функции обратной связи.

Листинг 12.6. Регистрация функций обратной связи NURBS-тесселятора: surfpoints.c

```
void init(void)
{
/* Это только часть процедуры init() с исключением нескольких строк */
   theNurb = gluNewNurbsRenderer();
   gluNurbsProperty(theNurb, GLU_NURBS_MODE, GLU_NURBS_TESSELLATOR);
   gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 100.0);
   gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
   gluNurbsCallback(theNurb, GLU_ERROR, nurbsError);
   gluNurbsCallback(theNurb, GLU_NURBS_BEGIN, beginCallback);   продолжение ⇩
```

Листинг 12.6 (*продолжение*)

```
    gluNurbsCallback(theNurb, GLU_NURBS_VERTEX, vertexCallback);
    gluNurbsCallback(theNurb, GLU_NURBS_NORMAL, normalCallback);
    gluNurbsCallback(theNurb, GLU_NURBS_END, endCallback);
}
```

В листинге 12.7 показано применение функций обратной связи. Здесь предложения `printf()` используются для вывода диагностических сообщений соответственно процессу построения объекта и возвращаемым данным для вершин. Кроме того, возвращаемые данные мозаичного разбиения повторно пересылаются в конвейер с целью выполнения обычного отображения.

Листинг 12.7. Применение функций обратной связи NURBS-тесселятора: surfpoints.c

```
void CALLBACK beginCallback(GLenum whichType)
{
   glBegin(whichType); /* директива повторной передачи на отображение */
   printf("glBegin(");
   switch (whichType) { /* вывод диагностических сообщений */
      case GL_LINES:
         printf("GL_LINES)\n"); break;
      case GL_LINE_LOOP:
         printf("GL_LINE_LOOP)\n"); break;
      case GL_LINE_STRIP:
         printf("GL_LINE_STRIP)\n"); break;
      case GL_TRIANGLES:
         printf("GL_TRIANGLES)\n"); break;
      case GL_TRIANGLE_STRIP:
         printf("GL_TRIANGLE_STRIP)\n"); break;
      case GL_TRIANGLE_FAN:
         printf("GL_TRIANGLE_FAN)\n"); break;
      case GL_QUADS:
         printf("GL_QUADS)\n"); break;
      case GL_QUAD_STRIP:
         printf("GL_QUAD_STRIP)\n"); break;
      case GL_POLYGON:
         printf("GL_POLYGON)\n"); break;
      default:
         break;
   }
}
void CALLBACK endCallback()
{
   glEnd(); /* директива повторной передачи на отображение */
   printf("glEnd()\n");
}

void CALLBACK vertexCallback(GLfloat *vertex)
{
   glVertex3fv(vertex); /* повторная передача вершин */
   printf("glVertex3f (%5.3f, %5.3f, %5.3f)\n",
   vertex[0], vertex[1], vertex[2]);
}
```

```
void CALLBACK normalCallback(GLfloat *normal)
{
   glNormal3fv(normal); /* повторная передача нормалей */
   printf("glNormal3f (%5.3f, %5.3f, %5.3f) ", normal[0], normal[1],
normal[2]);
}
```

Усечение NURBS-поверхностей

Для создания усеченных NURBS-поверхностей с помощью OpenGL начните с создания «сырой» поверхности. Затем, после вызовов `gluBeginSurface()` и `gluNurbsSurface()`, но перед `gluEndSurface()`, выполните усечение `gluBeginTrim()`.

void `gluBeginTrim`(GLUnurbsObj **nobj*);
void `gluEndTrim`(GLUnurbsObj **nobj*);
Отмечают начало и конец контура усечения. Он представляет собой набор ориентированных вырезанных из кривой отрезков (образующих замкнутую кривую), определяющих границы NURBS-поверхности.

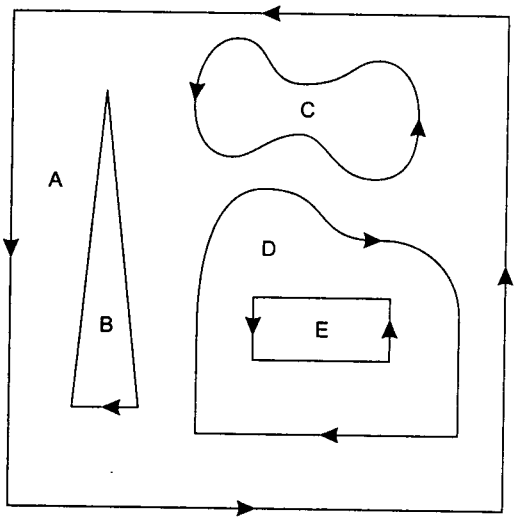
Вы можете создать усеченные кривые двух видов: кусочно-линейные с помощью `gluPwlCurve()` и NURBS-кривые с помощью `gluNurbsCurve()`. Кусочно-линейная кривая не похожа на кривую в прямом смысле этого слова, поскольку это совокупность отрезков прямых линий. NURBS-кривая для усечения должна лежать в пределах единичного квадрата параметрического (u, v) пространства. Тип NURBS-кривой для усечения — обычно `GLU_MAP1_TRIM2`. Тип `GLU_MAP1_TRIM3` применяется реже, здесь кривая описывается в двухмерном однородном пространстве (u', v', w') при (u, v) = (u'/w', v'/w').

void `gluPwlCurve`(GLUnurbsObj **nobj*, GLint *count*, GLfloat **array*, GLint *stride*, GLenum *type*);
Описывает кусочно-линейную кривую усечения для NURBS-объекта *nobj*. Аргумент *count* — это число точек кривой, содержащихся в массиве *array*. Аргумент *type* равен либо `GLU_MAP1_TRIM_2` (обычно), либо `GLU_MAP1_TRIM_3` (однородное параметрическое пространство (u, v, w)). Значение типа *type* обусловливает значение аргумента *stride* — 2 или 3. Шаг по индексу *stride* представляет количество значений с плавающей запятой до следующей вершины массива *array*.

Вы должны озаботиться ориентацией кривых усечения — то есть тем, как они обходятся: по часовой стрелке или наоборот, — чтобы быть уверенными, что вы включили в обработку нужную часть поверхности. Вы уже должны знать, что при обводе все, что находится слева от кривой, включается в рассмотрение, а все, что находится справа, — отбрасывается. Например, если ваш объект состоит из единственного контура, обходимого против часовой стрелки, к учету принимаются все его «внутренности». Если усечение выполняется для двух непересекающихся контуров, также ориентированных против часовой стрелки, к рассмотрению принимаются внутренние области обоих контуров. Если вы имеете один «против ча-

совой стрелки» контур и внутри него два контура «по часовой стрелке», в область усечения войдут два отверстия. Наиболее внешняя кривая усечения должна обходиться против часовой стрелки. Часто кривая усечения окружает единичный квадрат, предполагая обработку его внутренней области. Это также случай по умолчанию, когда вы самостоятельно не указываете никаких кривых для усечения.

Кривые усечения должны быть замкнутыми и непересекающимися. Вы можете комбинировать их, пока их конечные точки не сформируют замкнутую кривую. Вы можете вкладывать кривые, создавая «острова», «плавающие» в пространстве. Убедитесь, что ориентация кривых задана верно. Так, вы получите противоречие и, следовательно, ошибку, если укажете в качестве области усечения две кривые, обходимые против часовой стрелки, где одна вложена в другую. Рисунок 12.5 иллюстрирует несколько допустимых вариантов.




```
gluBeginSurface();
gluNurbsSurface(...);
gluBeginTrim();
gluPwlCurve(...); /* A */
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...); /* B */
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...); */ C */
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...); */ D */
gluPwlCurve(...); /* D */
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...); /* E */
gluEndTrim();
gluEndSurface();
```

Рис. 12.5. Параметрические кривые усечения

На рис. 12.6 показана та же небольшая возвышенность с рис. 12.4, но с кривой усечения, скомбинированной из кусочно-линейной кривой и NURBS-кривой. Программа, создающая эти фигуры, подобна представленной в листинге 12.5; разница заключается в процедурах, включенных в листинг 12.8.

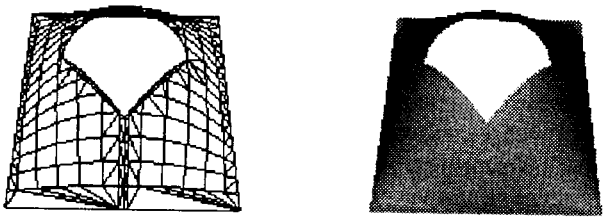


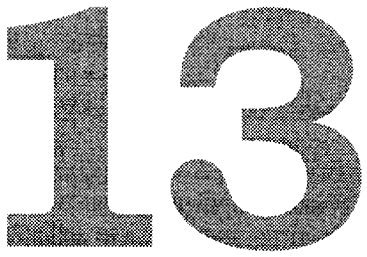
Рис. 12.6. Усеченная NURBS-поверхность

Листинг 12.8. Усечение NURBS-поверхности: trim.c

```
void display(void)
{
   GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   GLfloat edgePt[5][2] = /* против часовой стрелки */
                 {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
                 {0.0, 0.0}};
   GLfloat curvePt[4][2] = /* по часовой стрелке */
                 {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
   GLfloat curveKnots[8] =
                 {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   GLfloat pwlPt[3][2] = /* по часовой стрелке */
                 {{0.75, 0.5}, {0.5, 0.25}, {0.25, 0.5}};

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glRotatef(330.0, 1., 0., 0.);
   glScalef(0.5, 0.5, 0.5);

   gluBeginSurface(theNurb);
   gluNurbsSurface(theNurb, 8, knots, 8, knots,
      4 * 3, 3, &ctlpoints[0][0][0], 4, 4, GL_MAP2_VERTEX_3);
   gluBeginTrim(theNurb);
   gluPwlCurve(theNurb, 5, &edgePt[0][0], 2, GLU_MAP1_TRIM_2);
   gluEndTrim(theNurb);
   gluBeginTrim(theNurb);
      gluNurbsCurve(theNurb, 8, curveKnots, 2, &curvePt[0][0], 4,
           GLU_MAP1_TRIM_2);
      gluPwlCurve(theNurb, 3, &pwlPt[0][0], 2, GLU_MAP1_TRIM_2);
   gluEndTrim(theNurb);
   gluEndSurface(theNurb);

   glPopMatrix();
   glFlush();
}
```

В листинге 12.8 команды-скобки `gluBeginTrim()` и `gluEndTrim()` обрамляют каждую кривую усечения. Первая, с вершинами в массиве `edgePt[][]`, ориентирована против часовой стрелки вокруг единичного квадрата параметрического пространства. Это гарантирует отображение всего, что не исключается из рассмотрения вложенными кривыми усечения, обходимыми по часовой стрелке. Вторая кривая — это комбинация кусочно-линейной и NURBS-кривой. Концы NURBS-кривой в точках (0.9, 0.5) и (0.1, 0.5) замыкаются с помощью кусочно-линейной кривой.

Выбор и обратная связь



После прочтения этой главы вы научитесь следующему:

- Создавать приложения, позволяющие пользователю выбирать область экрана или нарисованный на экране объект
- Использовать режим обратной связи OpenGL для получения результатов построения сцены

Некоторые графические программы просто рисуют статические изображения двух- или трехмерных объектов. Другие приложения позволяют пользователю идентифицировать объект на экране и затем перемещать, модифицировать, удалять его или манипулировать этим объектом другими способами. Библиотека OpenGL ориентирована на поддержку именно таких интерактивных приложений. Поскольку обычно нарисованный на экране объект подвергается вращению, смещению и перспективным преобразованиям, это может усложнить задачу его идентификации пользователем на фоне трехмерной сцены. В помощь вам OpenGL предоставляет механизм выбора, автоматически информирующий вас, какие объекты отображены внутри указанной области окна. Вы вправе использовать этот механизм вместе со специальными процедурами-инструментами для выяснения того, какой объект в пределах области указан пользователем — *выделен* (picking) им с помощью указателя мыши.

Выбор фактически является режимом выполнения операций OpenGL; обратная связь — другой подобный режим. В режиме обратной связи вы используете свое аппаратное обеспечение и OpenGL для выполнения обыкновенных построения и отображения сцены. Вместо использования вычисленных результатов для рисования изображения на экране в этом случае OpenGL возвращает (feeds back) эту информацию вам. Например, если вы хотите нарисовать трехмерный объект с помощью графопостроителя, а не на экране, вам следует отобразить его элементы в режиме обратной связи, собрать вместе инструкции отображения и затем преобразовать их в команды, понятные графопостроителю.

Как при выборе, так и в режиме обратной связи информация об изображении возвращается приложению, а не посылается в буфер кадров, как это происходит при воспроизведении. Таким образом, экран остается «замороженным» — никакого рисования не происходит, — пока OpenGL не покинет режим выбора или обратной связи. В обоих режимах содержимое буферов цвета, глубины, трафарета и накопления не подвергается воздействию. В этой главе каждому из означенных режимов посвящен собственный раздел:

- **Выбор.** Рассказывает, как использовать режим выбора и связанные с ним подпрограммы, чтобы позволить пользователю вашего приложения выбирать объект на экране.
- **Обратная связь.** Здесь описывается, как получить данные, которые могут быть отображены на экране, и как обрабатывать эту информацию.

Выбор

Обычно, когда вы планируете использовать механизм выбора OpenGL, сначала вы рисуете свою сцену в буфер кадров и затем входите в режим выбора и перерисовываете сцену. Войдя в режим выбора, вы уже не можете изменить содержимое буфера кадров, пока не покинете этот режим. После выхода из него OpenGL возвращает список примитивов, попадающих в видимый объем (помните, что видимый объем определяется текущими модельно-видовой матрицей и матрицей проекции и любыми дополнительными плоскостями отсечения, как обсуждалось

в главе 3). Каждый примитив, пересекающий видимый объем, утвердительно реагирует на выбор самого себя (selection hit). Список примитивов фактически возвращается как массив целочисленных *идентификаторов* (names) — имен и связанных с ними данных — записей реакции системы (hit records) соответствующих текущему содержанию *стека имен* (name stack). Вы выстраиваете стек имен загрузкой имен во время применения команды отображения примитивов, находясь в режиме выбора. Таким образом, когда возвращается список имен, вы можете использовать его для определения, какой из примитивов может быть выбран на экране пользователем.

В дополнение к механизму выбора OpenGL предоставляет подпрограммы-инструменты, служащие для упрощения выбора в некоторых случаях, ограничивая рисование небольшой областью окна просмотра. Обычно вы используете такую подпрограмму, чтобы выяснить, какой объект нарисован вблизи указателя мыши, тем самым вы можете идентифицировать выбранный пользователем объект. (Вы можете также разграничить область выбора, указав дополнительные плоскости отсечения. Помните, что эти плоскости действуют в отношении мировых координат, но не в системе координат экрана.) Поскольку выбор с помощью указателя (picking) является частным случаем выбора (selection), сначала в этой главе мы рассмотрим выбор в принципе, и лишь затем — выделение указателем мыши.

Основные шаги

Чтобы задействовать механизм выбора, вы должны выполнить следующие шаги.

1. Указать массив, который будет использоваться для возвращаемых записей реакции, с помощью `glSelectBuffer()`.
2. Войти в режим выбора посредством задания параметра `GL_SELECT` для команды `glRenderMode()`.
3. Инициализировать стек имен командами `glInitNames()` и `glPushName()`.
4. Определить видимый объем, в котором будет возможен выбор. Обычно он не отличается от видимого объема вашей оригинальной нарисованной сцены, поэтому вероятно, что вы захотите сохранить и после восстановить текущее состояние преобразования — с помощью `glPushMatrix()` и `glPopMatrix()`.
5. Поочередно вызывать команды отображения примитивов и команды манипулирования стеком имен с целью присвоить каждому заслуживающему интерес примитиву соответствующее имя.
6. Выйти из режима выбора и обработать возвращенные механизмом выбора записи реакции.

void `glSelectBuffer`(GLsizei *size*, GLuint **buffer*);

Задает массив, используемый для возвращаемых при выборе данных. Аргумент *buffer* — это указатель на массив беззнаковых целых, куда помещаются эти данные, а аргумент *size* указывает максимальное количество значений, которые могут быть сохранены в массиве. Вам следует сделать вызов `glSelectBuffer()` перед входом в режим выбора.

GLint `glRenderMode`(GLenum *mode*);

Управляет тем, в каком режиме приложение будет выполнять визуализацию — в режиме выбора или обратной связи. Аргумент режима *mode* задается равным `GL_RENDER` (по умолчанию), `GL_SELECT` или `GL_FEEDBACK`. Приложение остается в указанном режиме до того момента, пока снова не будет выдана команда `glRenderMode()` с другим аргументом режима. Перед входом в режим выбора нужно вызвать команду `glSelectBuffer()` для задания массива выбора. Аналогично, перед активизацией режима обратной связи следует вызвать `glFeedbackBuffer()` для задания массива данных обратной связи. Возвращаемое значение `glRenderMode()` для текущего режима визуализации (то есть не значения аргумента *mode*): `GL_SELECT` или `GL_FEEDBACK` представляет собой количество зарегистрированных попаданий или количество значений, помещенных в массив обратной связи; отрицательное число означает выход за границы (переполнение) массива. Вы можете использовать параметр `GL_RENDER_MODE` совместно с `glGetIntegerv()` для получения идентификатора текущего режима.

В следующем разделе описаны команды для манипулирования стеком имен.

Создание стека имен

Как упоминалось в предыдущем разделе, стек имен закладывает основу для возвращаемой вам информации выбора. Чтобы создать стек имен, сначала инициализируйте его командой `glInitNames()`, которая просто очищает стек, и затем добавьте целочисленные идентификаторы-имена во время выполнения соответствующих команд отображения. Как и следовало ожидать, команды, манипулирующие *стеком*, позволяют вам затолкнуть имя в него (`glPushName()`), вытолкнуть из него имя (`glPopName()`) и заменить имя на вершине стека другим (`glLoadName()`). Листинг 13.1 демонстрирует, как программа может манипулировать стеком с помощью этих команд.

Листинг 13.1. Создание стека имен

```
glInitNames();
glPushName(0);
glPushMatrix(); /* сохранение текущего состояния преобразования */

/* здесь создается желаемый для вас видимый объем */

   glLoadName(1);
   drawSomeObject();
   glLoadName(2);
   drawAnotherObject();
   glLoadName(3);
   drawYetAnotherObject();
   drawJustOneMoreObject();

glPopMatrix(); /* восстановление предшествующего состояния преобразования */
```

В этом примере первые два рисуемых объекта имеют собственные имена, а третий и четвертый объекты совместно используют одно имя. При таких условиях выбор третьего или четвертого объекта повлечет возврат только одной записи реакции. Вы вправе иметь множество объектов, разделяющих одно и то же имя, если различение объектов между собой не обязательно.

void `glInitNames`(void);
Очищает стек имен, который в результате становится пустым.

void `glPushName`(GLuint *name*);
Заталкивает имя *name* в стек имен. Если при помещении имени в стек последний переполняется, генерируется ошибка `GL_STACK_OVERFLOW`. Глубина стека имен может варьироваться в зависимости от реализации OpenGL, но в него должны умещаться по меньшей мере 64 имени. Вы можете использовать параметр `GL_NAME_STACK_DEPTH` совместно с `glGetIntegerv()` для выяснения глубины вашего стека имен.

void `glPopName`(void);
Выталкивает одно имя из вершины стека имен. При выталкивании из пустого стека генерируется ошибка `GL_STACK_UNDERFLOW`.

void `glLoadName`(GLuint *name*);
Замещает значение на вершине стека имен именем *name*. Если стек в исходном положении пуст, как это должно быть после вызова `glInitNames()`, команда `glLoadName()` генерирует ошибку `GL_INVALID_OPERATION`. Во избежание этого хотя бы раз вызовите `glPushName()`, чтобы положить что-нибудь на вершину стека, перед вызовом `glLoadName()`.

Вызовы `glPushName()`, `glPopName()` и `glLoadName()` игнорируются, если вы не переключились в режим выбора. Вы заметите, что это упрощает кодирование, так как можно использовать один и тот же код, выполняющий отображение, в двух режимах — выбора и при обычной визуализации.

Буфер выбора

В режиме выбора примитивы, попадающие в видимый объем, вызывают запись реакции при выделении. Каждый раз при выполнении команды манипуляции стеком имен или вызове `glRenderMode()` OpenGL создает запись нажатия в массив таких записей. По ходу этого процесса объекты с одинаковым именем — например, скомпонованные из более чем одного примитива — не вызывают повторную генерацию записи. Также гарантируется, что при вызове `glRenderMode()` избыточные записи выделений не попадут в возвращаемый массив.

ПРИМЕЧАНИЕ

В дополнение к геометрическим примитивам допустимые координаты позиции растра, вырабатываемые вызовами glRasterPos() или glWindowPos(), тоже могут оказаться причиной выбора нажатия. Также, если при активном режиме выбора по ходу отображения отбраковываются (culling) многоугольники, это не приводит к формированию записи реакции.

Каждая запись нажатия состоит из следующих четырех элементов в указанном порядке:

- количество имен в стеке имен на момент выбора;
- максимальные и минимальные оконные координаты *z* для всех вершин всех примитивов, попадающих в видимый объем на момент последнего выбора. Эти два значения, которые лежат в диапазоне [0, 1], умножаются на $2^{32} - 1$ и округляются к ближайшему беззнаковому целому;
- содержимое стека имен во время ответа на выбор, где первым следует самый нижний элемент.

Когда вы входите в режим выбора, OpenGL инициализирует указатель на начало массива записей реакции. Каждый раз при включении в массив новой записи указатель соответствующим образом обновляется. Если при этом происходит выход за границы массива (превышение значения аргумента *size* команды `glSelectBuffer()`), OpenGL записывает столько данных, сколько вмещает массив, и устанавливает признак переполнения. Когда вы покидаете режим выбора с помощью `glRenderMode()`, эта команда возвращает количество принятых к рассмотрению записей реакции (включая неполную запись при ее наличии), очищает стек имен, сбрасывает флаг переполнения и указатель стека имен. Если был установлен флаг переполнения, возвращаемое значение равно –1.

Пример выбора

В программе из листинга 13.2 на экране рисуются четыре треугольника (зеленый, красный и два желтых, созданных с помощью `drawTriangle()`) и каркасная рамка, представляющая собой видимый объем (`drawViewVolume()`). Затем происходит повторное построение треугольников (`selectObjects()`), но уже при активном режиме выбора. Соответствующие записи реакции обрабатываются в процедуре `processHits()`, а массив записей распечатывается. Первый треугольник генерирует запись, второй — нет, а третий и четвертый генерируют одну запись на двоих.

Листинг 13.2. Пример выбора: select.c

```
void drawTriangle(GLfloat x1, GLfloat y1, GLfloat x2,
GLfloat y2, GLfloat x3, GLfloat y3, GLfloat z)
{
   glBegin(GL_TRIANGLES);
   glVertex3f(x1, y1, z);
   glVertex3f(x2, y2, z);
   glVertex3f(x3, y3, z);
   glEnd();
}

void drawViewVolume(GLfloat x1, GLfloat x2, GLfloat y1,
                    GLfloat y2, GLfloat z1, GLfloat z2)
{
   glColor3f(1.0, 1.0, 1.0);
   glBegin(GL_LINE_LOOP);
   glVertex3f(x1, y1, -z1);
```

продолжение ⇩

Листинг 13.2 (*продолжение*)

```
   glVertex3f(x2, y1, -z1);
   glVertex3f(x2, y2, -z1);
   glVertex3f(x1, y2, -z1);
   glEnd();

   glBegin(GL_LINE_LOOP);
   glVertex3f(x1, y1, -z2);
   glVertex3f(x2, y1, -z2);
   glVertex3f(x2, y2, -z2);
   glVertex3f(x1, y2, -z2);
   glEnd();

   glBegin(GL_LINES); /* 4 прямые */
   glVertex3f(x1, y1, -z1);
   glVertex3f(x1, y1, -z2);
   glVertex3f(x1, y2, -z1);
   glVertex3f(x1, y2, -z2);
   glVertex3f(x2, y1, -z1);
   glVertex3f(x2, y1, -z2);
   glVertex3f(x2, y2, -z1);
   glVertex3f(x2, y2, -z2);
   glEnd();
}

void drawScene(void)
{
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(40.0, 4.0/3.0, 1.0, 100.0);

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(7.5, 7.5, 12.5, 2.5, 2.5, -5.0, 0.0, 1.0, 0.0);
   glColor3f(0.0, 1.0, 0.0); /* зеленый треугольник */
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
   glColor3f(1.0, 0.0, 0.0); /* красный треугольник */
   drawTriangle(2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
   glColor3f(1.0, 1.0, 0.0); /* желтые треугольники */
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
   drawViewVolume(0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
}

void processHits(GLint hits, GLuint buffer[])
{
   unsigned int i, j;
   GLuint names, *ptr;

   printf("записей = %d\n", hits);
   ptr = (GLuint *) buffer;
   for (i = 0; i < hits; i++) { /* для каждой записи реакции */
      names = *ptr;
```

```
      printf(" число имен = %d\n", names); ptr++;
      printf(" z1 = %g;", (float) *ptr/0x7fffffff); ptr++;
      printf(" z2 = %g\n", (float) *ptr/0x7fffffff); ptr++;
      printf(" имя = ");
      for (j = 0; j < names; j++) { /* для каждого имени */
         printf("%d ", *ptr); ptr++;
      }
     printf("\n");
   }
}

#define BUFSIZE 512
void selectObjects(void)
{
   GLuint selectBuf[BUFSIZE];
   GLint hits;

   glSelectBuffer(BUFSIZE, selectBuf);
   (void) glRenderMode(GL_SELECT);

   glInitNames();
   glPushName(0);

   glPushMatrix();
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glLoadName(1);
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
   glLoadName(2);
   drawTriangle(2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
   glLoadName(3);
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
   glPopMatrix();
   glFlush();

   hits = glRenderMode(GL_RENDER);
   processHits(hits, selectBuf);
}

void init(void)
{
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

продолжение ⇩

Листинг 13.2 (*продолжение*)

```
    drawScene();
    selectObjects();
    glFlush();
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(200, 200);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Выбор с помощью указателя мыши

Процесс, описанный в предыдущем разделе, можно распространить на выбор объектов с помощью указателя мыши (picking). Для этого вы логически умножаете специальную матрицу на матрицу проекции с целью ограничить рисование небольшой областью окна просмотра, обычно рядом с указателем мыши. Затем вы организуете некоторую форму для реакции на ввод, такой как щелчок кнопкой мыши, чтобы активировать режим выбора. При всех перечисленных соблюденных условиях объекты, отображаемые вблизи указателя мыши, вызывают реакцию системы. Таким образом, во время такого выбора вы обычно определяете, какие объекты отображаются рядом с указателем.

Выбор с помощью указателя организуется почти так же, как обычный режим выбора, со следующим основными различиями:

- Такой выбор обычно производится переключением состояния (triggered) устройства позиционирования. В следующем примере кода нажатие левой кнопки мыши вызывает функцию, выполняющую выбор.
- Вы используете подпрограмму-инструмент `gluPickMatrix()` для умножения текущей матрицы проекции на специальную матрицу. Эта процедура должна быть вызвана до операции умножения матриц (такой как `gluPerspective()` или `glOrtho()`). Вероятно, вы захотите сохранить предшествующее состояние матрицы проекции, соответственно, последовательность операций будет выглядеть так:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPickMatrix(...);
gluPerspective, glOrtho, gluOrtho2D, or glFrustum
    /*... рисование сцены для выбора с помощью указателя; выполнение
выбора ... */
glPopMatrix();
```

Другой кардинально отличный способ выполнения выбора указателем мыши описан в разделе «Использование фонового буфера для выбора объектов» главы 14. Эта техника основана на идентификации различных компонентов объекта по значениям цвета.

void `gluPickMatrix`(GLdouble *x*, GLdouble *y*, GLdouble *width*, GLdouble *height*, GLint *viewport*[4]);

Создает матрицу проекции, ограничивающую рисование маленькой областью окна просмотра, и умножает матрицу на текущую матрицу в вершине стека. Центр области выбора задается оконными координатами (x, y) обычно в положении указателя мыши. Ширина *width* и высота *height* определяют размер области выбора в координатах экрана. (То есть ширину и высоту можно интерпретировать как чувствительность устройства позиционирования.) Массив *viewport*[] содержит текущие координаты окна просмотра, которые могут быть получены вызовом `glGetIntegerv(GL_VIEWPORT, GLint *viewport)`.

Результат, вырабатываемый командой `gluPickMatrix()`, преобразуется в область отсечения — единичный куб, где $-1 \le (x, y, z) \le 1$ (или $-w \le (wx, wy, wz) \le w$). Матрица выбора эффективно выполняет ортогональное преобразование, сводящее область отсечения к единичному кубу. Поскольку допускаются преобразования произвольного характера, вы можете делать различные действия над областью выбора — например, вращать прямоугольные участки окна. В отдельных ситуациях это легче, чем задавать дополнительные плоскости отсечения для определения области выбора.

Листинг 13.3 иллюстрирует выбор, делаемый с помощью указателя мыши. Здесь также показано, как используется множество имен для идентификации различных компонентов примитива; в этом случае объектами выбора являются строка и столбец. Рисуется сетка из квадратов 3 × 3 различных цветов. Массив *board*[3][3] хранит информацию о цветах для каждого квадрата. При нажатии левой кнопки мыши вызывается процедура `pickSquares()`, которая идентифицирует квадраты, выбранные мышью. С каждым квадратом сетки связаны два имени: одно — для строки и другое — для столбца. Также, когда нажимается левая кнопка мыши, изменяются цвета всех квадратов, расположенных в позиции указателя.

Листинг 13.3. Пример выбора с помощью указателя мыши: picksquare.c

```
int board[3][3]; /* количество цветов для каждого квадрата */

/* Сброс значений цвета для каждого квадрата доски */
void init(void)
{
  int i, j;
  for (i = 0; i < 3; i++)
     for (j = 0; j < 3; j ++)
        board[i][j] = 0;
  glClearColor(0.0, 0.0, 0.0, 0.0);
}
void drawSquares(GLenum mode)
{
```

продолжение ⇙

Листинг 13.3 (*продолжение*)

```
   GLuint i, j;

   for (i = 0; i < 3; i++) {
   if (mode == GL_SELECT)
      glLoadName(i);
   for (j = 0; j < 3; j ++) {
      if (mode == GL_SELECT)
         glPushName(j);
         glColor3f((GLfloat) i/3.0, (GLfloat) j/3.0,
            (GLfloat) board[i][j]/3.0);
         glRecti(i, j, i+1, j+1);
         if (mode == GL_SELECT)
            glPopName();
      }
   }
}

/* Процедура processHits выводит содержимое массива записей реакции */
void processHits(GLint hits, GLuint buffer[])
{
   unsigned int i, j;
   GLuint ii, jj, names, *ptr;

   printf("записей = %d\n", hits);
   ptr = (GLuint *) buffer;
   for (i = 0; i < hits; i++) { /* для каждой записи */
      names = *ptr;
      printf(" число имен для данной записи = %d\n", names);
      ptr++;
      printf(" z1 = %g;", (float) *ptr/0x7fffffff); ptr++;
      printf(" z2 = %g\n", (float) *ptr/0x7fffffff); ptr++;
      printf(" имена ");
      for (j = 0; j < names; j++) { /* для каждого имени */
         printf("%d ", *ptr);
         if (j == 0) /* установка строки и колонки */
            ii = *ptr;
         else if (j == 1)
            jj = *ptr;
         ptr++;
      }
      printf("\n");
      board[ii][jj] = (board[ii][jj] + 1) % 3;
   }
}

#define BUFSIZE 512

void pickSquares(int button, int state, int x, int y)
{
   GLuint selectBuf[BUFSIZE];
   GLint hits;
   GLint viewport[4];
```

```
   if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
      return;

   glGetIntegerv(GL_VIEWPORT, viewport);
   glSelectBuffer(BUFSIZE, selectBuf);
   (void) glRenderMode(GL_SELECT);

   glInitNames();
   glPushName(0);

   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
/* создает область выбора размером 5x5 пикселов в позиции указателя */
   gluPickMatrix((GLdouble) x, (GLdouble) (viewport[3] - y),
      5.0, 5.0, viewport);
   gluOrtho2D(0.0, 3.0, 0.0, 3.0);
   drawSquares(GL_SELECT);

   glMatrixMode(GL_PROJECTION);
   glPopMatrix();
   glFlush();

   hits = glRenderMode(GL_RENDER);
   processHits(hits, selectBuf);
   glutPostRedisplay();
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   drawSquares(GL_RENDER);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0, 3.0, 0.0, 3.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(100, 100);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
```

продолжение ⇘

Листинг 13.3 (*продолжение*)

```
    glutMouseFunc(pickSquares);
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Выбор мышью с помощью составных имен и иерархической модели

Составные имена могут также использоваться для выбора части иерархического объекта сцены. Например, если вы отображаете сборочную линию автомобилей, может потребоваться позволить пользователю передвинуть мышь и выделить третий болт на передней стороне колеса третьего автомобиля на линии. Для идентификации каждого уровня иерархии можно использовать отдельные имена: автомобиля, колеса и болта. Но есть и другой вариант, когда одним именем можно описать только одну молекулу во множестве молекул, где дополнительными именами можно различить отдельные атомы, принадлежащие молекуле.

Листинг 13.4 — это модификация листинга 3.4, где рисовался автомобиль с четырьмя одинаковыми колесами, а каждое колесо имело пять идентичных болтов. Код был дополнен лишь манипуляциями со стеком имен в иерархической модели объектов.

Листинг 13.4. Создание составных имен

```
draw_wheel_and_bolts()
{
    long i;
    draw_wheel_body();
    for (i = 0; i < 5; i++) {
        glPushMatrix();
            glRotate(72.0*i, 0.0, 0.0, 1.0);
            glTranslatef(3.0, 0.0, 0.0);
            glPushName(i);
            draw_bolt_body();
            glPopName();
        glPopMatrix();
    }
}

draw_body_and_wheel_and_bolts()
{
    draw_car_body();
    glPushMatrix();
        glTranslate(40, 0, 20); /* положение первого колеса */
        glPushName(1); /* имя колеса с номером 1 */
        draw_wheel_and_bolts();
        glPopName();
    glPopMatrix();
    glPushMatrix();
        glTranslate(40, 0, -20); /* положение второго колеса */
```

```
      glPushName(2); /* имя колеса с номером 2 */
      draw_wheel_and_bolts();
      glPopName();
   glPopMatrix();

/* аналогичное рисование для двух других колес */
}
```

В листинге 13.5 используются процедуры из листинга 13.4 для рисования трех различных автомобилей, нумерованных в порядке 1, 2 и 3.

Листинг 13.5. Использование составных имен

```
draw_three_cars()
{
   glInitNames();
   glPushMatrix();
      translate_to_first_car_position();
      glPushName(1);
      draw_body_and_wheel_and_bolts();
      glPopName();
   glPopMatrix();

   glPushMatrix();
      translate_to_second_car_position();
      glPushName(2);
      draw_body_and_wheel_and_bolts();
      glPopName();
   glPopMatrix();

   glPushMatrix();
      translate_to_third_car_position();
      glPushName(3);
      draw_body_and_wheel_and_bolts();
      glPopName();
   glPopMatrix();
}
```

Предположим, что выбор сделан. Ниже показаны некоторые возможные возвращаемые из стека имен значения и их интерпретация. В этих примерах в большинстве возвращается одна запись реакции; *d1* и *d2* — значения глубины.

2 *d1 d2* 2 1	Автомобиль 2, колесо 1
1 *d1 d2* 3	Корпус автомобиля 3
3 *d1 d2* 1 1 0	Болт 0 на колесе 1 автомобиля 1
пусто	Попытка выбора за пределами всех автомобилей

Последняя интерпретация исходит из того, что в одной области выделения нельзя одновременно выбрать болт и колесо. Пользователь может указать как колесо, так и болт, но это создает две записи реакции. В таком случае вы должны решить, какую запись обрабатывать, возможно, отталкиваясь от значения глубины, определенного для выбранного объекта и соответствующего максимальному прибли-

жению к точке наблюдения. Использование значений глубины разбирается в следующем разделе.

Выбор мышью и значения глубины

Листинг 13.6 демонстрирует, как использовать значения глубины для выбора из выбранных объектов. Эта программа рисует три перекрывающихся прямоугольника в нормальном режиме визуализации. Когда нажимается левая кнопка мыши, выполняется вызов процедуры `pickRects()`. Эта процедура возвращает позицию указателя, активирует режим выбора, инициализирует стек имен и производит умножение матрицы выбора на текущую матрицу ортогональной проекции. Выбор регистрируется для каждого прямоугольника, поверх которого находится указатель в момент нажатия левой кнопки мыши. Наконец, исследуется содержимое буфера выбора с целью идентификации именованных объектов, захваченных областью выбора.

Прямоугольники рисуются с различной глубиной, то есть значениями координаты *z*. Поскольку для идентификации всех трех прямоугольников используется одно имя, может быть зарегистрирована только одна запись реакции. Тем не менее эта одна запись обладает различающимися максимальным и минимальным значениями *z*.

Листинг 13.6. Выбор мышью с привлечением значений глубины: pickdepth.c

```
void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
   glDepthRange(0.0, 1.0); /* z-отображение по умолчанию */
}

void drawRects(GLenum mode)
{
   if (mode == GL_SELECT)
      glLoadName(1);
   glBegin(GL_QUADS);
   glColor3f(1.0, 1.0, 0.0);
   glVertex3i(2, 0, 0);
   glVertex3i(2, 6, 0);
   glVertex3i(6, 6, 0);
   glVertex3i(6, 0, 0);
   glEnd();
   if (mode == GL_SELECT)
      glLoadName(2);
   glBegin(GL_QUADS);
   glColor3f(0.0, 1.0, 1.0);
   glVertex3i(3, 2, -1);
   glVertex3i(3, 8, -1);
   glVertex3i(8, 8, -1);
   glVertex3i(8, 2, -1);
   glEnd();
```

```
   if (mode == GL_SELECT)
      glLoadName(3);
   glBegin(GL_QUADS);
   glColor3f(1.0, 0.0, 1.0);
   glVertex3i(0, 2, -2);
   glVertex3i(0, 7, -2);
   glVertex3i(5, 7, -2);
   glVertex3i(5, 2, -2);
   glEnd();
}

void processHits(GLint hits, GLuint buffer[])
{
   unsigned int i, j;
   GLuint names, *ptr;

   printf("записей = %d\n", hits);
   ptr = (GLuint *) buffer;
   for (i = 0; i < hits; i++) { /* для каждой записи */
      names = *ptr;
      printf(" количество имен для записи = %d\n", names); ptr++;
      printf(" z1 = %g;", (float) *ptr/0x7fffffff); ptr++;
      printf(" z2 = %g\n", (float) *ptr/0x7fffffff); ptr++;
      printf(" имя ");
      for (j = 0; j < names; j++) { /* для каждого имени */
         printf("%d ", *ptr); ptr++;
      }
      printf("\n");
   }
}

#define BUFSIZE 512
void pickRects(int button, int state, int x, int y)
{
   GLuint selectBuf[BUFSIZE];
   GLint hits;
   GLint viewport[4];
   if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
      return;
   glGetIntegerv(GL_VIEWPORT, viewport);

   glSelectBuffer(BUFSIZE, selectBuf);
   (void) glRenderMode(GL_SELECT);

   glInitNames();
   glPushName(0);

   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
/* создает область выбора размером 5x5 пикселов в позиции указателя мыши */
   gluPickMatrix((GLdouble) x, (GLdouble) (viewport[3] - y),
                 5.0, 5.0, viewport);
```

продолжение ⇩

Листинг 13.6 (*продолжение*)

```
    glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
    drawRects(GL_SELECT);
    glPopMatrix();
    glFlush();

    hits = glRenderMode(GL_RENDER);
    processHits(hits, selectBuf);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    drawRects(GL_RENDER);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(200, 200);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutMouseFunc(pickRects);
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Упражнения

- Измените листинг 13.6, добавив вызовы `glPushName()` так, чтобы при выборе в стеке сохранялись составные имена. Каково при этом будет содержимое буфера выбора?
- По умолчанию команда `glDepthRange()` устанавливает отображение *z*-значений в диапазон [0.0, 1.0]. Попробуйте изменить значения `glDepthRange()` и посмотрите, как ваши модификации повлияют на *z*-значения, возвращаемые в массиве записей нажатий.

Рекомендации по написанию программы, использующей механизм выбора

Большинство программ, позволяющих пользователю интерактивно редактировать геометрические объекты, предоставляют пользователю механизм для предварительного выбора групп элементов или отдельных элементов. Для программ двухмерного рисования (например, текстовых редакторов, программ верстки и проектирования электронных схем) может быть легче делать собственные вычисления для выбора, чем обращаться к механизму OpenGL. Часто нетрудно выявить контуры, ограничивающие двухмерные объекты, и выстроить их в иерархическую структуру для ускорения доступа. Например, выбор в стиле OpenGL для программ конструирования сверхбольших интегральных схем (VLSI), содержащих миллионы прямоугольников, может оказаться относительно медленным. Вместе с тем учет информации об ограничивающих рамках, когда прямоугольники обычно выровнены относительно экрана, может чрезвычайно ускорить процесс выбора в такой программе. Надо полагать, что и написать такой код тоже проще.

Другой пример. Поскольку нажатия регистрируются только для геометрических объектов, вы можете захотеть создать собственный метод для выбора текста. Установка текущей позиции растра — это геометрическая операция, но она эффективна только в отношении одиночной доступной для выбора точки, обычно по ней можно опознать только левый нижний угол текста. Если ваша программа-редактор требует манипулирования отдельными символами текстовой строки, необходим другой механизм выбора. Вы можете рисовать небольшие прямоугольники вокруг каждого символа в режиме выбора, но вряд ли такая обработка текста проще, да и носит индивидуальный характер.

Если вы решите использовать механизм выбора OpenGL, организуйте вашу программу и ее структуры данных так, чтобы облегчить рисование соответствующих списков объектов как в обычном режиме визуализации, так и в режиме выбора. В этом случае, когда пользователь выделяет что-либо мышью, вы можете использовать те же структуры данных для операции выбора, с помощью которых вы отображали элементы на экране. Также определитесь с тем, разрешить ли пользователю выбирать сложные объекты. Это можно сделать посредством выделения одного бита каждому подобъекту при сохранении устанавливаемого в момент выбора объекта (такой путь, однако, требует от вас сканирования полного списка элементов для нахождения выделенных). Вы можете посчитать это полезным для поддержания списка указателей на выбранные элементы в расчете на ускорение поиска. Возможно, это неплохая идея — выделение бита элементу, поскольку, когда вы рисуете полную картинку, вы можете захотеть отобразить выбранные элементы как-то иначе (например, другим цветом или в пределах рамки выделения). Наконец, примите во внимание интерфейс пользователя, которому можно разрешить:

- выбор элемента;
- захват группы элементов;
- добавление элемента к выделению;

- добавление захваченной группы к текущему выделению;
- удаление элемента из числа выделенных;
- выбор одного элемента из группы перекрывающихся элементов.

Типовое решение для программы двухмерного рисования может работать следующим образом:

1. Выбор всегда делается позиционированием указателя мыши и щелчком ее левой кнопки. Поэтому далее под *указателем* (cursor) мы будем всегда понимать указатель мыши, а не курсор клавиатуры, например. А под обобщенной *кнопкой* будет скрываться левая кнопка мыши.
2. Щелчок на элементе выделяет его и снимает выделение с любого другого ранее выделенного элемента. Если указатель находится поверх множества элементов, выделяется наименьший. (В трех измерениях существует много стратегий устранения неоднозначности выбора.)
3. При нажатии кнопки вне области с элементами, последующем ее удержании во время перетаскивания указателя и отпускании выделяются все элементы, попавшие в образованный этой операцией прямоугольник. Это действие мы назовем *захватом* (sweep selection). Элементы вне захваченной области не выделяются. Здесь вам нужно определиться, считать ли элемент выбранным на основании только полного попадания его в область захвата, или для этого достаточно частичного перекрытия элементом области. Стратегия «полностью внутри» обычно работает лучше.
4. Если в момент щелчка на элементе удерживается клавиша `Shift`, этот элемент добавляется в список выбранных. Если же этот элемент уже присутствует в списке, он, наоборот, удаляется из выделения.
5. Когда захват выполняется при нажатой клавише `Shift`, элементы добавляются к текущему выделению.
6. В чрезвычайно загроможденной области часто тяжело сделать правильный захват. Когда кнопка идет вниз, курсор может быть наведен на некоторый элемент, что в нормальном режиме соответствует выбору элемента. Вы можете делать любые операции захвата, но обычно интерфейс пользователя интерпретирует нажатие кнопки на элементе совместно с движением мыши как операцию «выбор плюс перетаскивание». Чтобы решить эту проблему, вы можете вынудить программу проводить захват при нажатии другой клавиши, скажем, клавиши `Alt`. Тогда операция захвата будет сводиться к последовательности из нажатия клавиши `Alt`, собственно захвата и отпускания кнопки. Тогда элементы, попавшие под указатель в момент нажатия кнопки, будут игнорироваться.
7. Если удерживать во время захвата клавишу `Shift`, все захваченные элементы будут добавлены к выделению.
8. Наконец, если пользователь щелкает на группе элементов, просто выберите один из них. Если указатель в этот момент недвижим (условно, он может смещаться в пределах пиксела) и пользователь щелкает повторно в том же месте,

снимите выделение с другого элемента и выделите элемент, находящийся под курсором. То есть используйте повторные щелчки в одной и той же точке для циклического прохода по всем возможным элементам.

К различным ситуациям применимы различные правила. В текстовом редакторе, надо полагать, вас не должно волновать расположение символов друг над другом, так как выбор множества символов всегда заключается в выборе последовательных символов документа. Таким образом, нужно пометить только начало и конец фрагмента для идентификации полного выделения. При работе с текстом часто лучше обрабатывать выделение, идентифицируя позиции между символами, а не сами символы. Это позволяет вам иметь пустой выбор, когда начало и конец выделения совпадают; это также позволяет вам устанавливать указатель перед первым символом документа или после последнего, не обращаясь к специальным кодам.

Для «трехмерного» редактора вы можете предоставить способы вращения и масштабирования между выделениями, поэтому отпадает нужда в изощренных схемах циклического обхода возможных вариантов. С другой стороны, выбор в трех измерениях труднее, так как по позиции указателя на экране нельзя узнать глубину объектов.

Обратная связь

Обратная связь напоминает выбор, когда сгенерированные пикселы не торопятся попасть на экран. Рисования не происходит; вместо этого информация о примитивах, которые могли быть отрисованы, посылается назад приложению. Ключевое различие между режимами выбора и обратной связи — это возврат информации. В режиме выделения присвоенные имена заносятся в массив целочисленных значений. В режиме обратной связи информация о преобразованных примитивах помещается в возвращаемый массив значений с плавающей запятой. Возвращаемые назад значения состоят из маркеров, указывающих тип примитива (точка, линия, многоугольник, изображение или битовый образ), обработанного и трансформированного, сопровождаемых вершинами, цветом и другими данными примитивов. Возвращаемые значения преобразованы освещением и видовыми операциями. Режим обратной связи активируется вызовом `glRenderMode()` с параметром `GL_FEEDBACK` в качестве аргумента.

Ниже описано, как выполняется вход в режим обратной связи и выход из него.

1. Вызовите команду `glFeedbackBuffer()` для задания массива, предназначенного для хранения данных обратной связи. Аргументы этой процедуры отвечают за то, данные какого типа и в каком количестве записываются в массив.
2. Вызовите `glRenderMode()` с параметром `GL_FEEDBACK` в качестве аргумента для входа в режим обратной связи. (На этом шаге вы можете игнорировать значение, возвращаемое `glRenderMode()`.) В дальнейшем примитивы не растеризуются в пикселы, пока вы не выйдете из режима обратной связи, а также не изменяется содержимое буфера кадров.

3. Рисуйте свои примитивы. Пока выполняются команды рисования, вы можете сделать несколько вызовов `glPassThrough()`, чтобы вставить маркеры в данные обратной связи и тем самым способствовать их анализу.
4. Покиньте режим обратной связи с помощью команды `glRenderMode()` с параметром `GL_RENDER` в качестве аргумента, если хотите вернуться в нормальный режим рисования. Целое значение, возвращаемое `glRenderMode()`, — это количество значений, сохраненных в массиве обратной связи.
5. Выполните разбор данных из массива обратной связи.

void `glFeedbackBuffer`(GLsizei *size*, GLenum *type*, GLfloat **buffer*);

Выделяет буфер для посылаемых назад в программу данных: аргумент *buffer* — это указатель на массив, отводимый под хранение данных. Размер *size* задает максимальное количество значений в массиве — его вместимость. Тип *type* описывает информацию обратной связи для каждой вершины в возвращаемом массиве; допустимые типы и их значения перечислены в табл. 13.1. Команда `glFeedbackBuffer()` должна быть вызвана до переключения в режим обратной связи. В таблице обозначение *k* равно 1 для режима индексации цвета и 4 — для RGBA-режима.

Таблица 13.1. Типы значений glFeedbackBuffer()

Тип	Координаты	Цвет	Текстура	Всего значений
GL_2D	x, y	–	–	2
GL_3D	x, y, z	–	–	3
GL_3D_COLOR	x, y, z	k	–	3 + k
GL_3D_COLOR_TEXTURE	x, y, z	k	4	7 + k
GL_4D_COLOR_TEXTURE	x, y, z, w	k	4	8 + k

ПРИМЕЧАНИЕ

Если поддержано мультитекстурирование, обратная связь возвращает только координаты текстуры модуля 0.

Массив обратной связи

В режиме обратной связи каждый подлежащий растеризации примитив (или каждый вызов `glBitmap()`, `glDrawPixels()` или `glCopyPixels()`, если позиция растра допустима) генерирует блок значений, копируемых в массив обратной связи. Количество значений зависит от аргумента *type* функции `glFeedbackBuffer()`, как показано в табл. 13.1. Используйте значения, соответствующие типу отображаемого примитива: `GL_2D` или `GL_3D` для неосвещенных двух- или трехмерных примитивов; `GL_3D_COLOR` — для освещенных трехмерных примитивов и `GL_3D_COLOR_TEXTURE` или `GL_4D_COLOR_TEXTURE` — для освещенных и текстурированных трех- или четырехмерных примитивов.

Каждый блок значений обратной связи начинается с кода, указывающего тип примитива, его продолжают значения, описывающие вершины примитива и связанные данные. Отдельные блоки выделяются для прямоугольников пикселов.

Кроме того, в массиве возвращаются явно созданные маркеры; следующий раздел рассматривает эти маркеры в подробностях. В табл. 13.2 показан синтаксис массива обратной связи; помните, что данные, связанные с каждой возвращаемой вершиной, описываются в табл. 13.1. Обратите внимание, что многоугольник может иметь n возвращаемых вершин. Также заметьте, что x-, y- и z-координаты в массиве — оконные; если возвращается координата w, то в координатах отсечения. Для битовых образов и прямоугольников пикселов координаты возвращаются относительно текущей позиции растра. Учтите, что маркер `GL_LINE_RESET_TOKEN` возвращается только для прерывистых линий при сбросе прерывания для отрезка.

Таблица 13.2. Синтаксис массива обратной связи

Тип примитива	Код	Связанные данные
Точка	GL_POINT_TOKEN	Вершина
Прямая	GL_LINE_TOKEN или GL_LINE_RESET_TOKEN	Вершина вершина
Многоугольник	GL_POLYGON_TOKEN	n вершина вершина... вершина
Битовый образ	GL_BITMAP_TOKEN	Вершина
Прямоугольник пикселов	GL_DRAW_PIXEL_TOKEN или GL_COPY_PIXEL_TOKEN	Вершина
Маркер	GL_PASS_THROUGH_TOKEN	Число с плавающей запятой

Использование маркеров в режиме обратной связи

Обратная связь имеет смысл после применения освещения, отбраковки многоугольников и интерпретации многоугольников командой `glPolygonMode()`. Она может иметь место также после разбиения многоугольников более чем с четырьмя ребрами на треугольники (если ваша реализация OpenGL применяет такую декомпозицию к многоугольникам). Таким образом, для вас может быть нелегко распознать примитивы, возвращаемые обратной связью. В помощь при разборе таких данных предусмотрена команда `glPassThrough()`, которая вызывается там, где нужно пометить место в последовательности команд рисования. Например, вы можете использовать вставляемые маркеры для разделения между собой возвращаемых значений, соответствующих разным примитивам. Эта команда вызывает запись маркера `GL_PASS_THROUGH_TOKEN` в массив обратной связи, сопровождаемого значением с плавающей запятой, переданным в качестве аргумента.

void `glPassThrough`(GLfloat *token*);

Вставляет маркер в поток значений, записываемых в массив обратной связи. Маркер состоит из кода `GL_PASS_THROUGH_TOKEN` и значения с плавающей запятой — *token*. Эта команда имеет эффект, когда вызов производится из режима обратной связи. Вызовы `glPassThrough()` между `glBegin()` и `glEnd()` генерируют ошибку `GL_INVALID_OPERATION`.

Пример обратной связи

Листинг 13.7 демонстрирует использование режима обратной связи. Данная программа рисует освещенную трехмерную сцену в обычном режиме визуализации. Затем включается режим обратной связи и сцена перерисовывается. Поскольку программа моделирует освещенные нетекстурированные трехмерные объекты, тип данных обратной связи задан как `GL_3D_COLOR`. Так как используется режим RGBA, для каждой неотсеченной вершины генерируется семь значений в буфер обратной связи: *x*, *y*, *z*, *r*, *g*, *b* и *a*.

Находясь в режиме обратной связи, программа рисует две линии как часть прерывистой линии и затем вставляет маркер. Далее рисуется точка с координатами (−100.0, −100.0, −100.0), выходящими за пределы видимого объема в ортогональной проекции. В результате здесь никакие значения в массив не попадают. Наконец, вставляется другой маркер и рисуется следующая точка.

Листинг 13.7. Режим обратной связи: feedback.c

```
void init(void)
{
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
}

void drawGeometry(GLenum mode)
{
   glBegin(GL_LINE_STRIP);
   glNormal3f(0.0, 0.0, 1.0);
   glVertex3f(30.0, 30.0, 0.0);
   glVertex3f(50.0, 60.0, 0.0);
   glVertex3f(70.0, 40.0, 0.0);
   glEnd();
   if (mode == GL_FEEDBACK)
      glPassThrough(1.0);
   glBegin(GL_POINTS);
   glVertex3f(-100.0, -100.0, -100.0); /* будет отсечена */
   glEnd();
   if (mode == GL_FEEDBACK)
      glPassThrough(2.0);
   glBegin(GL_POINTS);
   glNormal3f(0.0, 0.0, 1.0);
   glVertex3f(50.0, 50.0, 0.0);
   glEnd();
   glFlush();
}

void print3DcolorVertex(GLint size, GLint *count,
                        GLfloat *buffer)
{
   int i;

   printf(" ");
   for (i = 0; i < 7; i++) {
```

```
      printf("%4.2f ", buffer[size-(*count)]);
         *count = *count - 1;
   }
   printf("\n");
}

void printBuffer(GLint size, GLfloat *buffer)
{
   GLint count;
   GLfloat token;

   count = size;
   while (count) {
      token = buffer[size-count]; count--;
      if (token == GL_PASS_THROUGH_TOKEN) {
         printf("GL_PASS_THROUGH_TOKEN\n");
         printf(" %4.2f\n", buffer[size-count]);
         count--;
      }
      else if (token == GL_POINT_TOKEN) {
         printf("GL_POINT_TOKEN\n");
         print3DcolorVertex(size, &count, buffer);
      }
      else if (token == GL_LINE_TOKEN) {
         printf("GL_LINE_TOKEN\n");
         print3DcolorVertex(size, &count, buffer);
         print3DcolorVertex(size, &count, buffer);
      }
      else if (token == GL_LINE_RESET_TOKEN) {
         printf("GL_LINE_RESET_TOKEN\n");
         print3DcolorVertex(size, &count, buffer);
         print3DcolorVertex(size, &count, buffer);
      }
   }
}

void display(void)
{
   GLfloat feedBuffer[1024];
   GLint size;

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 100.0, 0.0, 100.0, 0.0, 1.0);

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   drawGeometry(GL_RENDER);

   glFeedbackBuffer(1024, GL_3D_COLOR, feedBuffer);
   (void) glRenderMode(GL_FEEDBACK);
   drawGeometry(GL_FEEDBACK);
   size = glRenderMode(GL_RENDER);
```

продолжение ⇩

Листинг 13.7 (*продолжение*)

```
    printBuffer(size, feedBuffer);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(100, 100);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Запуск этой программы приводит к генерации следующего потока вывода:

```
GL_LINE_RESET_TOKEN
 30.00 30.00 0.00 0.84 0.84 0.84 1.00
 50.00 60.00 0.00 0.84 0.84 0.84 1.00
GL_LINE_TOKEN
 50.00 60.00 0.00 0.84 0.84 0.84 1.00
 70.00 40.00 0.00 0.84 0.84 0.84 1.00
GL_PASS_THROUGH_TOKEN
 1.00
GL_PASS_THROUGH_TOKEN
 2.00
GL_POINT_TOKEN
 50.00 50.00 0.00 0.84 0.84 0.84 1.00
```

Таким образом, рисование прерывистой линии этими командами равнозначно двум примитивам:

```
glBegin(GL_LINE_STRIP);
    glNormal3f(0.0, 0.0, 1.0);
    glVertex3f(30.0, 30.0, 0.0);
    glVertex3f(50.0, 60.0, 0.0);
    glVertex3f(70.0, 40.0, 0.0);
glEnd();
```

Первый примитив начинается маркером `GL_LINE_RESET_TOKEN`, который указывает, что этот примитив является отрезком и что после него линия прерывается. Второй примитив начинается с `GL_LINE_TOKEN`, поэтому это также отрезок, но линия прервана и, следовательно, продолжается после конца предыдущего отрезка. Каждая из двух вершин этих линий вырабатывает семь значений для помещения в массив обратной связи. Заметьте, что RGBA-значения для всех четырех вершин — (0.84, 0.84, 0.84, 1.0), что соответствует очень светлому серому цвету с максимальным значением альфа-компонента. Эти цветовые значения — результат, выведенный из нормали к поверхности и параметров освещения.

Поскольку никаких данных обратной связи не генерируется между первым и вторым маркерами, вы можете догадаться, что любой примитив, нарисованный меж-

ду двумя вызовами `glPassThrough()`, был отсечен по видимому объему. Наконец, рисуется точка (50.0, 50.0, 0.0) и связанные с ней данные копируются в буфер обратной связи.

ПРИМЕЧАНИЕ

В обоих режимах — обратной связи и выбора — информация об объектах возвращается до тестов фрагментов. Таким образом, объекты, которые в обычном режиме не были бы нарисованы по причине неудачи альфа-теста, тестов отсечения, глубины или трафарета, все еще могут быть обработаны.

Упражнения

Сделайте изменения в листинге 13.7, чтобы увидеть, как они воздействуют на возвращаемые значения обратной связи. Например, координаты для `glOrtho()`. «Поиграйте» параметрами освещения или вовсе исключите освещение, соответственно, изменив тип обратной связи на `GL_3D`. Или же добавьте примитивов для оценки того, какой вклад вносят другие геометрические объекты (такие как закрашенные многоугольники) в массив обратной связи.

Теперь, когда вы знаете многое 14

Данная глава не ставит перед собой конкретную цель, в отличие от предшествующих. Это просто описание идей, которые вы можете найти полезными для своих приложений. Некоторые темы, такие как обработка ошибок, не вписываются в другие категории, и при этом слишком коротки для их выделения в отдельную главу.

OpenGL — тоже собрание инструментов низкого уровня; теперь, когда вы с ними познакомились, вы можете использовать их для реализации функций более высокого уровня. В этой главе представлено несколько примеров таких высокоуровневых инструментов.

В этой главе рассматриваются различные методики, основанные на командах OpenGL, иллюстрирующие некоторые не совсем очевидные точки приложения этих команд. Примеры представлены не в каком-либо логическом порядке и никак не связаны друг с другом. Идея состоит в том, чтобы прочитать заголовки разделов и пропустить не интересующие вас примеры. Для вашего удобства эти заголовки перечислены ниже, вместе с краткими пояснениями.

ПРИМЕЧАНИЕ

Большинство примеров в остальной части этого руководства являются полнофункциональными и могут быть скомпилированы и запущены на выполнение. Однако в этой главе нет законченных программ, и вам нужно немного поработать, чтобы они могли запускаться.

- **Обработка ошибок.** Рассказывает, как нужно обращаться с ошибками OpenGL.
- **Какую версию я использую?** Дает ответ на вопрос, где найти подробную информацию о реализации, включая номер версии. Это может быть полезно для написания приложений, способных запускаться под ранней версией OpenGL.
- **Расширения стандарта.** Представляет технологии для идентификации и использования расширений стандарта OpenGL (специфичных в зависимости от поставщика или одобренных ассоциацией OpenGL Architecture Review Board).
- **Полупрозрачность.** Объясняет, как использовать штриховку многоугольников для достижения полупрозрачности, что особенно полезно, когда аппаратное обеспечение не поддерживает смешивание.
- **Простой эффект «растворения» изображения.** Показывает, как использовать штриховку многоугольников для получения эффекта плавного обесцвечивания до уровня фона.
- **Использование фонового буфера для выбора объектов.** Описывает, как использовать фоновый буфер в системах с двойной буферизацией для обработки выбора объектов с помощью мыши.
- **Простая трансформация изображения.** Здесь обсуждается, как нарисовать деформированное изображение, представляя каждый пиксел четырехугольником.
- **Послойное отображение.** Объясняет, как отобразить множество слоев различных материалов и задать место перекрытия материалов.
- **Сглаживание символов.** Описывает, как отображать сглаженные шрифты.
- **Рисование округлых точек.** Описывает, как рисовать точки округлой формы.
- **Интерполяция изображений.** Показывает, как гладко смешать одно изображение с другим.
- **Создание устойчивых изображений.** Объясняет, как нарисовать два изображения, одно из которых представляет собой деколь, всегда появляющуюся поверх другого изображения.
- **Рисование закрашенных вогнутых многоугольников с помощью буфера трафарета.** Учит тому, как нарисовать с помощью буфера трафарета вогнутые

многоугольники, более сложные многоугольники и многоугольники с отверстиями.

- **Нахождение перекрывающихся областей.** Описывает, как выяснить, в каком месте накладываются друг на друга части трехмерного изображения.
- **Тени.** Описывает, как нарисовать тени для освещенных объектов.
- **Удаление невидимых линий.** Отвечает на вопрос, как нарисовать «проволочный» объект со скрытыми (удаленными) линиями, используя буфер трафарета.
- **Текстурирование.** Описывает несколько трюков при использовании наложения текстур, таких как вращение и трансформация одновременного увеличения и уменьшения изображений.
- **Рисование изображений с буферизацией глубины.** Говорит, как скомбинировать изображения в среде с буферизацией глубины.
- **Области Дирихле.** Объясняет, как найти область Дирихле для набора точек с помощью буфера глубины.
- **«Жизнь» в буфере трафарета.** Пример реализации игры «Жизнь» с использованием буфера трафарета.
- **Альтернативное применение glDrawPixels() и glCopyPixels().** Описывает, как использовать эти две команды для получения таких эффектов, как псевдовидео, аэрограф, а также для перестановки изображений.

Обработка ошибок

Из практики вытекает, что любое приложение, в том числе и ваше, не свободно от ошибок. Используйте подпрограммы-обработчики ошибок — это неотъемлемая сущность разработки и это строго рекомендуется для коммерческих приложений (разве что вы на 100% уверены, что ваша программа никогда не создаст аварийного состояния для OpenGL. Смотрите на вещи реально!). OpenGL имеет простые процедуры обработки ошибок в своей базовой библиотеке GL и в GLU.

Когда OpenGL обнаруживает ошибку (GL или GLU), регистрируется код текущей ошибки. Команда, послужившая причиной сбоя, игнорируется, поэтому она не влияет на состояние OpenGL или на содержимое буфера кадров. Тем не менее, если фиксируется ошибка `GL_OUT_OF_MEMORY`, результаты работы команды не определены. Однажды зарегистрированный код текущей ошибки не сбрасывается — то есть дополнительные ошибки не фиксируются до тех пор, пока вы не вызовете команду запроса `glGetError()`, возвращающую код текущей ошибки. После того как вы запросите и тем самым сбросите код текущей ошибки или если ошибок еще не было, как в начале выполнения, команда `glGetError()` будет возвращать значение `GL_NO_ERROR`.

GLenum glGetError(void);
Возвращает значение текущего признака ошибки. Когда ошибка фиксируется для команд GL или GLU, признак ошибки — это соответствующее значение кода ошибки. Если возвращено значение GL_NO_ERROR, это означает, что никаких ошибок с момента последнего вызова glGetError() не было или GL еще только инициализирована. Во время вызова glGetError(), возврата ею кода ошибки и сброса признака в GL_NO_ERROR не регистрируются никакие другие ошибки.

Настоятельно рекомендуется вызывать glGetError() хотя бы раз за одно выполнение процедуры display(). Таблица. 14.1 содержит описание основных кодов ошибок OpenGL.

Таблица 14.1. Коды ошибок OpenGL

Код ошибки	Описание
GL_INVALID_ENUM	Значение аргумента типа GLenum вне диапазона
GL_INVALID_VALUE	Числовой аргумент вне допустимого диапазона
GL_INVALID_OPERATION	Операция, недопустимая для текущего состояния
GL_STACK_OVERFLOW	Команда повлекла переполнение стека
GL_STACK_UNDERFLOW	Команда вызвала антипереполнение стека
GL_OUT_OF_MEMORY	Недостаточно памяти для выполнения команды

Также имеется 37 ошибок GLU NURBS (с неописательными символическими именами GLU_NURBS_ERROR1, GLU_NURBS_ERROR2 и др.); 14 ошибок тесселяции (GLU_TESS_MISSING_BEGIN_POLYGON, GLU_TESS_MISSING_END_POLYGON, GLU_TESS_MISSING_BEGIN_CONTOUR, GLU_TESS_MISSING_END_CONTOUR, GLU_TESS_COORD_ TOO_LARGE, GLU_TESS_NEED_COMBINE_CALLBACK, и восемь ошибок с именами GLU_TESS_ERROR*); а также ошибка GLU_INCOMPATIBLE_GL_VERSION. Кроме того, для GLU определены коды GLU_INVALID_ENUM, GLU_INVALID_VALUE и GLU_OUT_OF_MEMORY, интерпретируемые аналогично подобным кодам OpenGL.

Для получения пригодной для вывода, наглядной строки, соответствующей коду ошибки GL или GLU, используйте команду GLU gluErrorString().

const GLubyte* gluErrorString(GLenum *errorCode*);
Возвращает указатель на описательную строку, соответствующую номеру ошибки OpenGL или GLU, передаваемую в аргументе *errorCode*.

В листинге 14.1 показана простая подпрограмма обработки ошибок.

Листинг 14.1. Запрос и вывод информации об ошибке

```
GLenum errCode;
const GLubyte *errString;
if ((errCode = glGetError())!= GL_NO_ERROR) {
   errString = gluErrorString(errCode);
   fprintf (stderr, "Ошибка OpenGL: %s\n", errString);
}
```

ПРИМЕЧАНИЕ

Строка, возвращаемая gluErrorString(), не может быть переделана или освобождена приложением.

Какую версию я использую?

Переносимость приложений OpenGL — это одна из привлекательных характеристик OpenGL. Однако новые версии OpenGL предоставляют новые возможности, и если вы используете последние, у вас могут быть проблемы с запуском своего кода под старыми версиями OpenGL. Кроме того, вы можете захотеть, чтобы приложение работало одинаково хорошо в различных реализациях. Иначе, например, вы можете делать наложение текстур в обычном (по умолчанию) режиме визуализации на одном компьютере, но получить лишенную полутонов, неосвещенную версию сцены на другом. Здесь вам поможет команда `glGetString()`, сообщающая информацию относительно вашей реализации OpenGL.

const GLubyte* `glGetString`(GLenum *name*);

Возвращает указатель на строку, дающую оценку свойств реализации OpenGL. Аргумент *name* может быть одним из следующих: `GL_VENDOR`, `GL_RENDERER`, `GL_VERSION` или `GL_EXTENSIONS`.

Параметр `GL_VENDOR` возвращает название компании, ответственной за реализацию OpenGL. Значение `GL_RENDERER` возвращает идентификатор устройства визуализации, который обычно представляет физическое устройство. За большей информацией о `GL_EXTENSIONS` обращайтесь к разделу «Расширения стандарта».

При задании параметра `GL_VERSION` возвращается строка с номером версии вашей реализации OpenGL. Порядок ее элементов следующий:

```
<номер версии><space><информация, зависимая от распространителя>
```

Номер версии имеет вид

```
основной_номер.дополнительный_номер
```

или

```
основной_номер.дополнительный_номер.номер_реализации
```

где все числа содержат один и более разрядов. Информация, зависимая от распространителя, не обязательна. Например, если данная реализация OpenGL исходит от некоей корпорации XYZ, возвращаемая строка будет выглядеть так:

```
1.2.4 XYZ - OS 3.2
```

Это означает, что мы имеем дело с четвертой редакцией реализации OpenGL от корпорации XYZ, соответствующей техническим требованиям OpenGL версии 1.2. Вероятно также, что это реализация 3.2 собственной операционной системы XYZ.

Другой способ запросить номер версии OpenGL — использовать предложение препроцессора `#ifdef` на предмет присутствия символических имен `GL_VERSION_1_4`, `GL_VERSION_1_3`, `GL_VERSION_1_2` и `GL_VERSION_1_1`. Отсутствие константы `GL_VERSION_1_2` свидетельствует, что номер вашей версии OpenGL — 1.0 или 1.1. Отсутствие константы `GL_VERSION_1_1` означает, что вы имеете дело с версией 1.0.

ПРИМЕЧАНИЕ

В системе «клиент-сервер», такой, где выполняется непосредственная визуализация с помощью расширения OpenGL для X Window System, клиент и сервер могут иметь различные версии. Если ваша версия клиента новее серверной, клиент может выдать запрос на операцию, не поддерживаемую сервером.

Версия библиотеки инструментов

Функция запроса `gluGetString()` библиотеки инструментов (GLU) аналогична `glGetString()`.

const GLubyte* `gluGetString`(GLenum *name*);

Возвращает указатель на строку с характеристикой OpenGL-реализации. Аргумент *name* задается равным `GLU_VERSION` или `GLU_EXTENSIONS`.

Заметьте, что `gluGetString()` недоступна для GLU 1.0. Другой способ запросить номер версии GLU — выяснить, присутствует ли символическое имя `GLU_VERSION_1_3`. Отсутствие этой константы означает, что вы работаете с версией GLU 1.2 или более ранней. Тот же принцип имеет силу для констант `GLU_VERSION_1_1` и `GLU_VERSION_1_2`.

Также обратите внимание, что расширения GLU отличаются от расширений OpenGL.

Версии расширений для оконных систем

Для расширений оконных систем, таких как GLX, WGL, PGL и AGL, также существуют подобные подпрограммы (такие как `glXQueryExtensionString()`), запрашивающие информацию о версии. (См. приложение B для получения большей информации.)

(Вышеупомянутая команда `glXQueryExtensionString()` и связанные с ней подпрограммы были введены в GLX 1.1 и не поддерживаются для GLX 1.0.)

Расширения стандарта

OpenGL имеет формальные, зафиксированные на бумаге, технические требования, которые описывают операции, включенные в библиотеку. Отдельный распространитель или их группа могут решить внести дополнительную функциональность в своей редакции реализации. Имена функций и символические константы четко указывают, что относится к стандартным возможностям OpenGL, какая часть представляет собой расширение от распространителя или OpenGL-расширение, утвержденное ARB (Architecture Review Board).

Для создания зависящего от поставщика имени этот поставщик добавляет в конец названия идентификатор компании (заглавными буквами) и, по необходимости, дополнительную информацию, такую как номер компьютера. Например, если корпорация XYZ желает ввести новую функцию или символическое имя, оно может иметь вид `glCommandXYZ()` или `GL_DEFINITION_XYZ`. Если расширение кор-

порации XYZ работоспособно только для ее собственного видеоускорителя FooBar, имена могут выглядеть как `glCommandXYZfb()` и `GL_DEFINITION_XYZ_FB`.

Если два или более поставщика достигли договоренности о реализации одного расширения, эти функции и константы получают более общий суффикс `EXT` (`glCommandEXT()` и `GL_DEFINITION_EXT`).

Аналогично, если ARB одобряет расширение, функции и константы получают суффикс `ARB` (`glCommandARB()` и `GL_DEFINITION_ARB`).

Если вам нужно узнать, поддерживается ли определенное расширение в вашей реализации, сначала используйте `glGetString(GL_EXTENSIONS)`, а затем — `gluGetString(GLU_EXTENSIONS)`. В результате будет возвращен список названий всех имеющихся расширений, разделенных между собой пробелами.

Первые одобренные ARB расширения были представлены в OpenGL 1.2. Если поддерживается необязательное расширение Imaging Subset, в возвращаемой строке присутствует обозначение `GL_ARB_imaging`. Строка `GL_ARB_multitexture` указывает на присутствие одобренного ARB расширения для мультитекстурирования.

Если вы работаете с GLU 1.3, вы можете использовать команду `gluCheckExtension()` для проверки, поддерживается ли конкретное расширение.

GLboolean `gluCheckExtension`(char **extName*, const GLubyte **extString*);
Возвращает `GL_TRUE`, если имя *extName* найдено в *extString*, или `GL_FALSE` в противном случае.

Команда `gluCheckExtension()` способна проверить строки расширений OpenGL и GLX, а также GLU.

В версиях GLU до 1.3, чтобы убедиться в поддержке расширения, используйте код из листинга 14.2 для сканирования списка и поиска нужного имени. Процедура `QueryExtension()` возвращает `GL_TRUE`, если строка обнаружена, и `GL_FALSE` в противном случае.

Листинг 14.2. Проверка на поддержку расширения (до GLU 1.3)

```
static GLboolean QueryExtension(char *extName)
{
   char *p = (char *) glGetString(GL_EXTENSIONS);
   char *end;
   if (p == NULL)
      return GL_FALSE;
   end = p + strlen(p);
   while (p < end) {
      int n = strcspn(p, " ");
      if ((strlen(extName)==n) && (strncmp(extName,p,n)==0)) {
         return GL_TRUE;
      }
      p += (n + 1);
   }
   return GL_FALSE;
}
```

Расширения стандарта для Microsoft Windows (WGL)

Если вам требуется доступ к функциям расширения для платформы Microsoft Windows, вы должны:

- создать код, отталкиваясь от символической константы, идентифицирующей расширение;
- сделать запрос ранее обсуждавшейся строки расширения в реальном времени;
- использовать команду `wglGetProcAddress()` для получения указателя на нужную функцию.

Листинг 14.3 демонстрирует эти действия для функции с условным именем `glSpecialEXT()`. Заметьте, что условный код (директива препроцессора `#ifdef`) проверяет присутствие определения `GL_EXT_special`. Если расширение доступно, тип данных (в нашем случае `PFNGLSPECIALEXTPROC`) действительно определяет указатель на функцию. Наконец, команда `wglGetProcAddress()` выдает собственно указатель на функцию.

Листинг 14.3. Локальное позиционирование в OpenGL-расширении с помощью wglGetProcAddress()

```
#ifdef GL_EXT_special
   PFNGLSPECIALEXTPROC glSpecialEXT;
#endif

/* где-нибудь позднее в коде */
#ifdef GL_EXT_special
   glSpecialEXT = (PFNGLSPECIALEXTPROC)
                  wglGetProcAddress("glSpecialEXT");
   assert(glSpecialEXT != NULL);
#endif
```

Полупрозрачность

Вы можете использовать штриховку многоугольников для имитации такой стильной полупрозрачности материала. Это особенно хорошо в решениях для систем, в которых нет смешивания на аппаратном уровне. Поскольку шаблоны штриховки многоугольников имеют размеры 32 × 32 бита, то есть 1024 бита, можно получить до 1023 градаций прозрачности. (На практике такое большое количество совсем не требуется.) Например, если вы хотите моделировать поверхность, пропускающую 29 % освещения, просто сформируйте шаблон штриховки, в котором 29 % пикселов (примерно 297) маскированы нулем и остальные — единицами. Даже если ваши поверхности должны быть одинаково прозрачны, не используйте один шаблон на всех, потому что они покрывают собой одну и ту же часть экрана. Сделайте собственный шаблон для каждой поверхности, используя для маски случайно выбранные номера пикселов. (См. раздел «Отображение точек, линий и многоугольников» в главе 2 для получения подробной информации о штриховке многоугольников.)

Если вас не удовлетворяет эффект от случайного выбора пикселов, используйте регулярные шаблоны, но они не работают на должном уровне при складывании поверхностей в «стопку». Это часто не является проблемой, так как большинство сцен имеют относительно немного полупрозрачных областей с перекрытием. На картинке автомобиля с полупрозрачными окнами ваш взгляд может проникнуть сквозь два окна, но обычно хватает одного.

Простой эффект «растворения» изображения

Допустим, у вас есть изображение, которое нужно постепенно обесцветить до уровня фонового цвета. Определите серию масок штриховки многоугольников, для каждого из которых наращивайте количество единиц, чтобы они представляли все более прозрачные шаблоны. Затем примените шаблоны к большому многоугольнику, охватывающему интересующую вас область. Например, вы хотите свести изображение к черному за 16 шагов. Тогда сначала определите 16 различных массивов масок:

```
GLubyte stips[16][4*32];
```

Затем загрузите их таким образом, чтобы для каждого была «включена» одна шестнадцатая от всех пикселов маски с размерами 32 × 32 и чтобы поразрядное ИЛИ всех шаблонов штриховки в сумме давало единицу. После этого воспользуйтесь следующим приемом:

```
draw_the_picture();
glColor3f(0.0, 0.0, 0.0); /* установить черный цвет */
for (i = 0; i < 16; i++) {
   glPolygonStipple(&stips[i][0]);
   draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

В некоторых реализациях OpenGL вы можете добиться лучшей производительности, сначала скомпилировав шаблоны в списки отображения. Когда проводите инициализацию, выполните ее примерно так:

```
#define STIP_OFFSET 100
for (i = 0; i < 16; i++) {
   glNewList(i+STIP_OFFSET, GL_COMPILE);
   glPolygonStipple(&stips[i][0]);
   glEndList();
}
```

Затем замените строку первоначального фрагмента кода

```
glPolygonStipple(&stips[i][0]);
```

на

```
glCallList(i);
```

При компиляции набора команд штриховки в список отображения OpenGL способен переорганизовать данные в массиве *stips*[ ][ ] в удобную для аппаратного обеспечения форму, позволяющую достичь максимальной скорости штриховки.

Другое место приложения этой техники — случай, когда вы рисуете динамическую картинку и хотите, чтобы изображение оставляло за собой размытый, плавно пропадающий след для создания иллюзии движения. Например, пусть вы мо-

делируете планетарную систему и хотите оставлять следы планет, показывая тем самым траектории их движения. Снова будем считать, что след «растворяется» за 16 шагов, и зададим те же шаблоны штриховки (скажем, используя вариант со списками отображения). В результате главный цикл нашей модели будет выглядеть примерно так:

```
current_stipple = 0;
while (1) { /* цикл "навсегда" */
   draw_the_next_frame();
   glCallList(current_stipple++);
   if (current_stipple == 16) current_stipple = 0;
   glColor3f(0.0, 0.0, 0.0); /* установить черный цвет */
   draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

На каждом проходе цикла вы очищаете одну шестнадцатую пикселов. На всех шестнадцати кадрах любой пиксел, не попадающий в позицию планеты, очищается в черный цвет вакуума. Конечно, если вам доступно смешивание на аппаратном уровне, легче в каждом кадре добавлять некоторое количество фонового цвета. (См. раздел «Отображение точек, линий и многоугольников» главы 2 по поводу штриховки многоугольников, главу 7 для получения информации о списках отображения и раздел «Смешивание» главы 6.)

Использование фонового буфера для выбора объектов

Хотя механизм выбора OpenGL (см. раздел «Выбор» главы 13) — вещь мощная и гибкая, в использовании он несколько громоздок. Обычные ситуации достаточно просты: приложение рисует сцену, составленную из «осязаемого» количества объектов; пользователь указывает на объект мышью, а от приложения требуется идентифицировать объект под острием указателя.

Другой путь требует, чтобы приложение запускалось в режиме двойной буферизации. Когда пользователь щелкает на объекте, приложение перерисовывает всю сцену в фоновый буфер, но вместо нормальных цветов объектов используется код самого объекта. Затем приложение просто читает пикселы под курсором, и значения пикселов кодируются номером выбранного объекта. Если для одного статичного изображения ожидается много операций выбора, вы можете прочитать весь буфер цвета единожды и просматривать эту копию при каждой попытке выбора, что удобнее индивидуального чтения для каждого пиксела.

Заметьте, что эта схема выигрывает перед стандартным выбором в том, что выбираются объекты, появляющиеся на переднем крае из «стопки», покрывающей один пиксел. Поскольку изображение с «фальшивыми» цветами рисуется в задний буфер, пользователь не подозревает о «подделке»; вы можете перерисовать задний буфер (или его копию в передний буфер) перед перестановкой буферов местами. В режиме индексации цвета кодирование проще: идентификатор объекта высту-

пает в роли индекса. В режиме RGBA биты для идентификации кодируются в R-, G- и B-компоненты.

Убедитесь, что схема предусматривает достаточно идентификаторов в случае, когда сцена переполнена объектами. Например, пусть вы запускаете приложение в режиме индексации цвета в системе с 4-битовыми буферами для информации об индексе цвета (16 возможных индексов), а сцена включает в себя тысячи потенциальных объектов выбора. Чтобы адресовать их все, может потребоваться несколько проходов. Чтобы рассуждать об этом в конкретных терминах, предположим, что имеется менее 4096 элементов, поэтому для кодирования всех объектов достаточно 12 бит. На первом проходе сцена рисуется с использованием индексов, образованных четырьмя старшими битами, для рисования на втором и третьем проходах привлекаются средние и младшие четыре бита соответственно. После каждого прохода читается пиксел под курсором, его биты извлекаются и пакуются вместе с другими для получения идентификатора объекта.

В таком варианте подготовка к выбору сокращается в три раза по времени, но это возможно лишь иногда. Обратите внимание, что после того, как вы разобрались с четырьмя старшими битами, вы тем самым исключили из рассмотрения 15/16 от всех объектов, и реально вам нужно перерисовать 1/16 их на втором проходе. Аналогично, после второго прохода исключаются 255 из 256 возможных элементов. Таким образом, первый проход длится столько, сколько нужно для рисования одного кадра, а второй и третий могут быть быстрее в 16 и 256 раз соответственно.

Если вы стремитесь писать переносимый на различные системы код, разбивайте идентификаторы ваших объектов на порции, вписывающиеся в наименьший общий знаменатель всех систем. Также имейте в виду, что ваша система может автоматически генерировать полутона в режиме RGB. Тогда отключите эту искусственную передачу полутонов.

Простая трансформация изображения

При необходимости деформировать растровое изображение (возможно, просто растянуть или повернуть или даже радикально модифицировать некоторой математической функцией) сделать это можно различными способами. Вы можете использовать изображение как карту текстуры, позволяющую вам масштабировать, поворачивать или как-то иначе деформировать изображение. Если вас интересует просто масштабирование, используйте `glPixelZoom()`.

Во многих случаях вы можете добиться хороших результатов, если будете рисовать каждый пиксел изображения в виде четырехугольника. Хотя эта схема не создает изображения, выглядящие настолько привлекательно, какими они выходят из-под алгоритма фильтрации (вряд ли они удовлетворят тонкий вкус), она значительно быстрее.

Переходя к конкретным примерам, предположим, что оригинальное изображение имеет размеры $m \times n$ пикселов и координаты $[0, m-1] \times [0, n-1]$. Зададим функ-

ции трансформации $x(m, n)$ и $y(m, n)$. Например, если трансформация представляет собой обычное масштабирование с коэффициентом 3,2, то $x(m, n) = 3{,}2 \times m$ и $y(m, n) = 3{,}2 \times n$. Следующий код отображает деформированное изображение:

```
glShadeModel(GL_FLAT);
glScale(3.2, 3.2, 1.0);
for (j=0; j < n; j++) {
   glBegin(GL_QUAD_STRIP);
   for (i=0; i <= m; i++) {
      glVertex2i(i,j);
      glVertex2i(i, j+1);
      set_color(i,j);
   }
glEnd();
}
```

Этот код рисует каждый трансформированный пиксел сплошным цветом, равным цвету пиксела, и применяет к изображению масштабирование с коэффициентом 3,2. Процедура `set_color()` подменяет собой соответствующую команду OpenGL, устанавливающую цвет пиксела изображения.

Ниже представлена слегка усложненная версия для деформации изображения с использованием функций $x(i, j)$ и $y(i, j)$:

```
glShadeModel(GL_FLAT);
for (j=0; j < n; j++) {
   glBegin(GL_QUAD_STRIP);
   for (i=0; i <= m; i++) {
      glVertex2i(x(i,j), y(i,j));
      glVertex2i(x(i,j+1), y(i,j+1));
      set_color(i,j);
   }
   glEnd();
}
```

Более симпатично изображение можно трансформировать с помощью следующего кода:

```
glShadeModel(GL_SMOOTH);
for (j=0; j < (n - 1); j++) {
   glBegin(GL_QUAD_STRIP);
   for (i=0; i < m; i++) {
      set_color(i,j);
      glVertex2i(x(i,j), y(i,j));
      set_color(i,j+1);
      glVertex2i(x(i,j+1), y(i,j+1));
   }
   glEnd();
}
```

Этот код плавно интерполирует цвет в пределах каждого четырехугольника. Заметьте, что данная версия вырабатывает четырехугольники в количестве, на единицу меньшем в каждом измерении, чем это делает «плоская» модель, поскольку цветное изображение используется для указания цветов вершин четырехугольника. Кроме того, вы можете сгладить многоугольники с помощью соответствую-

щей функции смешивания (GL_SRC_ALPHA, GL_ONE), чтобы получить более привлекательное изображение.

Послойное отображение

Для отдельных приложений, таких как программы для разметки печатных плат, вам может понадобиться отобразить множество различных слоев (материалов) и описать характер их наложения друг на друга.

Простой пример. Предположим, что у вас есть три различных вещества, которые можно представить послойно. В любой точке допустимы восемь комбинаций слоев, согласно табл. 14.2.

Таблица 14.2. Восемь комбинаций слоев

	Слой 1	Слой 2	Слой 3	Цвет
0	Отсутствует	Отсутствует	Отсутствует	Черный
1	Присутствует	Отсутствует	Отсутствует	Красный
2	Отсутствует	Присутствует	Отсутствует	Зеленый
3	Присутствует	Присутствует	Отсутствует	Синий
4	Отсутствует	Отсутствует	Присутствует	Розовый
5	Присутствует	Отсутствует	Присутствует	Желтый
6	Отсутствует	Присутствует	Присутствует	Белый
7	Присутствует	Присутствует	Присутствует	Серый

Вы хотите, чтобы программа отображала восемь различных цветов, в зависимости от наличия того или иного слоя аналогично тому, как это показано в последнем столбце таблицы. Для этого используйте режим индексации цвета и загрузите свою карту цветов так, чтобы элемент 0 представлял черный цвет, элемент 1 — красный, элемент 2 — зеленый и т. д. Заметьте, что когда числа от 0 до 7 записываются двоичным кодом, четвертый бит устанавливается при каждом появлении третьего слоя, второй — при появлении второго слоя и первый бит — для слоя 1.

Чтобы очистить окно, установите маску записи в 7 (все три слоя) и цвет очистки — в 0. Чтобы нарисовать свое изображение, установите цвет в 7, а когда вы захотите нарисовать что-нибудь в слое *n*, задайте маску равной *n*. Для других типов приложений может потребоваться выборочное стирание в слое, тогда следует использовать маску в соответствии с изложенным, но установить для значения цвета 0 вместо 7. (См. раздел «Маскирование буферов» главы 10 для получения более подробной информации о масках записи.)

Сглаживание символов

Стандартная схема рисования символов с помощью `glBitmap()` подразумевает в отношении каждого пиксела подход «все или ничего» — пиксел либо «вклю-

чен», либо «выключен». Если вы выводите черные символы на белом фоне, результирующие пикселы будут такими же черными или белыми, без какой-либо тонкой разницы в тонах. Но можно добиться и сглаженных с высоким качеством изображений, если использовать при отображении символов промежуточные оттенки (в нашем примере серые).

Будем исходить из того, что вы все-таки рисуете черным по белому. Представьте сильно увеличенную, как под лупой, картинку с пикселами экрана, на которую наложен символ векторного шрифта, как показано слева на рис. 14.1.

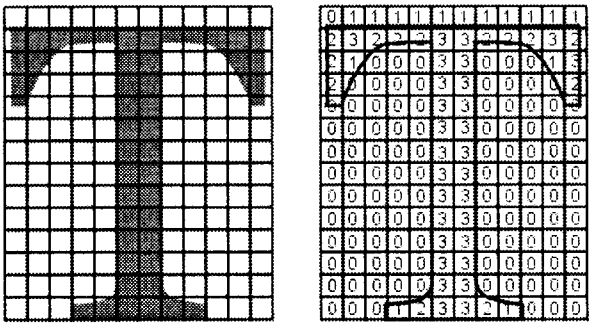


Рис. 14.1. Сглаженные символы

Обратите внимание, что в векторный контур попала какая-то часть полных пикселов, и их следует нарисовать черным. Другие пикселы находятся полностью за границами контура и должны быть белыми. Но остальным пикселам разумно присвоить градации серого, в зависимости от доли черного цвета в пикселе. Данный подход позволяет сделать результирующее изображение не таким контрастным и, соответственно, более приятным для зрения.

Если скорость и расход памяти для вас не критичны, каждый символ можно оформить как маленькое изображение, а не просто как битовый образ. Однако при использовании режима RGBA этот метод может потребовать 32 бита на пиксел хранимого и отображаемого символа, вместо одного бита для обычного битового образа. Как альтернативу, можно предложить использовать один 8-битовый на пиксел и преобразовывать эти индексы в RGBA таблично. Во многих случаях компромисс в том, чтобы позволить рисовать символ лишь несколькими уровнями серого, скажем, двумя-тремя, тогда символ шрифта будет расходовать только 2 или 3 бита памяти на пиксел.

Числа в правой части рис. 14.1 задают примерную относительную долю черного в каждом пикселе: 0 соответствует пустому квадрату сетки, 1 — трети покрытия, 2 — двум третям и 3 — означает полностью черный квадрат. При этом для поддержания четырех возможных уровней серого, 0, 1, 2 и 3, требуется только 2 бита.

В зависимости от цветового режима — RGBA или индексации цвета — вы можете использовать один из двух методов, реализующих сглаживание символов.

В режиме RGBA определите три различных изображения символа, соответствующих уровням 1, 2 и 3 на рис. 14.1. Установите белый цвет и очистите фон.

Установите цвет в две трети интенсивности серого (RGB = (0.666, 0.666, 0.666)) и нарисуйте все пикселы уровня 1. Затем установите значение RGB = (0.333, 0.333, 0.333), нарисуйте второй образ и используйте RGB = (0.0, 0.0, 0.0) для третьего образа. Теперь у вас есть три различных шрифта, и для того чтобы вывести строку текста, на каждом из трех проходов нужно заполнить те биты изображения, которые соответствуют плотности цвета.

В режиме индексации цвета принцип тот же самый, но достаточно двух образов и двух проходов. Установите для одного образа единицы там, где в символе появляются значения 1 или 3. Для второго образа установите единицы в местах значений 2 или 3. Загрузите карту цвета так, чтобы 0 давал белый цвет, 1 — светло-серый, 2 — темно-серый и 3 — черный. Установите индекс цвета 3 (двоичное значение 11) и маску записи в 1 и затем нарисуйте первый образ. Затем измените маску на 2 и нарисуйте второй. Там, где на рис. 14.1 появляется 0, в буфере кадров не будет ничего. Там, где встречаются 1, 2 и 3, те же значения — 1, 2 и 3 попадут в буфер кадров.

Мы немного сэкономили в последнем примере для четырех градаций серого, получив два прохода вместо трех. Если бы мы взяли восемь оттенков, RGBA-метод потребовал бы семь проходов, а техника маскирования карты цветов — только три. При 16 уровнях серого это соотношение выглядело бы как 15 к 4. (См. раздел «Маскирование буферов» главы 10 для получения большей информации о масках записи и раздел «Битовые образы и шрифты» главы 8 по вопросам рисования битовых образов.)

Упражнения

Можно ли представить такое решение, чтобы при визуализации в режиме RGBA использовать не больше изображений, чем в оптимизированном варианте с индексами цвета?

Подсказка: как фрагменты RGB обычно объединяются (merge) в буфере цвета, когда желательно сглаживание?

Рисование округлых точек

Точки, имеющие округлую форму, рисуются с помощью включенного сглаживания, выключенного смешивания и с помощью альфа-функции, которая пропускает только фрагменты со значением альфа-компонента, большим 0.5. (См. разделы «Сглаживание» и «Смешивание» главы 6.)

Интерполяция изображений

Предположим, что у вас есть пара изображений (под *изображением* понимается растровое изображение или картинка, сгенерированная с использованием векторных данных обычным способом), и вы хотите плавно смешать одно с другим. Такая «диффузия» может быть выполнена легко с использованием альфа-компо-

нента и соответствующих операций смешивания. «Перетекание» должно пройти в десять шагов, где изображение A показывается в кадре 0, а изображение B — в кадре 9. В глаза бросается следующее решение: в i-м кадре рисовать изображение A со значением альфа-компонента $(9 - i)/9$ и изображение B — со значением $i/9$.

Задача: нарисовать оба изображения в одном кадре. Быстрое решение это нарисовать изображение A в кадре 0. Чтобы получить кадр 1, смешайте 1/9 изображения B и 8/9 его текущего содержимого. Для кадра 2 возьмите 1/8 от изображения B и 7/8 содержимого кадра. Для кадра 3 — 1/7 от B и 6/7 из кадра и т. д. На последнем шаге вы должны нарисовать 1/1 изображения B, смешанного с долей 0/1 того, что имеется, а 1/1 B + 0 и есть изображение B. Вот как это работает.

Если для кадра i вы имеете

$$\frac{(9-i)\mathrm{A}}{9}+\frac{i\mathrm{B}}{9}$$

и смешиваете $\mathrm{B}/(9-i)$ в пропорции с долей $(8-i)/(9-i)$ того, что есть, вы получите

$$\frac{\mathrm{B}}{9-i}+\frac{8-i}{9-i}\left[\frac{(9-i)\mathrm{A}}{9}+\frac{i\mathrm{B}}{9}\right]=\frac{9-(i+1)\mathrm{A}}{9}+\frac{(i+1)\mathrm{B}}{9}.$$

(См. раздел «Смешивание» в главе 6.)

Создание устойчивых изображений

Допустим, что, рисуя сложную трехмерную картинку, вы прибегаете к буферизации глубины с целью удаления невидимых поверхностей. Также предположим, что одна часть картинки составлена из копланарных фигур A и B, где фигура B — своего рода деколь[1], которая всегда должна появляться поверх A.

Первое, что приходит в голову, — нарисовать фигуру B после рисования A, установив функцию буферизации глубины для замещения фрагментов, глубина которых больше значения, сохраненного в буфере глубины (GL_GEQUAL), или равна ему. Но, ввиду ограниченной точности представления с плавающей запятой для вершин, ошибка округления может сместить фигуру B немного вперед или назад относительно фигуры A. Эта проблема решается единственным образом:

1) отключите буфер глубины для записи и выполните визуализацию A;

2) включите буфер глубины для записи и выполните визуализацию B;

3) заблокируйте буфер цвета для записи и отобразите A снова;

4) разблокируйте буфер цвета для записи.

[1] Деколь — высокостойкое изображение на стекле, керамике и эмали, полученное путем высокотемпературного обжига. — *Примеч. перев.*

Заметьте, что во время этого процесса тест глубины отключен. На шаге 1 фигура A рисуется без ограничений, а значения буфера глубины не изменяются. Таким образом, на шаге 2, где B появляется поверх A, отображение B гарантируется. На шаге 3 вы просто убеждаетесь, что все значения глубины под A обновлены корректно, но, поскольку запись RGBA-данных заблокирована, цвет пикселов не принимается во внимание. Наконец, шаг 4 возвращает систему в состояние по умолчанию (запись включена для обоих буферов — глубины и цвета).

Если доступен буфер трафарета, можно использовать более простую методику:

1. Сконфигурируйте буфер трафарета для записи единиц при успехе теста глубины и записи нулей в противном случае. Выполните визуализацию A.
2. Сконфигурируйте буфер трафарета так, чтобы не изменять значения маски, а выполнять визуализацию лишь там, где значения маски равны 1. Отключите тест глубины и обновление буфера глубины. Выполните визуализацию B.

В таком варианте нет необходимости в инициализации содержимого буфера трафарета когда бы то ни было, так как значения маски для всех пикселов, представляющих интерес (то есть полученных при визуализации A), устанавливаются, когда выполняется визуализация фигуры A. Убедитесь, что заново включили тест глубины и отключили тест трафарета перед отображением дополнительных многоугольников. (См. разделы «Выбор буферов цвета для записи и чтения», «Тест глубины» и «Тест трафарета» главы 10.)

Рисование закрашенных вогнутых многоугольников с помощью буфера трафарета

Рассмотрим вогнутый многоугольник «1234567», показанный на рис. 14.2. Представьте, что он рисуется как серия треугольников: 123, 134, 145, 156 и 167, которые все показаны на рисунке. Толстая линия представляет границы оригинального многоугольника. Рисование всех этих треугольников делит буфер на девять областей: A, B, C, . . . , I, где I — это внешняя по отношению ко всем треугольникам область.

В «легенде» рисунка имена областей сопровождены списком треугольников, в них попадающих. Области A, D и F формируют оригинальный многоугольник; заметьте, что эти три области покрыты нечетным количеством треугольников. Для всех прочих областей количество треугольников четное (возможно, нулевое). Таким образом, чтобы выполнить визуализацию внутренней части многоугольника, достаточно сделать его для областей, охватывающих четное количество треугольников. Это может быть реализовано с использованием буфера трафарета, посредством двухпроходного алгоритма.

Сначала очистите буфер трафарета и отключите запись в буфер цвета. Далее нарисуйте каждый из треугольников в порядке очереди, используя функцию GL_INVERT для буфера трафарета. (Для лучшей производительности используйте «веера» треугольников.) Это зеркально (поразрядно) инвертирует значения всякий раз, когда рисуется треугольник, покрывающий пиксел. После того как все тре-

угольники будут нарисованы, если пиксел покрыт четное количество раз, значение в буфере трафарета будет нулевое; в противном случае — ненулевое. Наконец, нарисуйте большой многоугольник поверх всей области (или перерисуйте треугольники), но позволяйте рисование только там, где значение маски отлично от нуля.

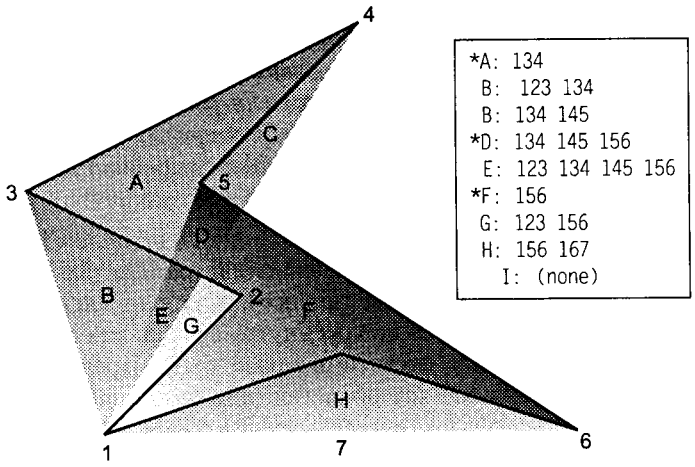


Рис. 14.2. Вогнутый многоугольник

ПРИМЕЧАНИЕ

Рассмотренную технику можно несколько обобщить в предположении, что вы не будете начинать рисование с вершины многоугольника. Пусть в примере «1234567» обозначение P соответствует любой точке — как снаружи, так и внутри многоугольника. Нарисуйте треугольники: P12, P23, P34, P45, P56, P67 и P71. Области, захватывающие четное количество треугольников, будут внутри многоугольника; другие области — снаружи. Это обобщение правила, гласящего, что если P оказывается принадлежащей одному из ребер многоугольника, один из треугольников будет «пустым».

Данная методика может быть применена для закраски также непростых многоугольников (самопересекающихся) и многоугольников с отверстиями. Следующий пример иллюстрирует, как можно обработать усложненный многоугольник с двумя областями: четырехсторонней и пятисторонней. Допустим, что имеются трех- и четырехстороннее отверстия (все равно в какой области). Назовем эти две области `abcd` и `efghi`, а отверстия — `jkl` и `mnop`. Под `z` будем понимать любую точку плоскости. Нарисуйте следующие треугольники:

```
zab zbc zcd zda zef zfg zgh zhi zie zjk zkl zlj zmn zno zop zpm
```

Пометьте области, охватывающие нечетное количество треугольников, как *in*, и области, покрытые четным числом треугольников, как *out*. (См. раздел «Тест трафарета» главы 10 для получения более подробной информации о буфере трафарета.)

Нахождение перекрывающихся областей

Разработчикам механических конструкций, собранных из небольших трехмерных деталей, может потребоваться отобразить области, где эти детали перекрывают друг друга. Во многих случаях такие области указывают на конструкторские ошибки. Если это движущиеся части, данный процесс может оказаться еще более значимым, поскольку поиск таких перекрытий может быть проведен сквозь весь цикл разработки. Метод, позволяющий делать такие вещи, сложен, а описания, приведенного ниже, может оказаться недостаточно для его понимания. Все подробности вы найдете в публикации *Interactive Inspection of Solids: Cross-sections and Interferences* Ярека Россиньяка (Jarek Rossignac), Эйба Мегахеда (Abe Megahed) и Бенгта-Олафа Шнейдера (Bengt-Olaf Schneider) (труды конференции SIGGRAPH, 1992).

Этот метод связан с алгоритмом кэпинга, описанным в разделе «Тест трафарета» главы 10. Идея в том, чтобы рассечь произвольной плоскостью объекты, подлежащие тесту на взаимные помехи (interference), и затем выяснить, какая часть плоскости отсечения одновременно попала внутрь нескольких объектов. Для статичных изображений плоскость отсечения может быть передвинута вручную с целью подсветки перекрывающихся областей; для динамических изображений может оказаться проще использовать сетку таких плоскостей, чтобы обнаружить все невыявленные перекрытия.

Нарисуйте объекты, подлежащие проверке, и проведите через них плоскость отсечения снова. Отметьте, какие пикселы внутри объекта попали на плоскость отсечения, опираясь на статистику «чет-нечет» для буфера трафарета, как объяснялось в предыдущем разделе. (Для объектов, образованных корректно, точка находится внутри объекта, если луч, проведенный из нее к глазам наблюдателя, пересекает поверхности объекта нечетное количество раз, и снаружи его — при четном числе пересечений.) Чтобы обнаружить «взаимные помехи», вам следует найти те пикселы в буфере кадров, для которых плоскость отсечения находится внутри двух и более областей сразу; другими словами, в пересечении внутренних областей какой-то пары объектов.

Если нужно проверить на взаимное пересечение множество объектов, фиксируйте один бит всякий раз, когда обнаруживаете пересечение, и другой бит — для образованного внутренними областями всех объектов буфера отсечения (объединения (union) внутренних областей объектов). Для каждого нового объекта вычислите его внутреннюю область, найдите ее пересечение с объединением внутренних областей уже проверенных объектов и проследите точки пересечения. Затем присоедините внутренние точки нового объекта к объединению.

Операции, описанные в предыдущем абзаце, можно выполнять с использованием различных битов буфера трафарета и различных операций маскирования. Вам требуется три бита буфера трафарета на пиксел: один — для переключения к вычислениям внутренней области объекта, один — для объединения всех уже обследованных внутренних областей и один — для областей, в которых имело место пересечение. Попробуем конкретизировать сказанное на примере, предположим,

что 1-й бит буфера трафарета служит для переключения внутренней и внешней областей, 2-й бит выполняет объединение и 4-й бит предназначен для регистрации пересечения. Для каждого объекта, отображаемого на экране, сбросьте 1-й бит (используя маску), затем переключите 1-й бит, удерживая единичную маску, и используйте операцию GL_INVERT.

Вы можете найти пересечения и объединения битов в буферах трафарета, используя операции шаблона. Например, чтобы сделать биты в буфере 2 объединением битов из буферов 1 и 2, замаскируйте шаблон этими двумя битами и нарисуйте что-нибудь поверх всего объекта с функцией шаблона, пропускающей ненулевые значения. Такие получаются, если биты в буфере 1, в буфере 2 или в обоих установлены. При успехе сравнения запишите 1 в буфер 2. Также убедитесь, что рисование в буфер цвета отключено. Вычисление пересечения проводится аналогично — установите функцию, пропускающую только совокупные по двум буферам значения, равные 3 (биты установлены в обоих буферах — 1 и 2). Запишите результат в надлежащий буфер. (См. раздел «Тест трафарета» главы 10.)

Тени

Каждая возможная проекция трехмерного пространства в трехмерное пространство может быть выполнена с использованием подходящей обратимой матрицы 4 × 4 и однородных координат. Если матрица не является обратимой, но имеет ранг 3, это проецирует трехмерное пространство на двухмерную плоскость. Каждая такая возможная проекция может быть получена с помощью подходящей 4 × 4 матрицы с рангом 3. Чтобы найти тень от произвольного объекта на произвольной плоскости от произвольного источника освещения (возможно, размещенного в бесконечности), вам следует найти матрицу, представляющую эту проекцию, умножить ее на матрицу в стеке и нарисовать объект цветом тени. Имейте в виду, что вы должны выполнять проекцию на каждую плоскость, которая называется «основанием» (ground).

В качестве простой иллюстрации предположим, что свет исходит из начала координат, а уравнение плоскости основания — $ax + by + cz + d = 0$. Для данной вершины $S = (sx, sy, sz, 1)$ линия от источника освещения до S охватывает все точки αS, где α — произвольное вещественное число. Эта линия пересекает плоскость в некоторой точке, согласно

$$\alpha\ (a \times sz + b \times sy + c \times sz) + d = 0.$$

Отсюда

$$\alpha = -\ d/(a \times sz + b \times sy + c \times sz).$$

Применив это выражение для линии, получим

$$-d\ (sx, sy, sz)/\ (a \times sz + b \times sy + c \times sz)$$

для точки пересечения.

Матрица, отображающая произвольные значения S в эту точку, такова:

$$\begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & -d & 0 \\ a & b & c & 0 \end{bmatrix}.$$

Эта матрица может использоваться, если вы сначала выполните преобразования координат так, чтобы источник света попал в начало координат.

Если источник освещения находится в бесконечности, все, что у вас есть, — это точка S и направление D = (*dx*, *dy*, *dz*). Точки на линии задаются выражением:

$S + \alpha D$.

Как и ранее, пересечение линии с плоскостью:

$$a\,(sx + \alpha \times dx) + b(sy + \alpha \times dy) + c(sz + \alpha \times dz) + d = 0.$$

Решение для α, подставленное в уравнение для линии, а затем определение матрицы проекции дают

$$\begin{bmatrix} b\times dy + c\times dz & -b\times dx & -c\times dx & -d\times dx \\ -a\times dy & a\times dx + c\times dz & -c\times dy & -d\times dy \\ -a\times dz & -b\times dz & a\times dx + b\times dy & -d\times dz \\ 0 & 0 & 0 & a\times dx + b\times dy + c\times dz \end{bmatrix}.$$

Последняя матрица работает для заданных плоскости и произвольного направления вектора. Вам не нужно делать какие-либо предварительные преобразования. (См. главу 3 и приложение E.)

Удаление невидимых линий

Если вам нужно нарисовать каркасный объект с удаленными невидимыми линиями, один подход заключается в рисовании контуров линиями и заполнении внутренних областей многоугольников, образующих поверхность, многоугольниками с цветом фона. При включении буферизации глубины эта внутренняя область заслоняет любые линии контура, которые должны быть загорожены гранями, более близкими к глазам наблюдателя. Этот метод будет работать, за исключением того, что не дает гарантии, что внутренняя область объекта полностью попадет внутрь контура многоугольника; фактически перекрытие может быть в различных местах.

Есть более простое, двухпроходное решение, где используется смещение многоугольника или буфер трафарета. Отконение многоугольника по глубине — обычно более предпочтительная техника, так как она практически всегда быстрее использования буфера трафарета. Оба метода описаны ниже, и вы можете сравнить между собой эти способы решения проблемы.

Удаление невидимых линий со смещением многоугольника

Чтобы использовать смещение многоугольника для удаления невидимых линий, нарисуйте объект дважды. Подсвеченные ребра рисуются цветом переднего плана, с использованием «закрашенных» полигонов (режим GL_LINE), с целью растеризовать многоугольники как ребра каркаса. Затем рисуются закрашенные многоугольники в режиме по умолчанию, в котором заполняется внутренняя область каркаса, и со смещением многоугольника, чтобы чуть-чуть оттолкнуть закрашенные многоугольники в направлении от наблюдателя. При этом внутренняя область отступает ровно настолько, чтобы подсвеченные ребра рисовались без неприятных визуальных дефектов.

```
glEnable(GL_DEPTH_TEST);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
set_color(foreground);
draw_object_with_filled_polygons();

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 1.0);
set_color(background);
draw_object_with_filled_polygons();
glDisable(GL_POLYGON_OFFSET_FILL);
```

Величину смещения можно регулировать (например, для широких линий). (См. раздел «Отклонение многоугольников по глубине» главы 6.)

Удаление невидимых линий с помощью буфера трафарета

Применение буфера трафарета для удаления невидимых линий — более сложная процедура. Для каждого многоугольника вы должны очистить буфер трафарета и нарисовать контур как в буфер кадров, так и в буфер трафарета. Затем, при заполнении внутренней области, разрешите рисование только там, где буфер трафарета еще чист. Простейший путь очистки буфера трафарета, во избежание очистки всего его для каждого многоугольника, — просто выводить нули в буфер, используя тот же контур многоугольника. Здесь вам достаточно очистить весь буфер трафарета только единожды.

Следующий код представляет цикл, пригодный для удаления невидимых линий. Каждый многоугольник рисуется контуром с цветом переднего плана, закрашивается цветом фона, и затем снова проводятся контуры с цветом переднего плана. Буфер трафарета используется для защиты от перезаписи закрашенной части контуром многоугольника. Чтобы оптимизировать производительность, параметры маски и цвета изменяются только дважды на цикл; оба раза, когда рисуется контур, это одни и те же значения.

```
glEnable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
```

```
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 1);
glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
set_color(foreground);
for (i=0; i < max; i++) {
   outline_polygon(i);
   set_color(background);
   glStencilFunc(GL_EQUAL, 0, 1);
   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
   fill_polygon(i);
   set_color(foreground);
   glStencilFunc(GL_ALWAYS, 0, 1);
   glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
   outline_polygon(i);
}
```

См. раздел «Тест трафарета» главы 10.

Текстурирование

Наложение текстур — мощная техника, и она используется во многих интересных областях. Ниже описано, как можно расширить ее применение:

- **Текст с устраненной ступенчатостью.** Для каждого символа определите карту текстуры относительно высокого разрешения, а затем отображайте такие карты на небольшие области, задействуя фильтрацию, предоставляемую текстурированием. Это также делает текст корректно выглядящим на поверхностях, которые повернуты относительно плоскости экрана и имеют некоторое искажение перспективы.
- **Сглаженные линии.** Как и в случае с текстом: образуйте линию текстуры шириной в несколько пикселов и используйте фильтрацию для сглаживания линий.
- **Масштабирование и вращение изображений.** Если вы в отображение текстуры поместите изображение и используете эту текстуру для многоугольника, вращение и масштабирование многоугольника будет эффективнее вращения и масштабирования изображения.
- **Деформирование изображений.** Сохраните изображение как карту текстуры, но отобразите ее на некую представленную сплайном поверхность (используя вычислители). Деформируя (warping) поверхность, вы будете деформировать изображение.
- **Проецирование изображений.** Поместите изображение в карту текстуры и спроецируйте ее в виде направленного света, создавая эффект слайд-проектора. (См. раздел «Координата q» главы 9 по вопросу, как моделировать прожектор с помощью текстур.)

(См. главу 3 для получения информации о вращении и масштабировании, главу 9 — о создании текстур и главу 12 — о вычислителях.)

Рисование изображений с буферизацией глубины

Для сложных статичных фоновых изображений время на их построение по геометрическому описанию может быть больше времени, затрачиваемого на рисование их в виде растрового изображения. Если имеется фиксированный фон и относительно просто изменяющийся передний план, вы можете захотеть отобразить фон и его версии с буферизацией глубины как растровое изображение, а не как воспроизведенное по геометрическим параметрам. Фон может также состоять из элементов, что непроизводительно расходует время, хотя их изображения в буферах кадра и глубины доступны. Вы можете выполнить визуализацию этих элементов в режиме с буферизацией глубины, применив двухпроходный алгоритм.

Например, если вы рисуете модель молекулы, собранную из сфер, можно использовать изображение сферы с превосходным качеством и связанные с ним значения буфера глубины, рассчитав их с помощью закрашивания Фонга или трассировки лучей или иной техники, непосредственно недоступной в OpenGL. Для комплексной модели могут потребоваться тысячи таких сфер, причем буферизованных по глубине.

Чтобы добавить буферизованное по глубине изображение в сцену, сначала выведите значения глубины изображения в буфер глубины командой `glDrawPixels()`. Затем включите буферизацию глубины, сбросьте маску записи в 0, чтобы ничего не рисовать, и включите обработку по маске, чтобы запись в буфер глубины сопровождалась записью в буфер трафарета.

Затем выведите изображение в буфер цвета, с учетом только что записанного буфера трафарета так, чтобы запись происходила при значении буфера трафарета, равного 1. Во время этой записи установите функцию обработки по маске в обнуление буфера трафарета так, чтобы он автоматически очищался к моменту добавления следующего изображения в сцену. Если объекты нужно передвинуть поближе к наблюдателю или подальше от него, используйте ортогональную проекцию; в этих случаях вы используете параметр `GL_DEPTH_BIAS` совместно с `glPixelTransfer*()` для перемещения изображения по глубине. (См. раздел «Управление системой координат» главы 2, «Тест глубины» и «Тест трафарета» главы 10, и главу 8 для получения информации об использовании `glDrawPixels()` и `glPixelTransfer*()`.)

Области Дирихле

Пусть S — множество точек плоскости, тогда область Дирихле (или многоугольник Вороного) одной из этих точек — это множество всех точек плоскости, более близких к данной точке, чем любые другие точки множества S. Эти точки позволяют решать многие проблемы вычислительной геометрии. На рис. 14.3 показан пример с контурами областей Дирихле.

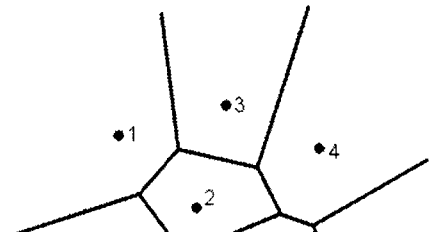


Рис. 14.3. Области Дирихле

Если для каждой точки S вы нарисуете буферизованный по глубине конус с вершкой в этой точке и с цветом, отличным от цвета других S-точек, область Дирихле будет нарисована этим цветом. Простейший путь сделать это — предварительно обсчитать глубину конуса в изображение и использовать изображение как значения буфера глубины, согласно описанию в предыдущем разделе. Здесь вам не нужно рисовать изображение в буфер кадров, как в случае закрашенных сфер. Пока вы рисуете в буфер глубины, используйте буфер трафарета для фиксирования пикселов, которые следует вывести при его первой очистке, а затем для записи ненулевых значений, прошедших тест глубины. Чтобы нарисовать область Дирихле, нарисуйте многоугольник поверх всего окна, но включите рисование только там, где значения буфера трафарета не нулевые.

Вы можете сделать это проще, отображая равномерно окрашенные конусы с простым буфером глубины, но хороший конус может потребовать для себя тысячи многоугольников. Техника, описанная в этом разделе, в состоянии выполнять визуализацию конусов высокого качества значительно быстрее. (См. разделы «Удаление невидимых поверхностей» главы 5 и «Тест глубины» главы 10.)

«Жизнь» в буфере трафарета

Игра «Жизнь», придуманная Джоном Конвеем (John Conway), играется на прямоугольной сетке, где каждая позиция сетки имеет статус «живой» или «мертвый». Чтобы вычислить следующее поколение на основе текущего, подсчитывают количество живых соседей для каждой клетки (в восьми примыкающих ячейках). Ячейка жива в поколении $n + 1$, если она жила в поколении n и имела двух или трех живых соседей или если она была мертвой в поколении n и имела *точно* трех живых соседей. Во всех других случаях она считается мертвой для поколения $n + 1$. Эта игра генерирует некоторые удивительно интересные модели при различных начальных конфигурациях. (См. работу Мартина Гарднера (Martin

Gardner), «Mathematical Games» в *Scientific American*, номер 223/4, Октябрь, 1970, с. 120–123.) На рис. 14.4 показаны шесть поколений игры.

Один из способов реализации игры средствами OpenGL заключается в использовании многопроходного алгоритма. Сохраните данные в буфере цвета из расчета один пиксел на каждую ячейку сетки. Предположим, что черный (все нули) — это цвет фона, а цвет живого пиксела отличен от нуля. Инициализируйте буферы глубины и трафарета их очисткой в нуль, установите маску записи буферизации глубины в нуль и задайте функцию сравнения такой, чтобы она пропускала не равные себе значения. По ходу итераций считайте изображение с экрана, включите рисование в буфер глубины и задайте функцию трафарета так, чтобы она инкрементировалась при успехе сравнения и не изменяла буфер трафарета в противном случае. Отключите рисование в буфер цвета.

Далее нарисуйте изображение восемь раз, со смещением на один пиксел в каждом из направлений: вертикальном, горизонтальном и диагональном. Когда вы закончите делать это, буфер трафарета будет содержать счетчики живых соседей для каждого пиксела. Отключите рисование в буфер цвета, установите цвет в значение цвета живых ячеек и задайте функцию трафарета, позволяющую рисование, только если значение в буфере трафарета равно 3 (три живых соседа). Кроме того, если рисование происходит, уменьшите значение в буфере трафарета. Затем нарисуйте прямоугольник поверх всего изображения; это закрасит каждую ячейку, имеющую точно трех живых соседей, «живым» цветом.

В этом месте буфер трафарета содержит 0, 1, 2, 4, 5, 6, 7 и 8, и значения меньше 2 считаются корректными. Значения 0, 1, 4, 5, 6, 7 и 8 должны быть очищены в «мертвый» цвет. Установите функцию трафарета в режим рисования, если значение не равно 2, и обнулите значения маски в других случаях. Затем нарисуйте большой многоугольник «мертвым» цветом поверх всего изображения. Готово.

Для получения пригодной к демонстрации программы может потребоваться масштабировать сетку до размера, большего одного пиксела; на сетке вида «пиксел на ячейку» детали различить трудно. (См. разделы «Управление системой координат» главы 2, «Тест глубины» и «Тест трафарета» главы 10.)

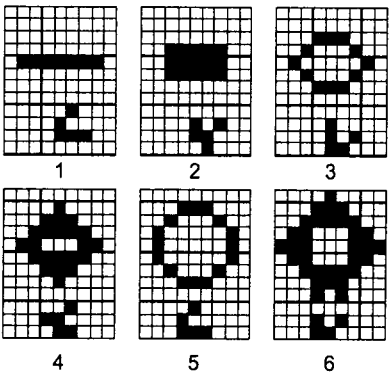


Рис. 14.4. Шесть поколений игры «Жизнь»

Альтернативное применение glDrawPixels() и glCopyPixels()

О команде `glDrawPixels()` обычно думают, как о способе нарисовать прямоугольную область пикселов на экране. Хотя чаще всего это именно так, есть и другие ее интересные применения:

- **Видео.** Даже если ваш компьютер не оснащен специальными аппаратными видеосредствами, вы можете показывать короткие фильмы, многократно рисуя кадры с помощью `glDrawPixels()` в одной и той же области заднего буфера и выполняя обмен содержимого буферов. Размер кадров, которые еще можно показывать с разумной производительностью таким способом, зависит от быстродействия вашего аппаратного обеспечения, поэтому имеет смысл ограничиться фильмами с размером кадра 100 × 100 пикселов (или меньше), если вы хотите иметь незаметную для глаза смену изображений.

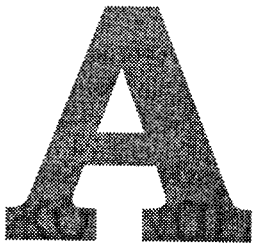
- **Аэрограф.** В программе рисования аэрограф (распылитель) может быть смоделирован с помощью значений альфа-компонента. Чтобы выполнить распыление с круговым отпечатком синего цвета, повторяйте рисование синего квадрата командой `glDrawPixels()`, изменяя при этом значения альфа-компонента от наибольшего в центре до вырождающихся в нуль на линии окружности, центр которой совпадает с центром квадрата. Рисуйте с помощью функции смешивания, предполагающей использование альфа-компонента и цвета имеющегося пиксела. Если значения альфа-компонента кисти много меньше единицы, для получения сплошного цвета потребуется выполнить многократное окрашивание области. Если они близки к 1, каждый мазок практически затирает цвета, на которые он приходится.

- **Масштабирование с фильтрацией.** Если вы масштабируете растровое изображение с нецелым коэффициентом, OpenGL эффективно использует рамочный фильтр, в результате чего сглаживание может оказаться грубым. Чтобы улучшить фильтрацию, выполните «дрожание» результирующего изображения — его перерисовку несколько раз со смещением меньше одного пиксела, используя альфа-смешивание для попиксельного усреднения. В результате вы получите масштабирование и фильтрацию одновременно.

- **Перестановка изображений.** Вы можете обменивать местами изображения одного размера при помощи команды `glCopyPixels()`, используя операцию исключающего ИЛИ (XOR). В таком случае можно исключить необходимость повторного чтения изображения в оперативную память. Если `A` и `B` — два изображения, операция выглядит следующим образом:
  - `A = A XOR B`;
  - `B = A XOR B`;
  - `A = A XOR B`.

Приложения

Порядок выполнения операций



В этой книге описаны все операции, выполняемые между спецификацией вершин и заключительной записью фрагментов в буфер кадров. Главы этой книги упорядочены так, чтобы облегчить изучение, а не в последовательности фактического выполнения операций. Иногда точный порядок операций не принципиален — например, поверхности можно преобразовать в многоугольники и затем трансформировать или наоборот с идентичными результатами, — и различные реализации OpenGL могут поступать в этом смысле по-разному.

Это приложение описывает возможный порядок; от любой реализации требуется, чтобы она вырабатывала однотипные результаты. За подробной информацией обращайтесь к руководству OpenGL Reference Manual.

Данное приложение разбито на следующие основные разделы:

- Общий обзор
- Векторные операции
- Операции над пикселами
- Операции над фрагментами
- Прочее

Общий обзор

Этот раздел дает общий обзор операций, как показано на рис. А1. Геометрические данные (вершины, линии и многоугольники) проходят на своем пути через отдельные блоки, такие как вычислители и операции над вершинами, а растровые данные (пикселы, изображения и битовые образы) обрабатываются по-другому. Оба типа данных подвергаются растеризации и операциям над фрагментами перед тем, как конечные растровые данные попадут в буфер кадров.

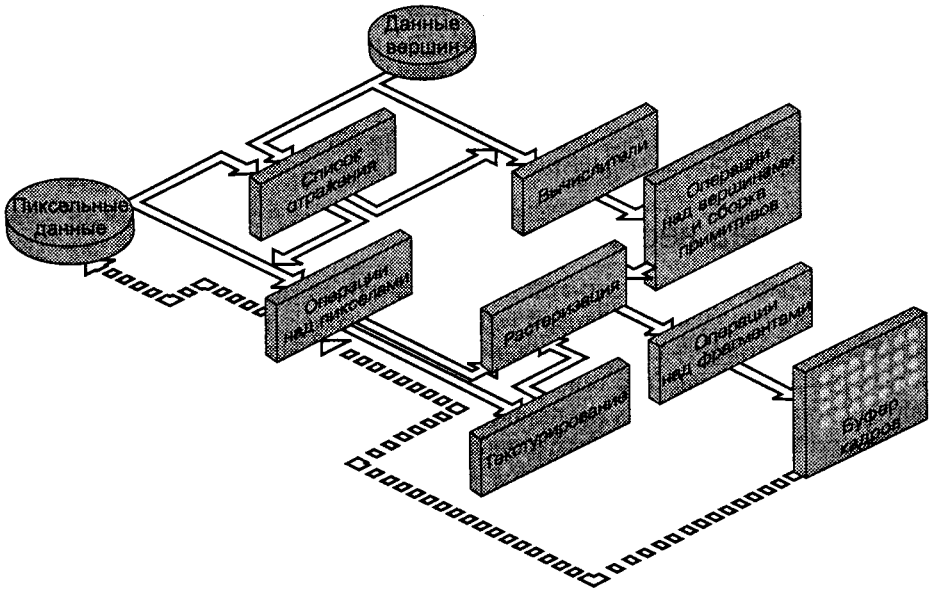


Рис. А1. Порядок операций

Все данные, векторные ли, растровые ли, могут быть сохранены в списке отображения и обработаны сразу же. Во время выполнения списка отображения данные, посылаемые из него, обрабатываются, как если бы они были посланы программой.

Любые геометрические примитивы в конечном счете описываются вершинами. Если используются вычислители, данные преобразуются в вершины и обрабатываются как вершины. Данные вершин могут быть также сохранены в специальных массивах вершин. Вычисления выполняются над каждой вершиной, с сопутствующей растеризацией во фрагменты. Для растровых данных применяются операции над пикселами, и результаты записываются в память текстуры, используемой для штрихования многоугольников или для их растеризации во фрагменты.

В итоге фрагменты подвергаются совокупности операций над фрагментами, после которых результирующие растровые данные отображаются в буфер кадров.

Векторные операции

Векторные данные, приходящие из списка отображения или вычислителя в виде вершин прямоугольника или необработанных данных, представляют собой набор вершин и типов описываемых примитивов (вершина, линия или многоугольник). Данные вершин включают в себя не только координаты (x, y, z, w), но также векторы нормалей, координаты текстуры, цвет RGBA, индекс цвета, свойства материала и признаки границ. Все эти элементы, за исключением координат вершины, могут быть заданы в любом порядке, или вообще не заданы, поскольку у них есть значения по умолчанию. Как только вызывается команда `glVertex*()`, компоненты дополняются до полного их описания, если это необходимо, для четырех измерений (где $z = 0$ и $w = 1$), и текущие значения всех этих элементов связываются с вершиной. Полный набор данных подается на обработку. (Если используются массивы вершин, такие данные могут быть обработаны в пакетном режиме, а однажды обработанные вершины могут использоваться повторно.)

Операции над вершинами

На этапе операций над вершинами каждая пространственная координата трансформируется по модельно-видовой матрице, а векторы нормалей преобразуются с помощью обратной транспонированной матрицы, если это указано. Когда включен режим автоматического генерирования текстур, из конвертированных координат вершин генерируются координаты текстуры, и они замещают собой старые координаты вершины. Координаты текстур преобразуются соответственно текущей матрице текстур и передаются на этап сборки примитивов.

Тем временем выполняются вычисления освещения, если это нужно, с использованием координат преобразованных вершин и векторов нормалей и текущих материалов, источников света и модели освещения. Эти вычисления вырабатывают новые цвета или индексы цвета, которые приводятся к соответствующему диапазону или маскируются и передаются на этап сборки примитивов.

Сборка примитивов

Сборка примитивов различается в зависимости от их типа — точка, прямая или многоугольник. Если включено плоское (равномерное) затенение, цвета и индексы цвета всех вершин линии или многоугольника устанавливаются в одно и то же значение. Если определены и активны плоскости отсечения, они используются для отсечения (clip) примитивов всех трех типов. (Уравнения плоскости отсечения, когда они указаны, преобразуются по обратной транспонированной модельно-видовой матрице.) При отсечении точек вершина либо передается дальше, либо отбраковывается. При отсечении линий и многоугольников можно добавить дополнительные вершины, в зависимости от характера отсечения. После отсечения пространственные координаты каждой вершины преобразуются по матрице проекции, и результаты используются взамен стандартных плоскостей отсечения $x = \pm w$, $y = \pm w$ и $z = \pm w$.

Если включен режим выбора, любой примитив, не отвергнутый при отсечении, генерирует отчет, и его дальнейшая обработка не выполняется. В отсутствие режима выбора имеет место разделение перспективы по парамеру w, и применяются операции окна просмотра и глубины. Также, если примитивом является многоугольник, он подвергается тесту отсева (culling) (если отсев включен). Многоугольник может быть сведен к вершинам или линиям, в зависимости от режима.

Наконец, точки, линии и многоугольники растеризуются во фрагменты, с учетом штриховки полигонов или прерывания линий, толщины линий и размеров точек. Растеризация выполняется в соответствии с тем, какие клетки целочисленной сетки в оконных координатах занимает примитив. Если включено устранение ступенчатости, вычисляется зона покрытия (часть занимаемой примитивом площади). Каждой такой зоне также назначаются цвет и глубина. Если включено смещение многоугольников, значения глубины слегка изменяются согласно вычисленному смещению.

Операции над пикселами

Растровые данные из основной памяти сначала распаковываются в надлежащее количество компонентов. Средства распаковки OpenGL оперируют различными форматами. Далее данные масштабируются, отклоняются (biased) и обрабатываются с использованием карты пикселов. Результаты ограничиваются (clamped) соответствующим типу данных диапазоном, а затем записываются в память текстур для использования при наложении текстур или для растеризации во фрагменты.

Если растровые данные читаются из буфера кадров, производятся операции при передаче пикселов (масштабирование, отклонение (bias), табличное преобразование (mapping) или ограничивание (clamping)). Результаты пакуются в надлежащий формат и возвращаются в оперативную память.

Операции копирования пикселов подобны комбинации паковки и распаковки, с той разницей, что последние излишни, и имеет место однократная передача данных перед тем, как они записываются обратно в буфер кадров.

Память текстур

Версия OpenGL 1.1 предоставляет дополнительное управление памятью текстур. Изображение текстуры может быть задано из памяти буфера кадров так же, как в оперативной памяти. Все части изображения текстуры могут быть подменены. Данные текстуры могут быть сохранены в объекте текстуры, который может быть загружен в память текстур. Если на память текстур одновременно претендует слишком много объектов, предпочтение отдается текстурам с большим приоритетом.

Операции над фрагментами

Если включено текстурирование, для каждого фрагмента генерируется элемент текстуры — тексел. Тогда, если включен расчет тумана, он выполняется с сопутствующими значениями покрытия (устранения ступенчатости), если активно устранение ступенчатости.

Далее приходит время отсечения (scissoring), представленного альфа-текстом (только в режиме RGBA), тестом трафарета и тестом глубины. Также в режиме RGBA, выполняется смешивание (blending). Смешивание сопровождается генерацией полутонов (dithering) и логическими операциями. Все эти операции могут быть не задействованы.

После фрагмент маскируется маской цвета или индекса цвета, в зависимости от режима, и отображается в соответствующий буфер. Если фрагменты были записаны в буфер трафарета или глубины, маскирование имеет место после тестов глубины и трафарета, и результаты помещаются в буфер кадров без смешивания, генерации полутонов и логических операций.

Прочее

Матричные операции выполняются с текущим стеком матриц: модельно-видовой, проекции или текстур. Команды `glMultMatrix*()`, `glLoadMatrix*()` и `glLoadIdentity()` применяются к вершине стека, а команды `glTranslate*()`, `glRotate*()`, `glScale*()`, `glOrtho()` и `glFrustum()` используются для создания матриц, умножаемых на верхнюю в стеке матрицу. Когда модифицируется модельно-видовая матрица, генерируется ее обратная транспонированная матрица для векторов нормалей.

Команды, устанавливающие текущую позицию растра, обрабатываются так же, как команды вершин, вплоть до этапа растеризации. Здесь эта позиция запоминается и используется при растеризации пикселов.

Различные варианты `glClear()` применимы ко всем операциям, кроме отсечения (scissoring), размытия (dithering) и маскирования записи (writemasking).

Параметры состояния Б

В этом приложении представлены доступные для запроса параметры состояния OpenGL, их значения по умолчанию и команды для получения значения этих параметров. Документ OpenGL Reference Manual содержит подробную информацию обо всех здесь перечисленных командах и константах, которые разбиты на две группы:

- Команды запроса
- Параметры состояния OpenGL

Команды запроса

В дополнение к основным командам для получения значения простых параметров состояния (таких как `glGetIntegerv()` и `glIsEnabled()`, о которых рассказывалось в разделе «Управление основными состояниями» главы 2), есть и другие специализированные команды, возвращающие более сложную информацию. Прототипы для этих особых команд перечислены ниже. Некоторые из них, такие как `glGetError()` и `glGetString()`, обсуждались в подробностях ранее.

Чтобы понять, когда вам нужно обращаться к этим командам и символическим константам, смотрите таблицы следующего раздела. Больше информации вы найдете в документе *OpenGL Reference Manual*.

void `glGetClipPlane`(GLenum *plane*, GLdouble **equation*);

void `glGetColorTable`(GLenum *target*, GLenum *pname*, GLenum *type*, GLvoid **table*);

void `glGetColorTableParameter{if}v`(GLenum *target*, GLenum *pname*, *TYPE* **params*);

void `glGetCompressedTexImage`(GLenum *target*, GLint *lod*, GLvoid **pixels*);

void `glGetConvolutionFilter`(GLenum *target*, GLenum *format*, GLenum *type*, GLvoid **image*);

void `glGetConvolutionParameter{if}v`(GLenum *target*, GLenum *pname*, *TYPE* **params*);

GLenum `glGetError`(void);

void `glGetHistogram`(GLenum *target*, GLboolean reset, GLenum *format*, GLenum *type*, GLvoid **values*);

void `glGetHistogramParameter{if}v`(GLenum *target*, GLenum *pname*, *TYPE* **params*);

void `glGetLight{if}v`(GLenum *light*, GLenum *pname*, *TYPE* **params*);

void `glGetMap{ifd}v`(GLenum *target*, GLenum *query*, *TYPE* **v*);

void `glGetMaterial{if}v`(GLenum *face*, GLenum *pname*, *TYPE* **params*);

void `glGetMinmax`(GLenum *target*, GLboolean reset, GLenum *format*, GLenum *type*, GLvoid **values*);

void `glGetMinmaxParameter{if}v` (GLenum *target*, GLenum *pname*, *TYPE* ***params*);

void `glGetPixelMap{f ui us}v`(GLenum *map*, *TYPE* **values*);

void `glGetPolygonStipple`(GLubyte **mask*);

void `glGetSeparableFilter`(GLenum *target*, GLenum *format*, GLenum *type*, GLvoid **row*, GLvoid **column*, GLvoid **span*);

const GLubyte * `glGetString`(GLenum *name*);

void `glGetTexEnv{if}v`(GLenum *target*, GLenum *pname*, *TYPE* **params*);

void `glGetTexGen{ifd}v`(GLenum *coord*, GLenum *pname*, *TYPE* **params*);

void `glGetTexImage`(GLenum *target*, GLint *level*, GLenum *format*, GLenum *type*, GLvoid **pixels*);

void `glGetTexLevelParameter{if}v`(GLenum *target*, GLint *level*, GLenum *pname*, *TYPE* **params*);

void `glGetTexParameter{if}v`(GLenum *target*, GLenum *pname*, *TYPE* **params*);

GLboolean `glIsList`(GLuint *list*);

GLboolean `glIsTexture`(GLuint *texObject*);

void `gluGetNurbsProperty`(GLUnurbsObj **nobj*, GLenum *property*, GLfloat **value*);

const GLubyte * `gluGetString`(GLenum *name*);

void `gluGetTessProperty`(GLUtesselator **tess*, GLenum *which*, GLdouble **data*).

Параметры состояния OpenGL

Далее до конца приложения следуют таблицы со списком названий запрашиваемых параметров состояния. Для каждого параметра приводится его краткое описание, указывается принадлежность к группе, представлены начальное и минимальное значения, а также вариант команды `glGet*()` для выполнения запроса. Для параметров состояния, которые могут быть получены с помощью `glGetBooleanv()`, `glGetIntegerv()`, `glGetFloatv()` и `glGetDoublev()`, указана только одна из этих команд — соответствующая типу данных. (Некоторые параметры массивов вершин могут быть запрошены посредством `glGetPointerv()`.) Эти параметры состояния не могут быть получены с помощью `glIsEnabled()`. Тем не менее те параметры, для которых `glIsEnabled()` задана как команда запроса, могут быть получены вызовами `glGetBooleanv()`, `glGetIntegerv()`, `glGetFloatv()` и `glGetDoublev()`. Любые другие параметры состояния могут быть получены только указанной для них командой.

ПРИМЕЧАНИЕ

Когда вы делаете запрос о состоянии текстуры, такой как GL_TEXTURE_MATRIX, в реализациях с присутствием расширения GL_ARB_multitexture значения, возвращаемые по ссылке, соответствуют только текущему модулю текстуры. См. раздел «Мультитекстурирование» главы 9 для получения более подробной информации.

Если указана группа (одна или более) атрибутов, параметр состояния считается принадлежащим этой группе. Если указание группы опущено, параметр не принадлежит ни одной из групп. Функции `glPushAttrib()`, `glPushClientAttrib()`, `glPopAttrib()` и `glPopClientAttrib()` могут быть использованы для сохранения и восстановления параметров состояния, принадлежащих конкретной группе (см. раздел «Наборы свойств» главы 2).

Все запрашиваемые параметры состояния имеют начальные значения; тем не менее, в зависимости от реализации, эти значения не всегда возможно перечислить. При отсутствии в описании начального значения обращайтесь за консультацией к руководству *OpenGL Reference Manual.*

Текущие значения и связанные данные

Таблица Б.1. Параметры состояния для текущих значений и связанных данных

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_CURRENT_COLOR	Текущий цвет	Текущие значения	(1, 1, 1, 1)	glGetIntegerv(), glGetFloatv()
GL_CURRENT_SECONDARY_COLOR	Текущий вторичный цвет	Текущие значения	(0, 0, 0, 0)	glGetIntegerv(), glGetFloatv()
GL_CURRENT_INDEX	Текущий индекс цвета	Текущие значения	1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_TEXTURE_COORDS	Текущие координаты текстуры	Текущие значения	(0, 0, 0, 1)	glGetFloatv()
GL_CURRENT_NORMAL	Текущая нормаль	Текущие значения	(0, 0, 1)	glGetFloatv()
GL_CURRENT_FOG_COORDINATE	Текущая координата тумана	Текущие значения	(0, 0, 1)	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_POSITION	Текущая позиция растра	Текущие значения	(0, 0, 0, 1)	glGetFloatv()
GL_CURRENT_RASTER_DISTANCE	Текущий интервал растра	Текущие значения	0	glGetFloatv()
GL_CURRENT_RASTER_COLOR	Цвет, связанный с текущей позицией растра	Текущие значения	(1, 1, 1, 1)	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_INDEX	Индекс цвета, связанный с текущей позицией растра	Текущие значения	1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_TEXTURE_COORDS	Координаты текстуры, связанные с текущей позицией растра	Текущие значения	(0, 0, 0, 1)	glGetFloatv()
GL_CURRENT_RASTER_POSITION_VALID	Бит допустимости текущей позиции растра	Текущие значения	GL_TRUE	glGetBooleanv()
GL_EDGE_FLAG	Признак границы		GL_TRUE	glGetBooleanv()

Массив вершин

Таблица Б.2. Параметры состояния массива вершин

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_CLIENT_ACTIVE_TEXTURE	Активный модуль текстуры, для массива координат текстуры	Массив вершин	GL_TEXTURE0	glGetIntegerv()
GL_VERTEX_ARRAY	Разрешение применения массива вершин	Массив вершин	GL_FALSE	glIsEnabled()
GL_VERTEX_ARRAY_SIZE	Число координат на вершину	Массив вершин	4	glGetIntegerv()
GL_VERTEX_ARRAY_TYPE	Тип координат вершины	Массив вершин	GL_FLOAT	glGetIntegerv()
GL_VERTEX_ARRAY_STRIDE	Шаг по индексу между вершинами	Массив вершин	0	glGetIntegerv()
GL_VERTEX_ARRAY_POINTER	Указатель на массив вершин	Массив вершин	NULL	glGetPointerv()
GL_NORMAL_ARRAY	Разрешение применения массива нормалей	Массив вершин	GL_FALSE	glIsEnabled()
GL_NORMAL_ARRAY_TYPE	Тип координат нормали	Массив вершин	GL_FLOAT	glGetIntegerv()
GL_NORMAL_ARRAY_STRIDE	Шаг по индексу между нормалями	Массив вершин	0	glGetIntegerv()
GL_NORMAL_ARRAY_POINTER	Указатель на массив нормалей	Массив вершин	NULL	glGetPointerv()
GL_FOG_COORDINATE_ARRAY	Разрешение применения массива координат тумана	Массив вершин	GL_FALSE	glIsEnabled()
GL_FOG_COORDINATE_ARRAY_TYPE	Тип компонентов координат тумана	Массив вершин	GL_FLOAT	glGetIntegerv()
GL_FOG_COORDINATE_ARRAY_STRIDE	Шаг по индексу между координатами тумана	Массив вершин	0	glGetIntegerv()
GL_FOG_COORDINATE_ARRAY_POINTER	Указатель на массив координат тумана	Массив вершин	NULL	glGetPointerv()
GL_COLOR_ARRAY	Разрешение применения массива цветов RGBA	Массив вершин	GL_FALSE	glIsEnabled()
GL_COLOR_ARRAY_SIZE	Число компонентов цвета на вершину	Массив вершин	4	glGetIntegerv()
GL_COLOR_ARRAY_TYPE	Тип компонентов цвета	Массив вершин	GL_FLOAT	glGetIntegerv()
GL_COLOR_ARRAY_STRIDE	Шаг по индексу между цветами	Массив вершин	0	glGetIntegerv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_COLOR_ARRAY_POINTER	Указатель на массив цветов	Массив вершин	NULL	glGetPointerv()
GL_SECONDARY_COLOR_ARRAY	Разрешение применения массива вторичных цветов	Массив вершин	GL_FALSE	glIsEnabled()
GL_SECONDARY_COLOR_ARRAY_SIZE	Число компонентов вторичного цвета на вершину	Массив вершин	3	glGetIntegerv()
GL_SECONDARY_COLOR_ARRAY_TYPE	Тип компонентов вторичного цвета	Массив вершин	GL_FLOAT	glGetIntegerv()
GL_SECONDARY_COLOR_ARRAY_STRIDE	Шаг по индексу между вторичными цветами	Массив вершин	0	glGetIntegerv()
GL_SECONDARY_COLOR_ARRAY_POINTER	Указатель на массив вторичных цветов	Массив вершин	NULL	glGetPointerv()
GL_INDEX_ARRAY	Разрешение использовать массив индексов цвета	Массив вершин	GL_FALSE	glIsEnabled()
GL_INDEX_ARRAY_TYPE	Тип индексов цвета	Массив вершин	GL_FLOAT	glGetIntegerv()
GL_INDEX_ARRAY_STRIDE	Шаг между индексами	Массив вершин	0	glGetIntegerv()
GL_INDEX_ARRAY_POINTER	Указатель на массив индексов	Массив вершин	NULL	glGetPointerv()
GL_TEXTURE_COORD_ARRAY	Разрешение применения массива координат текстуры	Массив вершин	GL_FALSE	glIsEnabled()
GL_TEXTURE_COORD_ARRAY_SIZE	Число координат текстуры на элемент	Массив вершин	4	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_TYPE	Тип координат текстуры	Массив вершин	GL_FLOAT	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_STRIDE	Шаг по индексу между координатами текстуры	Массив вершин	0	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_POINTER	Указатель на массив координат текстуры	Массив вершин	NULL	glGetPointerv()
GL_EDGE_FLAG_ARRAY	Разрешение применения массива признаков границ	Массив вершин	GL_FALSE	glIsEnabled()
GL_EDGE_FLAG_ARRAY_STRIDE	Шаг по индексу между признаками границ	Массив вершин	0	glGetIntegerv()
GL_EDGE_FLAG_ARRAY_POINTER	Указатель на массив признаков границ	Массив вершин	NULL	glGetPointerv()

Преобразования

Таблица Б3. Параметр состояния преобразований

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_COLOR_MATRIX	Стек матриц цвета	–	Единичная матрица	glGetFloatv()
GL_TRANSPOSE_COLOR_MATRIX	Стек транспонированных матриц цвета	–	Единичная матрица	glGetFloatv()
GL_MODELVIEW_MATRIX	Стек модельно-видовых матриц	–	Единичная матрица	glGetFloatv()
GL_TRANSPOSE_MODELVIEW_MATRIX	Стек транспонированных модельно-видовых матриц	–	Единичная матрица	glGetFloatv()
GL_PROJECTION_MATRIX	Стек матриц проекции	–	Единичная матрица	glGetFloatv()
GL_TRANSPOSE_PROJECTION_MATRIX–	Стек транспонированных матриц проекции	–	Единичная матрица	glGetFloatv()
GL_TEXTURE_MATRIX	Стек матриц текстур	–	Единичная матрица	glGetFloatv()
GL_TRANSPOSE_TEXTURE_MATRIX	Стек транспонированных матриц текстур	–	Единичная матрица	glGetFloatv()
GL_VIEWPORT	Начальные координаты и диапазон окна просмотра	Окно просмотра	–	glGetIntegerv()
GL_DEPTH_RANGE	Ближний и дальний уровни глубины	Окно просмотра	0, 1	glGetFloatv()
GL_COLOR_MATRIX_STACK_DEPTH	Указатель на стек матриц цвета	–	1	glGetIntegerv()
GL_MODELVIEW_STACK_DEPTH	Указатель на стек модельно-видовых матриц	–	1	glGetIntegerv()
GL_PROJECTION_STACK_DEPTH	Указатель на стек матриц проекции	–	1	glGetIntegerv()
GL_TEXTURE_STACK_DEPTH	Указатель на стек матриц текстур	–	1	glGetIntegerv()
GL_MATRIX_MODE	Текущий матричный режим	Преобразование	GL_MODELVIEW	glGetIntegerv()
GL_NORMALIZE	Нормализация нормали (вкл/выкл)	Преобразование/ доступность	GL_FALSE	glIsEnabled()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_RESCALE_NORMAL	Перемасштабирование нормали (вкл/выкл)	Преобразование/ доступность	GL_FALSE	glIsEnabled()
GL_CLIP_PLANE*i*	Коэффициенты пользовательской плоскости отсечения	Преобразование	(0, 0, 0, 0)	glGetClipPlane()
GL_CLIP_PLANE*i*	Состояние *i*-пользовательской плоскости отсечения (вкл/выкл)	Преобразование/ доступность	GL_FALSE	glIsEnabled()

Окрашивание

Таблица Б.4. Параметр состояния окрашивания

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_FOG_COLOR	Цвет тумана	Туман	(0, 0, 0, 0)	glGetFloatv()
GL_FOG_INDEX	Индекс цвета тумана	Туман	0	glGetFloatv()
GL_FOG_DENSITY	Степень плотности тумана	Туман	1.0	glGetFloatv()
GL_FOG_START	Глубина, с которой начинается туман	Туман	0.0	glGetFloatv()
GL_FOG_END	Глубина, на которой заканчивается туман	Туман	1.0	glGetFloatv()
GL_FOG_MODE	Режим тумана	Туман	GL_EXP	glGetIntegerv()
GL_FOG	Истина, если туман включен	Туман/доступность	GL_FALSE	glIsEnabled()
GL_FOG_COORDINATE_SOURCE	Источник координат для вычисления тумана	Туман	GL_FRAGMENT_DEPTH	glGetIntegerv()
GL_COLOR_SUM	Истина, если суммирование цветов включено	Туман/доступность	GL_FALSE	glIsEnabled()
GL_SHADE_MODEL	Настройки функции glShadeModel()	Освещение	GL_SMOOTH	glGetIntegerv()

Освещение

Информацию о значениях инициализации см. также в табл. 5.1 и 5.3.

Таблица Б.5. Параметры состояния освещения

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_LIGHTING	Истина, если освещение включено	Освещение/ доступность	GL_FALSE	glIsEnabled()
GL_COLOR_MATERIAL	Истина, если включена трассировка цвета	Освещение	GL_FALSE	glIsEnabled()
GL_COLOR_MATERIAL_PARAMETER	Свойства материала при отслеживании текущего цвета	Освещение	GL_AMBIENT_AND_ DIFFUSE	glGetIntegerv()
GL_COLOR_MATERIAL_FACE	Грани, в отношении которых эффективно отслеживание цвета	Освещение	GL_FRONT_AND_ BACK	glGetIntegerv()
GL_AMBIENT	Цвет неосвещенной части материала (ambient)	Освещение	(0.2, 0.2, 0.2, 1.0)	glGetMaterialfv()
GL_DIFFUSE	Цвет освещенной части материала (diffuse)	Освещение	(0.8, 0.8, 0.8, 1.0)	glGetMaterialfv()
GL_SPECULAR	Цвет, отраженный материалом (specular)	Освещение	(0.0, 0.0, 0.0, 1.0)	glGetMaterialfv()
GL_EMISSION	Цвет самосвечения материала (emissive)	Освещение	(0.0, 0.0, 0.0, 1.0)	glGetMaterialfv()
GL_SHININESS	Степень зеркального отражения света от материала	Освещение	0.0	glGetMaterialfv()
GL_LIGHT_MODEL_AMBIENT	Фоновый цвет сцены	Освещение	(0.2, 0.2, 0.2, 1.0)	glGetFloatv()
GL_LIGHT_MODEL_LOCAL_VIEWER	Режим локального наблюдателя	Освещение	GL_FALSE	glGetBooleanv()
GL_LIGHT_MODEL_TWO_SIDE	Двухстороннее освещение	Освещение	GL_FALSE	glGetBooleanv()
GL_LIGHT_MODEL_COLOR_CONTROL	Управление цветом	Освещение	GL_SINGLE_COLOR	glGetIntegerv()
GL_AMBIENT	Интенсивность фонового света от *i*-го источника	Освещение	(0.0, 0.0, 0.0, 1.0)	glGetLightfv()
GL_DIFFUSE	Интенсивность диффузного света от *i*-го источника	Освещение	–	glGetLightfv()
GL_SPECULAR	Интенсивность зеркального света от *i*-го источника	Освещение	–	glGetLightfv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_POSITION	Расположение *i*-го источника света	Освещение	(0.0, 0.0, 1.0, 0.0)	glGetLightfv()
GL_CONSTANT_ATTENUATION	Постоянный коэффициент затухания	Освещение	1.0	glGetLightfv()
GL_LINEAR_ATTENUATION	Линейный коэффициент затухания	Освещение	0.0	glGetLightfv()
GL_QUADRATIC_ATTENUATION	Квадратичный коэффициент затухания	Освещение	0.0	glGetLightfv()
GL_SPOT_DIRECTION	Направление света от *i*-го источника	Освещение	(0.0, 0.0, –1.0)	glGetLightfv()
GL_SPOT_EXPONENT	Сила света *i*-го источника	Освещение	0.0	glGetLightfv()
GL_SPOT_CUTOFF	Угол выхода света от *i*-го источника	Освещение	180.0	glGetLightfv()
GL_LIGHTi	Истина, если *i*-й источник света включен	Освещение/ доступность	GL_FALSE	glIsEnabled()
GL_COLOR_INDEXES	Величины c_a, c_d, и c_s для режима индексации цвета	Освещение/ доступность	0, 1, 1	glGetMaterialfv()

Растеризация

Таблица Б6. Параметры состояния растеризации

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_POINT_SIZE	Размер точки	Точка	1.0	glGetFloatv()
GL_POINT_SMOOTH	Сглаживание точек (antialiasing)	Точка/доступность	GL_FALSE	glIsEnabled()
GL_POINT_SIZE_MIN	Минимальный размер точки ослабления	Точка/доступность	0.0	glGetFloatv()
GL_POINT_SIZE_MAX	Максимальный размер точки ослабления	Точка/доступность	См. примечание в конце табл. Б6	glGetFloatv()
GL_POINT_FADE_THRESHOLD_SIZE	Порог для альфа-ослабления	Точка/доступность	1.0	glGetFloatv()
GL_POINT_DISTANCE_ATTENUATION	Коэффициенты ослабления	Точка/доступность	1, 0, 0	glGetFloatv()
GL_LINE_WIDTH	Ширина линии	Линия	1.0	glGetFloatv()
GL_LINE_SMOOTH	Сглаживание линий	Линия/доступность	GL_FALSE	glIsEnabled()
GL_LINE_STIPPLE_PATTERN	Шаблон прерывистых линий	Линия	Единицы	glGetIntegerv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_LINE_STIPPLE_REPEAT	Число повторов битов маски прерывистой линии	Линия	1	glGetIntegerv()
GL_LINE_STIPPLE	Прерывистые линии	Линия/доступность	GL_FALSE	glIsEnabled()
GL_CULL_FACE	Отсечение граней	Многоугольник/ доступность	GL_FALSE	glIsEnabled()
GL_CULL_FACE_MODE	Режим отсечения граней (лицевых и задних)	Многоугольник	GL_BACK	glGetIntegerv()
GL_FRONT_FACE	Направление обхода многоугольника при отсечении лицевых граней (по/против часовой стрелки)	Многоугольник	GL_CCW	glGetIntegerv()
GL_POLYGON_SMOOTH	Сглаживание многоугольников	Многоугольник/ доступность	GL_FALSE	glIsEnabled()
GL_POLYGON_MODE	Режим растеризации многоугольника (спереди и сзади)	Многоугольник	GL_FILL	glGetIntegerv()
GL_POLYGON_OFFSET_FACTOR	Фактор смещения (offset) многоугольника	Многоугольник	0	glGetFloatv()
GL_POLYGON_OFFSET_UNITS	Единицы смещения многоугольника	Многоугольник	0	glGetFloatv()
GL_POLYGON_OFFSET_POINT	Смещение многоугольника для режима GL_POINT	Многоугольник/ доступность	GL_FALSE	glIsEnabled()
GL_POLYGON_OFFSET_LINE	Смещение многоугольника для режима GL_LINE	Многоугольник/ доступность	GL_FALSE	glIsEnabled()
GL_POLYGON_OFFSET_FILL	Смещение многоугольника для режима GL_FILL	Многоугольник/ доступность	GL_FALSE	glIsEnabled()
GL_POLYGON_STIPPLE	Штриховка (stipple) многоугольников	Многоугольник/ доступность	GL_FALSE	glIsEnabled()
–	Шаблон штриховки многоугольника	Штриховка многоугольника	Единицы	glGetPolygon-Stipple()

Примечание. Значение инициализации GL_POINT_SIZE_MAX – это максимальное из GL_ALIASED_POINT_SIZE_RANGE и GL_SMOOTH_POINT_SIZE_RANGE.

Мультивыборка

Таблица Б7. Мультивыборка

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_MULTISAMPLE	Мультивыборка (multisampling)	Мультивыборка/ доступность	GL_TRUE	glIsEnabled()
GL_SAMPLE_ALPHA_TO_COVERAGE	Изменение покрытия для альфа-канала	Мультивыборка/ доступность	GL_FALSE	glIsEnabled()
GL_SAMPLE_ALPHA_TO_ONE	Установка альфа-канала в максимум	Мультивыборка/ доступность	GL_FALSE	glIsEnabled()
GL_SAMPLE_COVERAGE	Маска покрытия	Мультивыборка/ доступность	GL_FALSE	glIsEnabled()
GL_SAMPLE_COVERAGE_VALUE	Значение для маски	Мультивыборка	1	glGetFloatv()
GL_SAMPLE_COVERAGE_INVERT	Значение для инвертированной маски	Мультивыборка	GL_FALSE	glGetBooleanv()

Текстурирование

Таблица Б.8. Параметры состояния текстурирования

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_COMPRESSED_TEXTURE_FORMATS	Список поддерживаемых форматов текстур	–	Варианты	glGetIntegerv()
GL_TEXTURE_*x*	Истина, если текстурирование включено (*x* — 1D, 2D, 3D)	Текстура/ доступность	GL_FALSE	glIsEnabled()
GL_TEXTURE_*x*	x-D изображение текстуры *i*-го уровня детализации (*x* — 1D, 2D, 3D)	–	–	glGetTexImage()
GL_TEXTURE_BINDING_*x*	Объект текстуры, связанный с GL_ TEXTURE_*x* (*x* — 1D, 2D, 3D)	Текстура	0	glGetIntegerv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_TEXTURE_BINDING_CUBE_MAP	Объект текстуры, связанный с GL_TEXTURE_CUBE_MAP	Текстура	0	glGetIntegerv()
GL_TEXTURE_COMPRESSED	Истина, если карта текстуры сохранена в сжатом виде	–	GL_FALSE	glGetTexLevelParameter*()
GL_TEXTURE_COMPRESSED_IMAGE_SIZE	Количество байтов в массиве сжатого изображения текстуры	–	0	glGetTexLevelParameter*()
GL_TEXTURE_CUBE_MAP	Истина, если включено cube map-текстурирование	Текстура/доступность	GL_FALSE	glIsEnabled()
GL_TEXTURE_CUBE_MAP_POSITIVE_X	+x грань куба текстуры *i*-го уровня детализации	–	–	glGetTexImage()
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	–x грань куба текстуры *i*-го уровня детализации	–	–	glGetTexImage()
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	+y грань куба текстуры *i*-го уровня детализации	–	–	glGetTexImage()
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	–y грань куба текстуры *i*-го уровня детализации	–	–	glGetTexImage()
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	+z грань куба текстуры *i*-го уровня детализации	–	–	glGetTexImage()
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	–z грань куба текстуры *i*-го уровня детализации	–	–	glGetTexImage()
GL_TEXTURE_BORDER_COLOR	Цвет границы текстуры	Текстура	(0, 0, 0, 0)	glGetTexParameter*()
GL_TEXTURE_MIN_FILTER	Функция экранной миниатюризации (minification) текстуры относительно оригинальной	Текстура	GL_NEAREST_MIPMAP_LINEAR	glGetTexParameter*()
GL_TEXTURE_MAG_FILTER	Функция экранного увеличения (magnification) текстуры	Текстура	GL_LINEAR	glGetTexParameter*()
GL_TEXTURE_WRAP_S	Режим управления (wrap) S-координатой текстуры	Текстура	GL_REPEAT	glGetTexParameter*()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_TEXTURE_WRAP_T	Режим управления T-координатой текстуры (только для 2D-, 3D-текстур и кубов текстур)	Текстура	GL_REPEAT	glGetTexParameter*()
GL_TEXTURE_WRAP_R	Режим управления R-координатой текстуры (только для 3D-текстур)	Текстура	GL_REPEAT	glGetTexParameter*()
GL_TEXTURE_PRIORITY	Приоритет объекта текстуры	Текстура	1	glGetTexParameter*()
GL_TEXTURE_RESIDENT	Резидентность текстуры	Текстура	Варианты	glGetTexParameteriv()
GL_TEXTURE_MIN_LOD	Уровень минимальной детализации	Текстура	−1000	glGetTexParameterfv()
GL_TEXTURE_MAX_LOD	Уровень максимальной детализации	Текстура	1000	glGetTexParameterfv()
GL_TEXTURE_BASE_LEVEL	Базовый уровень детализации массива текстуры	Текстура	0	glGetTexParameterfv()
GL_TEXTURE_MAX_LEVEL	Уровень максимальной детализации массива текстуры	Текстура	1000	glGetTexParameterfv()
GL_DEPTH_TEXTURE_MODE	Режим глубины (depth) текстуры	Текстура	GL_LUMINANCE	glGetTexParameteriv()
GL_TEXTURE_COMPARE_MODE	Режим сравнения (comparison) текстур	Текстура	GL_NONE	glGetTexParameteriv()
GL_TEXTURE_COMPARE_FUNC	Функция сравнения текстур	Текстура	GL_LEQUAL	glGetTexParameteriv()
GL_GENERATE_MIPMAP	Автоматическая генерация мини-текстур (mipmap)	Текстура	GL_FALSE	glGetTexParameter*()
GL_TEXTURE_WIDTH	Ширина текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_HEIGHT	Высота 2D/3D-текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_DEPTH	Глубина 3D-текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_BORDER	Толщина границы текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_INTERNAL_FORMAT	Внутренний формат изображения текстуры *i*-го уровня детализации	–	1	glGetTexLevelParameter*()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_TEXTURE_RED_SIZE	Разрешение (resolution) красного компонента изображения текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_GREEN_SIZE	Разрешение зеленого компонента изображения текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_BLUE_SIZE	Разрешение синего компонента изображения текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_ALPHA_SIZE	Разрешение альфа-компонента изображения текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_LUMINANCE_SIZE	Разрешение яркости изображения текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_INTENSITY_SIZE	Разрешение интенсивности изображения текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_DEPTH_SIZE	Разрешение глубины изображения текстуры *i*-го уровня детализации	–	0	glGetTexLevelParameter*()
GL_TEXTURE_COMPRESSED	Истина, если изображение текстуры сохранено в сжатом внутреннем формате	–	GL_FALSE	glGetTexLevelParameter*()
GL_TEXTURE_COMPRESSED_IMAGE_SIZE	Размер (в GLubyte) сжатого изображения текстуры	–	0	glGetTexLevelParameter*()
GL_ACTIVE_TEXTURE	Модуль активной текстуры	Текстура	GL_TEXTURE0	glGetIntegerv()
GL_TEXTURE_ENV_MODE	Функция взаимодействия с материалом объекта (application) текстуры	Текстура	GL_MODULATE	glGetTexEnviv()
GL_TEXTURE_ENV_COLOR	Цвет материала, окружающего текстуру	Текстура	(0, 0, 0, 0)	glGetTexEnvfv()
GL_TEXTURE_LOD_BIAS	Уровень детализации отклонения (bias) для текстуры	Текстура	0.0	glGetTexEnvfv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_TEXTURE_GEN_*x*	Генерация координаты текстуры (x — S, T, R, Q)	Текстура/ доступность	GL_FALSE	glIsEnabled()
GL_EYE_PLANE	Генерация коэффициентов уравнения плоскости	Текстура	–	glGetTexGenfv()
GL_OBJECT_PLANE	Генерация линейных коэффициентов текстуры объекта	Текстура	–	glGetTexGenfv()
GL_TEXTURE_GEN_MODE	Функция, используемая для генерации координат текстуры	Текстура	GL_EYE_LINEAR	glGetTexGeniv()
GL_COMBINE_RGB	Функция смешивания (combiner) RGB	Текстура	GL_MODULATE	glGetTexEnviv()
GL_COMBINE_ALPHA	Функция смешивания альфа-компонента	Текстура	GL_MODULATE	glGetTexEnviv()
GL_SOURCE0_RGB	0-источник RGB	Текстура	GL_TEXTURE	glGetTexEnviv()
GL_SOURCE1_RGB	1-источник RGB	Текстура	GL_PREVIOUS	glGetTexEnviv()
GL_SOURCE2_RGB	2-источник RGB	Текстура	GL_CONSTANT	glGetTexEnviv()
GL_SOURCE0_ALPHA	0-источник альфа-компонента	Текстура	GL_TEXTURE	glGetTexEnviv()
GL_SOURCE1_ALPHA	1-источник альфа-компонента	Текстура	GL_PREVIOUS	glGetTexEnviv()
GL_SOURCE2_ALPHA	2-источник альфа-компонента	Текстура	GL_CONSTANT	glGetTexEnviv()
GL_OPERAND0_RGB	0-операнд RGB	Текстура	GL_SRC_COLOR	glGetTexEnviv()
GL_OPERAND1_RGB	1-операнд RGB	Текстура	GL_SRC_COLOR	glGetTexEnviv()
GL_OPERAND2_RGB	2-операнд RGB	Текстура	GL_SRC_ALPHA	glGetTexEnviv()
GL_OPERAND0_ALPHA	0-операнд альфа-компонента	Текстура	GL_SRC_ALPHA	glGetTexEnviv()
GL_OPERAND1_ALPHA	1-операнд альфа-компонента	Текстура	GL_SRC_ALPHA	glGetTexEnviv()
GL_OPERAND2_ALPHA	2-операнд альфа-компонента	Текстура	GL_SRC_ALPHA	glGetTexEnviv()
GL_RGB_SCALE	Пост-масштабирование для RGB	Текстура	1.0	glGetTexEnviv()
GL_ALPHA_SCALE	Пост-масштабирование для альфа-компонента	Текстура	1.0	glGetTexEnviv()

Операции над пикселами

Таблица Б.9. Операции над пикселами

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_SCISSOR_TEST	Обрезка (scissor)	Обрезка/доступность	GL_FALSE	glIsEnabled()
GL_SCISSOR_BOX	Рамка обрезки	Обрезка	–	glGetIntegerv()
GL_ALPHA_TEST	Альфа-тест	Буфер цвета/ доступность	GL_FALSE	glIsEnabled()
GL_ALPHA_TEST_FUNC	Функция альфа-теста	Буфер цвета	GL_ALWAYS	glGetIntegerv()
GL_ALPHA_TEST_REF	Опорное (reference) значение альфа-теста	Буфер цвета	0	glGetIntegerv()
GL_STENCIL_TEST	Трафарет	Буфер трафарета/ доступность	GL_FALSE	glIsEnabled()
GL_STENCIL_FUNC	Функция трафарета	Буфер трафарета	GL_ALWAYS	glGetIntegerv()
GL_STENCIL_VALUE_MASK	Маска трафарета	Буфер трафарета	Единицы	glGetIntegerv()
GL_STENCIL_REF	Опорное значение теста трафарета	Буфер трафарета	0	glGetIntegerv()
GL_STENCIL_FAIL	Действие при неудаче теста трафарета	Буфер трафарета	GL_KEEP	glGetIntegerv()
GL_STENCIL_PASS_DEPTH_FAIL	Действие, выполняемое при неудаче теста глубины	Буфер трафарета	GL_KEEP	glGetIntegerv()
GL_STENCIL_PASS_DEPTH_PASS	Действие, выполняемое при успехе теста глубины	Буфер трафарета	GL_KEEP	glGetIntegerv()
GL_DEPTH_TEST	Буфер глубины	Буфер глубины/ доступность	GL_FALSE	glIsEnabled()
GL_DEPTH_FUNC	Функция теста буфера глубины	Буфер глубины	GL_LESS	glGetIntegerv()
GL_BLEND	Смешивание	Буфер цвета/ доступность	GL_FALSE	glIsEnabled()
GL_BLEND_SRC_RGB	Функция источника RGB для смешивания	Буфер цвета	GL_ONE	glGetIntegerv()
GL_BLEND_SRC_ALPHA	Функция источника альфа-компонента для смешивания	Буфер цвета	GL_ONE	glGetIntegerv()
GL_BLEND_DST_RGB	Функция приемника RGB для смешивания	Буфер цвета	GL_ZERO	glGetIntegerv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_BLEND_DST_ALPHA	Функция приемника альфа-компонента для смешивания	Буфер цвета	GL_ZERO	glGetIntegerv()
GL_BLEND_EQUATION	Уравнение смешивания	Буфер цвета	GL_FUNC_ADD	glGetIntegerv()
GL_BLEND_COLOR	Постоянный цвет смешивания	Буфер цвета	(0, 0, 0, 0)	glGetFloatv()
GL_DITHER	Генерация полутонов (dithering)	Буфер цвета/ доступность	GL_TRUE	glIsEnabled()
GL_INDEX_LOGIC_OP	Логические операции в режиме индексации цвета	Буфер цвета/ доступность	GL_FALSE	glIsEnabled()
GL_COLOR_LOGIC_OP	Логические операции в режиме RGBA	Буфер цвета/ доступность	GL_FALSE	glIsEnabled()
GL_LOGIC_OP_MODE	Функция логических операций	Буфер цвета	GL_COPY	glGetIntegerv()

Управление буфером кадров

Таблица Б.10. Параметры состояния управления буфером кадра

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_DRAW_BUFFER	Буферы, выбранные для рисования	Буфер цвета	–	glGetIntegerv()
GL_INDEX_WRITEMASK	Маска записи индексов цвета	Буфер цвета	Единицы	glGetIntegerv()
GL_COLOR_WRITEMASK	Включение маски записи цветов; R, G, B и A	Буфер цвета	GL_TRUE	glGetBooleanv()
GL_DEPTH_WRITEMASK	Включение буфера глубины для записи	Буфер глубины	GL_TRUE	glGetBooleanv()
GL_STENCIL_WRITEMASK	Маска записи буфера трафарета	Буфер трафарета	Единицы	glGetIntegerv()
GL_COLOR_CLEAR_VALUE	Значение для очистки буфера цвета (режим RGBA)	Буфер цвета	(0, 0, 0, 0)	glGetFloatv()
GL_INDEX_CLEAR_VALUE	Значение для очистки буфера цвета (режим индексации цвета)	Буфер цвета	0	glGetFloatv()
GL_DEPTH_CLEAR_VALUE	Значение для очистки буфера глубины	Буфер глубины	1	glGetIntegerv()
GL_STENCIL_CLEAR_VALUE	Значение для очистки буфера трафарета	Буфер трафарета	0	glGetIntegerv()
GL_ACCUM_CLEAR_VALUE	Значение для очистки буфера накопления	Буфер накопления	0	glGetFloatv()

Пикселы

Таблица Б.11. Параметры состояния пиксела

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_UNPACK_SWAP_BYTES	Значение GL_UNPACK_SWAP_BYTES	Хранение пиксела	GL_FALSE	glGetBooleanv()
GL_UNPACK_LSB_FIRST	Значение GL_UNPACK_LSB_FIRST	Хранение пиксела	GL_FALSE	glGetBooleanv()
GL_UNPACK_IMAGE_HEIGHT	Значение GL_UNPACK_IMAGE_HEIGHT	Хранение пиксела	0	glGetIntegerv()
GL_UNPACK_SKIP_IMAGES	Значение GL_UNPACK_SKIP_IMAGES	Хранение пиксела	0	glGetIntegerv()
GL_UNPACK_ROW_LENGTH	Значение GL_UNPACK_ROW_LENGTH	Хранение пиксела	0	glGetIntegerv()
GL_UNPACK_SKIP_ROWS	Значение GL_UNPACK_SKIP_ROWS	Хранение пиксела	0	glGetIntegerv()
GL_UNPACK_SKIP_PIXELS	Значение GL_UNPACK_SKIP_PIXELS	Хранение пиксела	0	glGetIntegerv()
GL_UNPACK_ALIGNMENT	Значение GL_UNPACK_ALIGNMENT	Хранение пиксела	4	glGetIntegerv()
GL_PACK_SWAP_BYTES	Значение GL_PACK_SWAP_BYTES	Хранение пиксела	GL_FALSE	glGetBooleanv()
GL_PACK_LSB_FIRST	Значение GL_PACK_LSB_FIRST	Хранение пиксела	GL_FALSE	glGetBooleanv()
GL_PACK_IMAGE_HEIGHT	Значение GL_PACK_IMAGE_HEIGHT	Хранение пиксела	0	glGetIntegerv()
GL_PACK_SKIP_IMAGES	Значение GL_PACK_SKIP_IMAGES	Хранение пиксела	0	glGetIntegerv()
GL_PACK_ROW_LENGTH	Значение GL_PACK_ROW_LENGTH	Хранение пиксела	0	glGetIntegerv()
GL_PACK_SKIP_ROWS	Значение GL_PACK_SKIP_ROWS	Хранение пиксела	0	glGetIntegerv()
GL_PACK_SKIP_PIXELS	Значение GL_PACK_SKIP_PIXELS	Хранение пиксела	0	glGetIntegerv()
GL_PACK_ALIGNMENT	Значение GL_PACK_ALIGNMENT	Хранение пиксела	4	glGetIntegerv()
GL_MAP_COLOR	Истина, если преобразование по таблице соответствия цвета включено	Пиксел	GL_FALSE	glGetBooleanv()
GL_MAP_STENCIL	Истина, если преобразование значений трафарета включено	Пиксел	GL_FALSE	glGetBooleanv()
GL_INDEX_SHIFT	Значение GL_INDEX_SHIFT	Пиксел	0	glGetIntegerv()
GL_INDEX_OFFSET	Значение GL_INDEX_OFFSET	Пиксел	0	glGetIntegerv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_*x*_SCALE	Значение GL_*x*_SCALE: *x* — GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA или GL_DEPTH	Пиксел	1	glGetFloatv()
GL_*x*_BIAS	Значение GL_*x*_BIAS: *x* — GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA или GL_DEPTH	Пиксел	0	glGetFloatv()
GL_COLOR_TABLE	Истина, если включено табличное преобразование цвета	Пиксел/доступность	GL_FALSE	glIsEnabled()
GL_POST_CONVOLUTION_COLOR_TABLE	Истина, если включено табличное преобразование цвета на выходе свертки	Пиксел/доступность	GL_FALSE	glIsEnabled()
GL_POST_COLOR_MATRIX_COLOR_TABLE	Истина, если включено табличное преобразование цвета по матрице цвета	Пиксел/доступность	GL_FALSE	glIsEnabled()
GL_COLOR_TABLE	Таблицы цвета	–	Пусто	glGetColorTable()
GL_COLOR_TABLE_FORMAT	Внутренний формат таблиц цвета	–	GL_RGBA	glGetColorTable Parameteriv()
GL_COLOR_TABLE_WIDTH	Ширина таблицы цвета	–	0	glGetColorTable Parameteriv()
GL_COLOR_TABLE_*x*_SIZE	Разрешение (resolution) компонента таблицы цвета: *x* — GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE или GL_INTENSITY	–	0	glGetColorTable Parameteriv()
GL_COLOR_TABLE_SCALE	Масштабные коэффициенты для элементов таблицы цвета	Пиксел	(1, 1, 1, 1)	glGetColorTable Parameterfv()
GL_COLOR_TABLE_BIAS	Значения отклонения для элементов таблицы цвета	Пиксел	(0, 0, 0, 0)	glGetColorTable Parameterfv()
GL_CONVOLUTION_1D	Истина, если включена 1D-свертка	Пиксел/доступность	GL_FALSE	glIsEnabled()
GL_CONVOLUTION_2D	Истина, если включена 2D-свертка	Пиксел/доступность	GL_FALSE	glIsEnabled()
GL_SEPARABLE_2D	Истина, если включена разъемная 2D-свертка	Пиксел/доступность	GL_FALSE	glIsEnabled()
GL_CONVOLUTION_1D	1D-фильтр свертки	–	Пусто	glGetConvolution Filter()
GL_CONVOLUTION_2D	2D-фильтр свертки	–	Пусто	glGetConvolution Filter()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_SEPARABLE_2D	Разъемный 2D-фильтр свертки	–	Пусто	glGetSeparable Filter()
GL_CONVOLUTION_BORDER_ COLOR	Цвет границ при свертке	Пиксел	(0, 0, 0, 0)	glGetConvolution Parameterfv()
GL_CONVOLUTION_BORDER_ MODE	Режим обработки границ при свертке	Пиксел	GL_REDUCE	glGetConvolution Parameteriv()
GL_CONVOLUTION_FILTER_SCALE	Масштабные коэффициенты для элементов фильтра свертки	Пиксел	(1, 1, 1, 1)	glGetConvolution Parameterfv()
GL_CONVOLUTION_FILTER_BIAS	Масштабные коэффициенты для значений отклонения элементов фильтра свертки	Пиксел	(0, 0, 0, 0)	glGetConvolution Parameterfv()
GL_CONVOLUTION_FORMAT	Внутренний формат фильтра свертки	–	GL_RGBA	glGetConvolution Parameteriv()
GL_CONVOLUTION_WIDTH	Ширина фильтра свертки	–	0	glGetConvolution Parameteriv()
GL_CONVOLUTION_HEIGHT	Высота фильтра свертки	–	0	glGetConvolution Parameteriv()
GL_POST_CONVOLUTION_*x*_ SCALE	Коэффициенты масштабирования для компонента на выходе свертки: *x* — RED, GREEN, BLUE или ALPHA	Пиксел	1	glGetFloatv()
GL_POST_CONVOLUTION_*x*_BIAS	Коэффициенты масштабирования значений отклонения для компонента на выходе свертки: *x* — RED, GREEN, BLUE или ALPHA	Пиксел	0	glGetFloatv()
GL_POST_COLOR_MATRIX_*x*_ SCALE	Коэффициенты масштабирования на выходе преобразования по матрице цвета: *x* — RED, GREEN, BLUE или ALPHA	Пиксел	1	glGetFloatv()
GL_POST_COLOR_MATRIX_*x*_BIAS	Коэффициенты масштабирования для значений отклонения на выходе преобразования по матрице цвета: *x* — RED, GREEN, BLUE или ALPHA	Пиксел	0	glGetFloatv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_HISTOGRAM	Истина, если включен сбор статистики гистограммы	Пиксел/доступность	GL_FALSE	glIsEnabled()
GL_HISTOGRAM	Таблица гистограммы	–	Пусто	glGetHistogram()
GL_HISTOGRAM_WIDTH	Ширина таблицы гистограммы	–	0	glGetHistogram Parameteriv()
GL_HISTOGRAM_FORMAT	Внутренний формат таблицы гистограммы	–	GL_RGBA	glGetHistogram Parameteriv()
GL_HISTOGRAM_*x*_SIZE	Разрешение (resolution) компонента таблицы гистограммы: *x* — RED, GREEN, BLUE, ALPHA или LUMINANCE	–	0	glGetHistogram Parameteriv()
GL_HISTOGRAM_SINK	Истина, если гистограмма поглощает (consumes) группы пикселов	–	GL_FALSE	glGetHistogram Parameteriv()
GL_MINMAX	Истина, если метод минимакса включен	Пиксел/доступность	GL_FALSE	glIsEnabled()
GL_MINMAX	Таблица минимакса	–	См. примечание в конце табл. Б11	glGetMinmax()
GL_MINMAX_FORMAT	Внутренний формат таблицы минимакса	–	GL_RGBA	glGetMinmax Parameteriv()
GL_MINMAX_SINK	Истина, если метод минимакса поглощает (consumes) группы пикселов	–	GL_FALSE	glGetMinmax Parameteriv()
GL_ZOOM_X	Фактор масштабирования по x	Пиксел	1.0	glGetFloatv()
GL_ZOOM_Y	Фактор масштабирования по y	Пиксел	1.0	glGetFloatv()
GL_PIXEL_MAP_*x*	Таблицы преобразования для glPixelMap(): *x* — имя карты из табл. 8.1	–	Нули	glGetPixelMap*()
GL_PIXEL_MAP_*x*_SIZE	Размер *x*-таблицы преобразования для glPixelMap()	–	1	glGetIntegerv()
GL_READ_BUFFER	Буфер источника чтения	Пиксел	–	glGetIntegerv()

Примечание. Таблица минимумов инициализируется набором максимально представимых чисел. Таблица максимумов устанавливается в минимально представимые значения.

Вычислители

Таблица Б.12. Параметры состояния вычислителей

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_ORDER	Порядок 1D-карты	–	1	glGetMapiv()
GL_ORDER	Порядок 2D-карты	–	1, 1	glGetMapiv()
GL_COEFF	Контрольные точки (1D)	–	–	glGetMapfv()
GL_COEFF	Контрольные точки (2D)	–	–	glGetMapfv()
GL_DOMAIN	Конечные точки области (1D)	–	–	glGetMapfv()
GL_DOMAIN	Конечные точки области (2D)	–	–	glGetMapfv()
GL_MAP1_*x*	Включенные 1D-карты: *x* — тип карты	Вычислитель/доступность	GL_FALSE	glIsEnabled()
GL_MAP2_*x*	Включенные 2D-карты: *x* — тип карты	Вычислитель/доступность	GL_FALSE	glIsEnabled()
GL_MAP1_GRID_DOMAIN	Конечные точки сетки (1D)	Вычислитель	0, 1	glGetFloatv()
GL_MAP2_GRID_DOMAIN	Конечные точки сетки (2D)	Вычислитель	0, 1; 0, 1	glGetFloatv()
GL_MAP1_GRID_SEGMENTS	Деления сетки (1D)	Вычислитель	1	glGetFloatv()
GL_MAP2_GRID_SEGMENTS	Деления сетки (2D)	Вычислитель	1, 1	glGetFloatv()
GL_AUTO_NORMAL	Истина, если включена автогенерация нормалей	Вычислитель	GL_FALSE	glIsEnabled()

Рекомендации

Таблица Б.13. Параметры состояния рекомендаций

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_PERSPECTIVE_CORRECTION_HINT	Коррекция перспективы	Рекомендация	GL_DONT_CARE	glGetIntegerv()
GL_POINT_SMOOTH_HINT	Сглаживание точки	Рекомендация	GL_DONT_CARE	glGetIntegerv()
GL_LINE_SMOOTH_HINT	Сглаживание линии	Рекомендация	GL_DONT_CARE	glGetIntegerv()
GL_POLYGON_SMOOTH_HINT	Сглаживание многоугольника	Рекомендация	GL_DONT_CARE	glGetIntegerv()
GL_GENERATE_MIPMAP_HINT	Генерация мини-текстур (mipmap)	Рекомендация	GL_DONT_CARE	glGetIntegerv()
GL_FOG_HINT	Туман	Рекомендация	GL_DONT_CARE	glGetIntegerv()
GL_TEXTURE_COMPRESSION_HINT	Сжатие текстур	Рекомендация	GL_DONT_CARE	glGetIntegerv()

Параметры, зависящие от реализации

Таблица Б.14. Параметры состояния, зависящие от реализации

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_MAX_LIGHTS	Максимальное количество источников света	–	8	glGetIntegerv()
GL_MAX_CLIP_PLANES	Максимальное число пользовательских плоскостей отсечения	–	6	glGetIntegerv()
GL_MAX_COLOR_MATRIX_STACK_DEPTH	Максимальная глубина стека матриц цвета	–	2	glGetIntegerv()
GL_MAX_MODELVIEW_STACK_DEPTH	Максимальная глубина стека модельно-видовых матриц	–	32	glGetIntegerv()
GL_MAX_PROJECTION_STACK_DEPTH	Максимальная глубина стека матриц проекции	–	2	glGetIntegerv()
GL_MAX_TEXTURE_STACK_DEPTH	Максимальная глубина стека матриц текстур	–	2	glGetIntegerv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_SUBPIXEL_BITS	Количество битов для точности «подпикселов» по x и y	–	4	glGetIntegerv()
GL_MAX_3D_TEXTURE_SIZE	См. раздел «Текстуры-посредники» главы 9	–	16	glGetIntegerv()
GL_MAX_TEXTURE_SIZE	См. раздел «Текстуры-посредники» главы 9	–	64	glGetIntegerv()
GL_MAX_TEXTURE_LOD_BIAS	Максимальный уровень детализации отклонения для текстуры	–	2.0	glGetFloatv()
GL_MAX_CUBE_MAP_TEXTURE_SIZE	Максимальный уровень детализации отклонения для куба текстуры	–	16	glGetIntegerv()
GL_MAX_PIXEL_MAP_TABLE	Максимальный размер таблицы преобразования для glPixelMap()	–	32	glGetIntegerv()
li0@tabl_body = GL_MAX_NAME_ STACK_DEPTH	Максимальная глубина стека выбора имен	–	64	glGetIntegerv()
GL_MAX_LIST_NESTING	Максимальная вложенность списков отображения	–	64	glGetIntegerv()
GL_MAX_EVAL_ORDER	Максимальный порядок многочлена вычислителя	–	8	glGetIntegerv()
GL_MAX_VIEWPORT_DIMS	Максимальные размеры окна просмотра	–	–	glGetIntegerv()
GL_MAX_ATTRIB_STACK_DEPTH	Максимальная глубина стека атрибутов	–	16	glGetIntegerv()
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	Максимальная глубина стека атрибутов клиента	–	16	glGetIntegerv()
GL_NUM_COMPRESSED_TEXTURE_ FORMATS	Количество поддерживаемых форматов сжатия текстур	–	–	glGetIntegerv()
GL_AUX_BUFFERS	Количество дополнительных буферов	–	0	glGetBooleanv()
GL_RGBA_MODE	Истина, если режим хранения для буфера цвета — RGBA	–	–	glGetBooleanv()
GL_INDEX_MODE	Истина, если данные в буфере цвета хранятся по индексам цвета	–	–	glGetBooleanv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_DOUBLEBUFFER	Истина при наличии буферов переднего и заднего плана	–	–	glGetBooleanv()
GL_STEREO	Истина при наличии левого и правого буферов	–	–	glGetBooleanv()
GL_ALIASED_POINT_SIZE_RANGE	Диапазон размеров не сглаженных точек	–	1, 1	glGetFloatv()
GL_SMOOTH_POINT_SIZE_RANGE	Диапазон размеров сглаженных точек	–	1, 1	glGetFloatv()
GL_SMOOTH_POINT_SIZE_ GRANULARITY	Гранулярность сглаженных точек	–	–	glGetFloatv()
GL_ALIASED_LINE_WIDTH_RANGE	Диапазон толщины не сглаженных линий	–	1, 1	glGetFloatv()
GL_SMOOTH_LINE_WIDTH_RANGE	Диапазон толщины сглаженных линий	–	1, 1	glGetFloatv()
GL_SMOOTH_LINE_WIDTH_ GRANULARITY	Детализация толщины сглаженных линий	–	–	glGetFloatv()
GL_MAX_CONVOLUTION_WIDTH	Максимальная ширина фильтра свертки	–	3	glConvolution Parameteriv()
GL_MAX_CONVOLUTION_HEIGHT	Максимальная высота фильтра свертки	–	3	glConvolution Parameteriv()
GL_MAX_ELEMENTS_INDICES	Рекомендованное максимальное количество индексов для glDrawRangeElement()	–	–	glGetIntegerv()
GL_MAX_ELEMENTS_VERTICES	Рекомендованное максимальное количество вершин для glDrawRangeElement()	–	–	glGetIntegerv()
GL_MAX_TEXTURE_UNITS	Максимальное количество модулей текстуры (не больше 32)	–	2	glGetIntegerv()
GL_SAMPLE_BUFFERS	Количество буферов мультивыборки	–	0	glGetIntegerv()
GL_SAMPLES	Размер маски покрытия	–	0	glGetIntegerv()
GL_COMPRESSED_TEXTURE_FORMATS	Зарегистрированные форматы сжатия текстур	–	–	glGetIntegerv()
GL_NUM_COMPRESSED_TEXTURE_ FORMATS	Число зарегистрированных форматов сжатия текстур	–	0	glGetIntegerv()

Параметры глубины, зависящие от реализации

Таблица Б.15. Параметры состояния параметров глубины, зависящие от реализации

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_*x*_BITS	Количество битов для компонента *x*-буфера цвета (*x* — RED, GREEN, BLUE, ALPHA или INDEX)	–	–	glGetIntegerv()
GL_DEPTH_BITS	Количество битовых плоскостей буфера глубины	–	–	glGetIntegerv()
GL_STENCIL_BITS	Количество битовых плоскостей буфера трафарета	–	–	glGetIntegerv()
GL_ACCUM_*x*_BITS	Количество битов для компонента *x*-буфера накопления (*x* — RED, GREEN, BLUE или ALPHA)	–	–	glGetIntegerv()

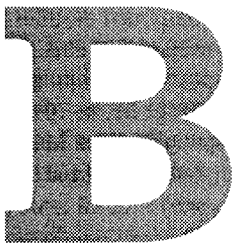
Прочее

Таблица Б.16. Прочие параметры состояния

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_LIST_BASE	Настройки glListBase()	Список	0	glGetIntegerv()
GL_LIST_INDEX	Количество не готовых (under construction) списков отображения; 0, если таких нет	–	0	glGetIntegerv()
GL_LIST_MODE	Режим для не готовых списков отображения; не определен, если таких нет	–	0	glGetIntegerv()
GL_ATTRIB_STACK_DEPTH	Указатель на стек атрибутов	–	0	glGetIntegerv()
GL_CLIENT_ATTRIB_STACK_DEPTH	Указатель на стек атрибутов клиента	–	0	glGetIntegerv()
GL_NAME_STACK_DEPTH	Глубина стека имен	–	0	glGetIntegerv()
GL_RENDER_MODE	Настройки glRenderMode()	–	GL_RENDER	glGetIntegerv()
GL_SELECTION_BUFFER_POINTER	Указатель на буфер выбора	Выбор	0	glGetPointerv()
GL_SELECTION_BUFFER_SIZE	Размер буфера выбора	Выбор	0	glGetIntegerv()

Параметр состояния	Описание	Группа	Значение инициализации	Команда запроса
GL_FEEDBACK_BUFFER_POINTER	Указатель на буфер обратной связи	Обратная связь	0	glGetPointerv()
GL_FEEDBACK_BUFFER_SIZE	Размер буфера обратной связи	Обратная связь	0	glGetIntegerv()
GL_FEEDBACK_BUFFER_TYPE	Тип буфера обратной связи	Обратная связь	GL_2D	glGetIntegerv()
–	Код текущей ошибки	–	0	glGetError()

OpenGL и оконные системы



Система OpenGL доступна на различных платформах и работает с различными оконными системами. Ее задача — дополнить собой оконные системы, но не дублировать их функциональность. Исходя из этого, OpenGL занимается тем, что воспроизводит векторные и растровые изображения в двух и трех измерениях, но не управляет окнами и не обрабатывает события.

Тем не менее на уровне базовых определений большинство оконных систем не поддерживают такую изощренную библиотеку, как OpenGL, с ее сложными и разнообразными растровыми форматами, с такими элементами, как буферы глубины, трафарета и накопления, а также двойная буферизация. Поддержка OpenGL в большинстве оконных систем потребовала добавления специальных подпрограмм.

В этом приложении представлены расширения для нескольких оконных и операционных систем: X Window System, Apple Mac OS, OS/2 Warp от IBM и Microsoft Windows. Чтобы это приложение не вызывало у вас непонимания, вы должны иметь знания о функционировании перечисленных оконных сред. Данное приложение содержит следующие основные разделы:

- Обращение к новым функциям OpenGL
- GLX: расширение OpenGL для X Window System
- AGL: расширение OpenGL для Apple Macintosh
- PGL: расширение OpenGL для IBM OS/2 Warp
- WGL: расширение OpenGL для Microsoft Windows 95/98/NT/ME/2000/XP

Обращение к новым функциям OpenGL

OpenGL все время изменяется. Производители аппаратного обеспечения под OpenGL добавляют новые расширения, а ассоциация OpenGL Architecture Review Board утверждает эти расширения и присоединяет их к ядру OpenGL. Так как каждый производитель вносит обновления в свою версию, заголовочные файлы (наподобие gl.h) и библиотека, используемая вами при компиляции (например, opengl32.lib в Microsoft Windows), могут иметь расхождения с последней версией, и, следовательно, вероятны ошибки при компиляции (или при сборке, что бывает реже) вашей программы.

С целью решить эту проблему в библиотеки для каждой из оконных систем был добавлен механизм обращения к новым функциям. Практические реализации его различны, но концепция одна для всех случаев. Вам потребуется указатель на функцию OpenGL, если вы захотите ее использовать. Нужен он тогда, когда функция недоступна явно из вашей библиотеки (если вы не знаете местонахождения процедуры, вы получите ошибку при сборке).

Каждая оконная система имеет свои собственные средства для получения указателей на функции OpenGL; конкретные специфики описаны в соответствующем разделе этого приложения.

Ниже приведен пример для Microsoft Windows:

```
#include <windows.h> /* для wglGetProcAddress() */
#include «glext.h»
PFNGLBINDPROGRAMARBPROC glBindProgramARB;
void
init(void)
{
   glBindProgramARB = (PFNGLBINDPROGRAMARB)
      wglGetProcAddress("glBindProgramARB");
   ...
}
```

Тип указателя на функцию в этом случае PFNGLBINDPROGRAMARBPROC, он определен в glext.h, как и любой из именованных идентификаторов, передаваемых вами в функции OpenGL. Если возвращаемое значение равно нулю, функция не поддерживается в данной реализации OpenGL.

GLX: расширение OpenGL для X Window System

В системе X Window визуализация OpenGL доступна как X-расширение в формальном смысле. GLX — это разрешение протокола X Window (и связанного с ним API) для взаимодействия команд OpenGL с сервером X Window. Подключение и аутентификация выполняются с помощью обычных X-механизмов.

Как и для других X-расширений, для команд OpenGL существует определенный сетевой протокол, инкапсулированный в байтовый поток X Window, тем самым поддерживается клиент-серверный вариант визуализации OpenGL. Поскольку

производительность критична для трехмерного воспроизведения изображения, X-расширение OpenGL позволяет OpenGL обходиться без участия X-сервера при кодировании данных, их копировании и интерпретации, используя непосредственно графический конвейер.

В GLX Version 1.3 появилось несколько радикальных изменений, в первую очередь это новая структура данных `GLXFBConfig`, описывающая конфигурацию буфера кадров GLX (включая глубину цветности буферов цвета, их типы, количество, а также размеры буферов глубины, трафарета, накопления и дополнительных буферов). Структура `GLXFBConfig` описывает эти атрибуты буфера кадров для воспроизводимой «поверхности» `GLXDrawable`.

GLX 1.3 предоставляет три типа `GLXDrawable`-поверхностей: `GLXWindow` (окно), `GLXPixmap` (пиксельная карта) и `GLXPbuffer` (буфер). Первая относится к экранным, а остальные — к внеэкранным. Поскольку структура `GLXPixmap` связана с пиксельной картой X, ее могут использовать как X, так и GLX. Расширение GLX выполняет визуализацию в `GLXPbuffer` единственным способом — эта «поверхность» хранит растровые данные в неотображаемой на экране памяти буфера кадров. (Наличие внеэкранного изображения не гарантирует, что поддерживается непосредственное.)

Структура объекта визуализации (visual) X очень важна для сопровождения информации растровых форматов, без которых не обходится окно OpenGL. Структура `XVisualInfo` хранит информацию растрового формата, к которой относятся тип пиксела (RGBA или индекс цвета), тип буферизации — однократная или двойная, глубина цветности и наличие буферов глубины, трафарета и накопления. Стандартный X-объект визуализации (например, `PseudoColor` или `TrueColor`) не описывает растровый формат в подробностях, в расчете на то, что каждая реализация сама расширяет число поддерживаемых объектов визуализации.

В GLX 1.3 поверхность `GLXWindow` имеет объект визуализации, ассоциированный с ее структурой `GLXFBConfig`. Для `GLXPixmap` или `GLXPbuffer` наличие такого объекта не обязательно. До появления GLX 1.3 все поверхности (окна или пиксельные карты) были связаны с объектами визуализации, а буферы не были частью GLX.

Подпрограммы GLX представлены более подробно в руководстве *OpenGL Reference Manual*. Интеграция приложений OpenGL с X Window System и пакетом Motif widget set досконально описана в книге *OpenGL Programming for the X Window System* Марка Килгарда (Mark Kilgard) (Addison–Wesley, 1996) с полными исходными кодами примеров. Если вы во что бы то ни стало хотите изучить GLX «изнутри», почитайте техническую документацию к расширению, которую можно найти на сайте http://www.opengl.org.

Инициализация

Используйте команды `glXQueryExtension()` и `glXQueryVersion()`, чтобы выяснить, определено ли расширение GLX для X-сервера и, если это так, какова его

версия. Команда `glXQueryExtensionsString()` возвращает информацию о клиент-серверном подключении, а `glXGetClientString()` — о библиотеке клиента, включая номера расширения и версии. Команда `glXQueryServerString()` возвращает аналогичную информацию о сервере.

Команда `glXChooseFBConfig()` возвращает указатель на массив структур `GLXFBConfig`, описывающих все конфигурации буфера кадров GLX согласно указанным клиентами атрибутам. Вы можете использовать команду `glXGetFBConfigAttrib()` для запроса конфигурации буфера кадров относительно поддержки в ней определенного атрибута GLX. Вы можете также вызвать `glXGetVisualFromFBConfig()` для извлечения информации об объекте визуализации, связанном с `GLXFBConfig`.

Создание областей изображения выполняется с некоторыми различиями, в зависимости от типа «поверхности» (drawable). Для `GLXWindow` сначала создайте X-окно с объектом визуализации X, соответствующим `GLXFBConfig`. Затем используйте это окно при вызове команды `glXCreateWindow()`, возвращающей структуру `GLXWindow`. Аналогично действуйте для `GLXPixmap` — сначала создайте пиксельную карту X с глубиной цветности, согласованной с `GLXFBConfig`. Далее используйте эту карту, когда вызываете `glXCreatePixmap()` для создания `GLXPixmap`. Буфер `GLXPbuffer` не требует X-окна или X-карты; просто вызовите `glXCreatePbuffer()` с соответствующей `GLXFBConfig`.

ПРИМЕЧАНИЕ

Если вы используете GLX 1.2 или более раннюю версию, вы не имеете структур GLXFBConfig. Вместо них используйте команду glXChooseVisual(), возвращающую указатель на структуру VisualInfo, описывающую объект визуализации X, согласованный с указанными клиентом атрибутами. Вы можете запросить объект визуализации на предмет поддержки им определенного атрибута OpenGL посредством glXGetConfig(). Для отображения в пиксельную карту вам следует использовать ранний вариант glXCreateGLXPixmap().

Обращение к функциям OpenGL

Чтобы получить указатель на функцию для расширений и новых возможностей X Window System, используйте процедуру `glXGetProcAddress()`. Она определена в заголовочном файле glxext.h, доступном для загрузки с веб-узла OpenGL.

Управление визуализацией

Несколько GLX-процедур отвечают за создание контекста воспроизведения изображения OpenGL и за управление им, а также за решение таких задач, как управление событиями GLX, синхронизация потоков X и OpenGL, перестановка переднего и заднего буферов и использование X-шрифтов.

Управление контекстом визуализации OpenGL

Контекст визуализации OpenGL создается с помощью команды `glXCreateNewContext()`. Один из аргументов этой процедуры позволяет вам напрямую запрашивать контекст визуализации, не привлекая к этому X-сервер. (Для выполнения непосредственного воспроизведения изображения подключе-

ние к X-серверу должно быть локальным, а реализация OpenGL должна поддерживать непосредственную визуализацию.) Команда `glXCreateNewContext()` также позволяет создавать индексы и определения списков отображения и объектов текстур, разделяемые множеством контекстов. Вы можете определить, является ли GLX-контекст непосредственным, с помощью команды `glXIsDirect()`.

Команда `glXMakeContextCurrent()` связывает контекст визуализации с текущим программным потоком (thread) визуализации и также создает две текущие поверхности. Вы можете рисовать в одну из них и читать растровые данные из другой. Во многих случаях поверхности *draw* (вывод) и *read* (чтение) представлены одной и той же структурой `GLXDrawable`. Команда `glXGetCurrentContext()` возвращает идентификатор текущего контекста. Вы можете также получить текущую *draw*-поверхность вызовом `glXGetCurrentDrawable()`, текущую *read*-поверхность вызовом `glXGetCurrentReadDrawable()` и текущий X-экран — вызовом `glXGetCurrentDisplay()`. В вашем распоряжении команда `glXQueryContext()`, позволяющая узнать текущие значения параметров контекста.

С каждым потоком в один момент времени может быть связан только один контекст. При наличии множества контекстов вы можете скопировать группу выбранных параметров состояния OpenGL из одного контекста в другой посредством `glXCopyContext()`. Когда контекст станет ненужным, удалите его командой `glXDestroyContext()`.

ПРИМЕЧАНИЕ

В GLX 1.2 и более ранних версиях используйте glXCreateContext() для создания контекста воспроизведения изображения и glXMakeCurrent() для его активизации. Вы можете объявить поверхность как отдельную read-поверхность, из-за чего здесь нет команды glXGetCurrentReadDrawable().

Обработка событий GLX

В GLX 1.3 были представлены события GLX, возвращаемые в стандартный поток событий X11. Обработка событий GLX была добавлена специально по причине потенциальной неопределенности содержимого буфера `GLXPbuffer`, который может быть затерт в любое время. В GLX 1.3 в вашем распоряжении команда `glXSelectEvent()` для выбора только одного события с маской `GLX_PBUFFER_CLOBBER_MASK`. С помощью стандартных процедур обработки X-событий вы можете сразу узнать, повреждена ли часть `GLXPbuffer` (или `GLXWindow`), а затем выполнить действия по ее восстановлению, если это нужно. (Также вам предоставлен вызов `glXGetSelectedEvent()`, позволяющий вспомнить, отслеживается ли уже данное GLX-событие.)

Синхронизация выполнения

Чтобы предотвратить запросы X Window до момента завершения визуализации OpenGL, используйте `glXWaitGL()`. Тогда любая ранее инициированная команда OpenGL гарантированно выполнится до передачи управления вызовам X-визуализации, сделанным после вызова `glXWaitGL()`. Хотя те же самые результаты можно получить с помощью `glFinish()`, команда `glXWaitGL()` не требует пере-

дачи данных серверу и обратно и, таким образом, более эффективна в случаях, когда клиент и сервер находятся на разных компьютерах.

И наоборот, чтобы блокировать выполнение последовательности команд OpenGL на время завершения любого ожидающего выполнения X-запроса, используйте `glXWaitX()`. Эта процедура гарантирует, что X-вызовы выполнятся до OpenGL-вызовов, сделанных после `glXWaitX()`.

Перестановка буферов

Для поверхностей с двойной буферизацией можно поменять местами передний и задний буферы посредством вызова `glXSwapBuffers()`. Вызов `glFlush()` неявно входит в эту процедуру.

Использование X-шрифтов

Скорейший путь использования X-шрифтов в OpenGL — воспользоваться командой `glXUseXFont()`. Эта процедура формирует списки отображения, каждый из которых вызывает `glBitmap()` для каждого символа указанных шрифта и размера его символов.

«Очистка» поверхностей

По завершении визуализации вы вправе удалить «поверхности», соответствующими вызовами `glXDestroyWindow()`, `glXDestroyPixmap()` или `glXDestroyPbuffer()`. (Эти подпрограммы недоступны до GLX 1.3, хотя в ранних версиях есть команда `glXDestroyGLXPixmap()`, ведущая себя аналогично `glXDestroyPixmap()`.)

Прототипы GLX

Инициализация

Выясняет, определено ли расширение GLX на X-сервере:

Bool `glXQueryExtension`(Display **dpy*, int **errorBase*, int **eventBase*);

Запрос номеров версии и расширения для клиента и сервера:

Bool `glXQueryVersion`(Display **dpy*, int **major*, int **minor*);

const char* `glXGetClientString`(Display **dpy*, int *name*);

const char* `glXQueryServerString`(Display **dpy*, int *screen*, int *name*);

const char* `glXQueryExtensionsString`(Display **dpy*, int *screen*);

Получение доступных конфигураций буфера кадров GLX:

GLXFBConfig * `glXGetFBConfigs`(Display **dpy*, int *screen*, int **nelements*);

GLXFBConfig * `glXChooseFBConfig`(Display **dpy*, int *screen*, const int *attribList*, int **nelements*);

Запрос к конфигурации буфера кадров GLX для получения информации об объекте визуализации:

int `glXGetFBConfigAttrib`(Display *dpy, GLXFBConfig *config*, int *attribute*, int **value*);

XVisualInfo * `glXGetVisualFromFBConfig`(Display *dpy, GLXFBConfig *config*);

Создание поверхностей, допускающих воспроизведение изображения (экранных и внеэкранных):

GLXWindow `glXCreateWindow`(Display *dpy, GLXFBConfig *config*, Window *win*, const int **attribList*);

GLXPixmap `glXCreatePixmap`(Display *dpy, GLXFBConfig *config*, Pixmap *pixmap*, const int **attribList*);

GLXPbuffer `glXCreatePbuffer`(Display *dpy, GLXFBConfig *config*, const int **attribList*);

Получение указателя на функцию OpenGL:

__GLXextFuncPtr `glXGetProcAddress`(const char* *funcName*);

Управление визуализацией

Управление контекстом визуализации OpenGL и запросы:

GLXContext `glXCreateNewContext`(Display *dpy, GLXFBConfig *config*, int *renderType*, GLXContext *shareList*, Bool *direct*);

Bool `glXMakeContextCurrent`(Display *dpy, GLXDrawable *drawable*, GLXDrawable *read*, GLXContext *context*);

void `glXCopyContext`(Display *dpy, GLXContext *source*, GLXContext *dest*, unsigned long *mask*);

Bool `glXIsDirect`(Display *dpy, GLXContext *context*);

GLXContext `glXGetCurrentContext`(void);

Display* `glXGetCurrentDisplay`(void);

GLXDrawable `glXGetCurrentDrawable`(void);

GLXDrawable `glXGetCurrentReadDrawable`(void);

int `glXQueryContext`(Display *dpy, GLXContext *context*,int *attribute*, int **value*);

void `glXDestroyContext`(Display *dpy, GLXContext *context*);

Получение или запрос события GLX:

int `glXSelectEvent`(Display *dpy, GLXDrawable *drawable*, unsigned long *eventMask*);

int `glXGetSelectedEvent`(Display *dpy, GLXDrawable *drawable*, unsigned long **eventMask*);

Синхронизация выполнения:

void `glXWaitGL`(void);

void `glXWaitX`(void);

Перестановка буферов:

void `glXSwapBuffers`(Display *dpy, GLXDrawable *drawable*);

Использование X-шрифта:

void `glXUseXFont`(Font *font*, int *first*, int *count*, int *listBase*);

«Очистка» поверхностей:

void `glXDestroyWindow`(Display **dpy*, GLXWindow *win*);

void `glXDestroyPixmap`(Display **dpy*, GLXPixmap *pixmap*);

void `glXDestroyPbuffer`(Display **dpy*, GLXPbuffer *pbuffer*);

Устаревшие прототипы GLX

Следующие подпрограммы были признаны устаревшими для GLX 1.3. Если вы используете GLX 1.2 или ее предшественниц, эти процедуры вам могут пригодиться.

Получение объекта визуализации:

XVisualInfo* `glXChooseVisual`(Display **dpy*, int *screen*, int **attribList*);

int `glXGetConfig`(Display **dpy*, XVisualInfo **visual*, int *attrib*, int **value*);

Управление контекстом воспроизведения изображения OpenGL:

GLXContext `glXCreateContext`(Display **dpy*, XVisualInfo **visual*, GLXContext *shareList*, Bool *direct*);

Bool `glXMakeCurrent`(Display **dpy*, GLXDrawable *drawable*, GLXContext *context*);

Выполнение внеэкранного воспроизведения изображения:

GLXPixmap `glXCreateGLXPixmap`(Display **dpy*, XVisualInfo **visual*, Pixmap *pixmap*);

void `glXDestroyGLXPixmap`(Display **dpy*, GLXPixmap *pix*);

AGL: расширение OpenGL для Apple Macintosh

В этом разделе охвачены процедуры, определенные как расширение OpenGL для Apple Macintosh (AGL). Для пользования ими нужно понимать, как компьютеры Macintosh управляют графическим выводом (QuickDraw). Также полезно иметь под рукой руководства *Macintosh Toolbox Essentials* и *Imaging with QuickDraw* из серии *Inside Macintosh*.

Для получения более полной информации (в том числе, где взять библиотеку OpenGL для Macintosh), обращайтесь к веб-сайту Apple: http://developer.apple.com/opengl.

Для платформы Macintosh визуализация OpenGL доступна как библиотека, которая либо уже скомпилирована, либо резидентна в виде расширения для приложения, намеревающегося ее использовать. Библиотека OpenGL реализована программно для систем, не имеющих видеоускорителей. Если ускорение графики доступно, им обеспечиваются возможности, родственные конвейеру OpenGL, а прочая функциональность реализуется программно.

Тип данных `AGLPixelFormat` (AGL-эквивалент `XVisualInfo`) предназначен для сопровождения растровой информации, к которой относятся тип пикселов (RGBA или индекс цвета), буферизация — однократная или двойная, глубина цветности и наличие буферов глубины, трафарета и накопления.

В отличие от реализаций OpenGL для других систем (таких как X Window), клиент-серверная модель не используется. Тем не менее требуется вызывать `glFlush()`, так как некоторые аппаратные ускорители буферизуют конвейер OpenGL и требуют его принудительного сброса.

Инициализация

Используйте команду `aglGetVersion()` для выяснения, какая версия AGL для Macintosh доступна. Возможности графических устройств и ваши требования для буферов визуализации согласуются с помощью команды `aglChoosePixelFormat()`. Она возвращает структуру AGLPixelFormat или NULL, в зависимости от способностей системы удовлетворить ваши требования.

Визуализация и контексты

Несколько подпрограмм AGL отвечают за создание контекста визуализации OpenGL и за управление им. Вы можете использовать такой контекст для отображения в окно или во внеэкранный графический мир за его пределами. Другие подпрограммы позволяют вам выполнять перестановку фонового и рабочего буферов визуализации, подстраивать буферы в ответ на события движения, изменения размеров или смены графического устройства, а также использовать шрифты Macintosh. Для программной визуализации (и, в некоторых случаях, визуализации с аппаратным ускорением) буферы создаются в пространстве системной памяти.

Управление контекстом визуализации OpenGL

Контекст визуализации OpenGL создается (окну должен быть назначен хотя бы один контекст) с помощью `aglCreateContext()`. Контекст инициализируется на основе указанного вами растрового формата.

Используйте команду `aglSetDrawable()` для прикрепления контекста к «поверхности» и затем команду `aglSetCurrentContext()`, чтобы сделать этот контекст текущим. В один момент времени с потоком управления может быть связан только один контекст и, следовательно, связанная с ним поверхность. То есть допустимо применение более чем одного контекста к определенной поверхности, но не одновременно. Две процедуры позволяют вам выяснить, какие из контекстов и поверхностей являются текущими: `aglGetCurrentContext()` и `aglGetDrawable()`.

При наличии множества контекстов вы можете копировать выбранные группы параметров состояния OpenGL из одного контекста в другой с помощью команды `aglCopyContext()`. Ставший ненужным контекст может быть удален вызовом `aglDestroyContext()`.

Экранная визуализация

При отображении на экран сначала нужно создать растровый формат. Контекст, основанный на этом растровом формате, закрепляется за окном вызовом `aglSetDrawable()`. Прямоугольник буфера может быть изменен вызовом `aglSetInteger(AGL_BUFFER_RECT,...)`.

Внеэкранная визуализация

Для отображения во внеэкранный контекст создайте растровый формат с помощью атрибута `AGL_OFFSCREEN`. Контекст, основанный на этом формате, прикрепляется к экрану с помощью команды `aglSetOffScreen()`.

Полноэкранная визуализация

Для отображения в полноэкранный контекст создайте растровый формат с помощью атрибута `AGL_FULLSCREEN`. Контекст, основанный на этом формате, прикрепляется к экрану с помощью команды `aglSetFullScreen()`.

Перестановка буферов

Для поверхностей с двойной буферизацией (согласно растровому формату текущего контекста визуализации), вызов `aglSwapBuffers()` обменивает местами передний и задний буферы. Прямоугольник обмена может быть изменен вызовом `aglSetInteger(AGL_SWAP_RECT,...)`. Неявный вызов `glFlush()` производится как часть этой операции.

Обновление буферов визуализации

Инструменты Apple Macintosh требуют, чтобы вы сами обрабатывали события, и не предоставляют библиотекам готового пути для перехвата событий из потока. Поэтому для того, чтобы поверхности, поддерживаемые OpenGL, могли приспособиться к изменению размеров, положения поверхности и к глубине пикселов, предусмотрен вызов `aglUpdateContext()`.

Эту подпрограмму следует вызывать в вашем коде обработки событий, когда одно из имевших место событий связано с текущей поверхностью. В идеале сцена должна быть заново перевоспроизведена после вызова на обновление с учетом изменений, сделанных в буферах визуализации.

Использование шрифтов Apple Macintosh

Скорейший путь использования шрифтов Macintosh предоставляется командой `aglUseFont()`. Эта процедура формирует списки отображения, каждый из которых вызывает `glBitmap()` для каждого нужного символа указанных шрифта и размера символов.

Обработка ошибок

Расширение OpenGL для Apple Macintosh предоставляет механизм для обработки ошибок — когда ошибка возникает, вы можете вызвать `aglGetError()` для получения более полного описания повлекших ее причин.

Прототипы AGL

Инициализация

Запрашивает информацию о версии:

void `aglGetVersion`(GLint **major*, GLint **minor*);

Запрашивает доступные растровые форматы:

AGLPixelFormat `aglChoosePixelFormat`(const AGLDevice **gdevs*, GLint *ndev*, const GLint **attribs*);

void `aglDestroyPixelFormat`(AGLPixelFormat *pix*);

AGLPixelFormat `aglNextPixelFormat`(AGLPixelFormat *pix*);

GLboolean `aglDescribePixelFormat`(AGLPixelFormat *pix*, GLint *attrib*, GLint **value*);

AGLDevice *`aglDevicesOfPixelFormat`(AGLPixelFormat *pix*, GLint **ndevs*);

Информационные функции воспроизведения изображения:

AGLRendererInfo `aglQueryRendererInfo`(const AGLDevice **gdevs*, GLint *ndev*);

void `aglDestroyRendererInfo`(AGLRendererInfo *rend*);

AGLRendererInfo `aglNextRendererInfo`(AGLRendererInfo *rend*);

GLboolean `aglDescribeRenderer`(AGLRendererInfo *rend*, GLint *prop*, GLint **value*);

Управление визуализацией

Управление контекстами визуализации OpenGL:

AGLContext `aglCreateContext`(AGLPixelFormat *pix*, AGLContext *share*);

GLboolean `aglDestroyContext`(AGLContext *ctx*);

GLboolean `aglCopyContext`(AGLContext *src*, AGLContext *dst*, GLuint *mask*);

GLboolean `aglUpdateContext`(AGLContext *ctx*);

Функции текущего состояния:

GLboolean `aglSetCurrentContext`(AGLContext *ctx*);

AGLContext `aglGetCurrentContext`(void);

Функции поверхностей:

GLboolean `aglSetDrawable`(AGLContext *ctx*, AGLDrawable *draw*);

GLboolean `aglSetOffScreen`(AGLContext *ctx*, GLsizei *width*,

GLsizei *height*, GLsizei *rowbytes*, GLvoid **baseaddr*);

GLboolean `aglSetFullScreen`(AGLContext *ctx*, GLsizei *width*, GLsizei *height*, GLsizei *freq*, GLint *device*);

AGLDrawable `aglGetDrawable`(AGLContext *ctx*);

Функции виртуального экрана:

GLboolean `aglSetVirtualScreen`(AGLContext *ctx*, GLint *screen*);

GLint `aglGetVirtualScreen`(AGLContext *ctx*);

Настройка конфигурации библиотеки:

GLboolean `aglConfigure`(GLenum *pname*, GLuint *param*);

Функции перестановки:

void `aglSwapBuffers`(AGLContext *ctx*);

Настройки контекста:

GLboolean `aglEnable`(AGLContext *ctx*, GLenum *pname*);

GLboolean `aglDisable`(AGLContext *ctx*, GLenum *pname*);

GLboolean `aglIsEnabled`(AGLContext *ctx*, GLenum *pname*);

GLboolean `aglSetInteger`(AGLContext *ctx*, GLenum *pname*, const GLint **params*);

GLboolean `aglGetInteger`(AGLContext *ctx*, GLenum *pname*, GLint **params*);

Функции для шрифтов:

GLboolean `aglUseFont`(AGLContext *ctx*, GLint *fontID*, Style *face*, GLint *size*, GLint *first*, GLint *count*, GLint *base*);

Функции обработки ошибок:

GLenum `aglGetError`(void);

const GLubyte *`aglErrorString`(GLenum *code*);

Функция перевода библиотеки в исходное состояние:

void `aglResetLibrary`(void);

PGL: расширение OpenGL для IBM OS/2 Warp

Визуализация OpenGL для IBM OS/2 Warp осуществляется с использованием процедур PGL, добавленных в OpenGL для интеграции с графическим API IBM—менеджером представлений (Presentation Manager, PM). PGL поддерживает как непосредственный контекст OpenGL (обычно более быстрый), так и косвенный контекст (допускающий некоторую интеграцию интерфейса Graphics Programming Interface (GPI) и визуализации OpenGL).

Тип данных `VISUALCONFIG` (PGL-эквивалент `XVisualInfo`) служит для поддержки конфигурации отображения, включающей в себя тип пикселов (RGBA или индекс цвета), буферизацию — однократную или двойную, глубину цветности и наличие буферов глубины, трафарета и накопления.

Для получения большей информации (в том числе, где взять библиотеку OpenGL для IBM OS/2 Warp, версия 3.0) вы можете пройти по ссылке http://www.austin.ibm.com/software/OpenGL.

Включенное в состав программного обеспечения руководство *OpenGL on OS/2 Warp* содержит более обстоятельную информацию. OpenGL поддерживается на уровне операционной системы, начиная с OS/2 версии 4.

Инициализация

Используйте `pglQueryCapability()` и `pglQueryVersion()` на предмет выяснения, поддерживается ли OpenGL на данном компьютере и, если это так, какой версией система представлена. Команда `pglChooseConfig()` возвращает указатель на структуру VISUALCONFIG, описывающую конфигурацию отображения, наилучшим образом удовлетворяющую показателям, указанным клиентом. Список конфигураций, поддерживаемых устройством графического вывода, может быть запрошен командой `pglQueryConfigs()`.

Управление визуализацией

Несколько PGL-подпрограмм отвечают за создание контекста визуализации OpenGL и за управление им, захват содержимого двоичных карт (bitmaps), синхронизацию взаимодействия менеджера представлений и потоков OpenGL, перестановку рабочего и фонового заднего буферов, использование цветовых палитр и использование логических шрифтов OS/2.

Управление контекстом визуализации OpenGL

Здесь контекст визуализации OpenGL создается с помощью вызова `pglCreateContext()`. Один из аргументов этой процедуры позволяет вам запрашивать непосредственный контекст, где вывод осуществляется напрямую без участия GPI в окно PM, что значительно быстрее. Определить, является ли контекст непосредственным, можно посредством вызова `pglIsIndirect()`.

Чтобы сделать контекст визуализации текущим, используйте команду `pglMakeCurrent()`; команда `pglGetCurrent-Context()` возвращает идентификатор текущего контекста. Вы можете также получить идентификатор текущего окна с помощью вызова `glGetCurrentWindow()`. Вы можете копировать несколько параметров состояния OpenGL из одного контекста в другой посредством `pglCopyContext()`. Ставший ненужным контекст удаляется командой `pglDestroyContext()`.

Доступ к переднему буферу

Для захвата доступа к представлению битового образа, содержащегося в переднем буфере, используйте `pglGrabFrontBitmap()`. При этом неявно вызывается `glFlush()`, и вы можете прочитать доступный только для чтения битовый образ. Непосредственно после завершения доступа вы должны вызвать команду `pglReleaseFrontBitmap()` для освобождения доступа к рабочему буферу.

Синхронизация взаимодействия

Чтобы предотвратить запросы GPI-визуализации на выполнение в то время, когда имеются ожидающие своей очереди команды OpenGL, вызовите `pglWaitGL()`. Тогда любые ранее вызванные команды OpenGL гарантированно будут обработаны до обработки вызовов GPI-визуализации, сделанных после `pglWaitGL()`.

И наоборот, с целью отложить выполнение команд OpenGL, пока обрабатываются GPI-запросы, используйте `pglWaitPM()`. Эта процедура гарантирует, что ранее инициированные вызовы GPI-визуализации выполнятся до любых OpenGL-вызовов, сделанных после `pglWaitPM()`.

ПРИМЕЧАНИЕ

Визуализация OpenGL и визуализация GPI могут быть интегрированы в одном окне только для случая косвенного контекста OpenGL.

Перестановка буферов

Для окон с двойной буферизацией рабочий и фоновый буферы можно поменять местами с помощью команды `pglSwapBuffers()`. Здесь по ходу неявно вызывается `glFlush()`.

Использование палитр индексов цвета

Когда вы работаете в 8-битовом (256 цветов) режиме, вы должны принимать в расчет управление цветовыми палитрами. Для конфигураций с режимом индексации цвета вызов `pglSelectColorIndexPalette()` информирует OpenGL, какая палитра индексов цвета нужна вашему контексту. Цветовая палитра должна быть выбрана перед тем, как контекст будет прикреплен к окну. В режиме RGBA OpenGL устанавливает палитру автоматически.

Использование логических шрифтов OS/2

Скорейший способ использования логических шрифтов OS/2 в OpenGL предоставляется командой `pglUseFont()`. Эта процедура формирует списки отображения, каждый из которых вызывает `glBitmap()` для каждого символа указанного шрифта с заданными размерами символов.

Прототипы PGL

Инициализация

Выясняет, поддерживается ли OpenGL, и, если это так, запрашивает номер версии:

long `pglQueryCapability`(HAB *hab*);

void `pglQueryVersion`(HAB *hab*, int **major*, int **minor*);

Выбор конфигурации отображения:

PVISUALCONFIG `pglChooseConfig`(HAB *hab*, int **attribList*);

PVISUALCONFIG * `pglQueryConfigs`(HAB *hab*);

Управление визуализацией

Управляют запросами к контексту визуализации OpenGL:

HGC `pglCreateContext`(HAB *hab*, PVISUALCONFIG *pVisualConfig*, HGC *shareList*, Bool is*Direct*);

Bool `pglDestroyContext`(HAB *hab*, HGC *hgc*);

Bool `pglCopyContext`(HAB *hab*, HGC *source*, HGC *dest*, GLuint *mask*);

Bool `pglMakeCurrent`(HAB *hab*, HGC *hgc*, HWND *hwnd*);

long `pglIsIndirect`(HAB *hab*, HGC *hgc*);

HGC `pglGetCurrentContext`(HAB *hab*);

HWND `pglGetCurrentWindow`(HAB *hab*);

Захват и освобождение доступа к битовому образу в переднем буфере:

Bool `pglGrabFrontBitmap`(HAB *hab*, HPS **hps*, HBITMAP **phbitmap*);

Bool `pglReleaseFrontBitmap`(HAB *hab*);

Синхронизация выполнения:

HPS `pglWaitGL`(HAB *hab*);

void `pglWaitPM`(HAB *hab*);

Обмен заднего и переднего буферов:

void `pglSwapBuffers`(HAB *hab*, HWND *hwnd*);

Нахождение нужной палитры индекса цвета:

void `pglSelectColorIndexPalette`(HAB *hab*, HPAL, *hpal*, HGC *hgc*);

Использование логического шрифта OS/2:

Bool `pglUseFont`(HAB *hab*, HPS *hps*, FATTRS **fontAttribs*, long *logicalId*, int *first*, int *count*, int *listBase*);

WGL: расширение OpenGL для Microsoft Windows 95/98/NT/ME/2000/XP

Визуализация OpenGL поддерживается для любых современных версий Microsoft Windows (Windows 95 и старше). Процедуры библиотеки Win32 нужны для инициализации растрового формата и управления визуализацией, а также для доступа к расширениям OpenGL. Часть подпрограмм, с префиксом `wgl`, расширяет Win32 с целью полной поддержки OpenGL.[1]

[1] По данным OpenGL.org, разработчики Microsoft отныне не будут включать в состав Windows свежие версии OpenGL, замораживая тем самым его развитие на версии 1.3. Более того, все приложения, работающие на OpenGL, будут иметь к нему доступ не напрямую, а через интерфейс DirectX, что заметно снизит скорость работы. Этот шаг был предпринят Microsoft в связи с тем, что новые версии OpenGL оказались быстрее DirectX, что могло перевернуть с ног на голову ситуацию с созданием игр. — *Примеч. перев.*

Для Win32/WGL структура PIXELFORMATDESCRIPTOR является ключевой структурой данных для поддержки информации растрового формата в отношении окна OpenGL. Тип данных PIXELFORMATDESCRIPTOR отслеживает растровую информацию, такую как тип пикселов (RGBA или индекс цвета), буферизация – однократная или двойная, глубина цветности и наличие буферов глубины, трафарета и накопления.

За подробной информацией о WGL обратитесь для начала к техническим статьям, доступным на сайте MSDN.

Инициализация

Используйте команду GetVersion() или более новую — GetVersionEx(), для получения информации о версии. Команда ChoosePixelFormat() пытается найти конфигурацию PIXELFORMATDESCRIPTOR с указанными атрибутами. Если подходящая конфигурация обнаружена, нужно вызвать функцию SetPixelFormat() для подтверждения использования растрового формата. Вы должны выбрать растровый формат для контекста устройства до вызова wglCreateContext().

Если вам требуется более детальная информация о формате, используйте DescribePixelFormat() или wglDescribeLayerPlane().

Обращение к функциям OpenGL

Чтобы получить указатели на функции расширения и на новые средства Microsoft Windows, используйте wglGetProcAddress(). Эта функция определена в заголовочном файле wingdi.h, который автоматически включается в состав windows.h при использовании последнего вашим приложением.

Управление визуализацией

Несколько процедур WGL отвечают за создание контекста визуализации OpenGL и управление им, визуализацию битовых образов, перестановку рабочего и фонового буферов, подбор цветовой палитры и за использование растровых или векторных шрифтов.

Управление контекстом визуализации OpenGL

Команда wglCreateContext() создает контекст визуализации OpenGL для отображения на устройство в выбранном растровом формате контекста устройства. (Чтобы создать контекст визуализации OpenGL для выше- или нижележащих окон, используйте команду wglCreateLayerContext().) Для того чтобы сделать контекст текущим, используйте команду wglMakeCurrent(); вызов wglGetCurrentContext() возвращает идентификатор текущего контекста. Вы можете также получить идентификатор текущего контекста устройства с помощью wglGetCurrentDC(). Вы можете скопировать часть параметров состояния OpenGL из одного контекста в другой посредством wglCopyContext() или создать два контекста, разделяемых списками изображения и объектами текстур,

с помощью `wglShareLists()`. Ставший ненужным контекст удаляется командой `wglDestroyContext()`.

Доступ к расширениям OpenGL

Используйте `wglGetProcAddress()` для доступа к зависящим от реализации расширениям OpenGL, для вызова их процедур. Узнать, какие расширения поддержаны в вашей реализации, можно с помощью `glGetString(GL_EXTENSIONS)`. Команда `wglGetProcAddress()`, когда ей передается правильное имя расширения, запрошенное `glGetString()`, возвращает указатель на функцию или `NULL`, если расширение не поддерживается.

Визуализация OpenGL в битовый образ

Библиотека Win32 включает в себя несколько подпрограмм для выделения (и удаления) битовых образов, в которые вы можете производить непосредственную визуализацию OpenGL. Команда `CreateDIBitmap()` — независимый от устройства образ (device-dependent bitmap, DDB) из зависимого от устройства битового образа (device-independent bitmap, DIB). Команда `CreateDIBSection()` — создает DIB-образ, в который приложение может записывать данные напрямую. Для удаления битового образа (освобождения занимаемой им памяти) используйте `DeleteObject()`.

Синхронизация выполнения

В Win32 нет полноценных эквивалентов функций `glXWaitGL()`, `glXWaitX()` и `pglWaitGL()`, позволяющих сочетать GDI- и OpenGL-визуализацию. Но хотя `glXWaitGL()` не воспроизведена в Win32, вы можете добиться того же эффекта с помощью `glFinish()`, которая выжидает, пока все задержанные команды OpenGL выполнятся, или с помощью `GdiFlush()`, делающей то же самое в отношении GDI-отображения.

Перестановка буферов

Для окон с двойной буферизацией передний и задний буферы можно поменять местами, вызвав `SwapBuffers()` или `wglSwapLayerBuffers()`; последняя используется для вышележащих и нижележащих окон.

Подбор палитры цветов

Чтобы получить доступ к цветовой палитре для стандартных (не слоев) битовых плоскостей, используйте стандартные функции GDI, устанавливающие отдельные элементы палитры. Для выше- и нижележащих слоев используйте `wglRealizeLayerPalette()`, которая отображает элементы индексно-цветовой палитры из заданной плоскости в физическую палитру или инициализирует палитру RGBA. Команды `wglGetLayerPaletteEntries()` и `wglSetLayerPaletteEntries()` используются для послойного запроса и установки элементов палитры.

Использование растровых и векторных шрифтов

В WGL имеется две подпрограммы, `wglUseFontBitmaps()` и `wglUseFontOutlines()`, предназначенные для преобразования системных шрифтов с целью их использования в OpenGL. Обе подпрограммы формируют список отображения для каждого символа указанного шрифта.

Прототипы WGL

Инициализация

Запрашивает информацию о версии:

BOOL `GetVersion`(LPOSVERSIONINFO *lpVersionInformation*);

BOOL `GetVersionEx`(LPOSVERSIONINFO *lpVersionInformation*);

Определение доступности растрового формата, его выбор и характеристики:

int `ChoosePixelFormat`(HDC *hdc*, CONST PIXELFORMATDESCRIPTOR * *ppfd*);

BOOL `SetPixelFormat`(HDC *hdc*, int *iPixelFormat*, CONST PIXELFORMATDESCRIPTOR * *ppfd*);

int `DescribePixelFormat`(HDC *hdc*, int *iPixelFormat*, UINT *nBytes*, LPPIXELFORMATDESCRIPTOR *ppfd*);

BOOL `wglDescribeLayerPlane`(HDC *hdc*, int *iPixelFormat*, int *iLayerPlane*, UINT *nBytes*, LPLAYERPLANEDESCRIPTOR *plpd*);

Получение указателя на функцию OpenGL:

PROC `wglGetProcAddress`(LPCSTR *funcName*);

Управление визуализацией

Запрос контекста визуализации OpenGL и управление им:

HGLRC `wglCreateContext`(HDC *hdc*);

HGLRC `wglCreateLayerContext`(HDC *hdc*, int *iLayerPlane*);

BOOL `wglShareLists`(HGLRC *hglrc1*, HGLRC *hglrc2*);

BOOL `wglDeleteContext`(HGLRC *hglrc*);

BOOL `wglCopyContext`(HGLRC *hglrcSource*, HGLRC *hlglrcDest*, UINT *mask*);

BOOL `wglMakeCurrent`(HDC *hdc*, HGLRC *hglrc*);

HGLRC `wglGetCurrentContext`(VOID);

HDC `wglGetCurrentDC`(VOID);

Доступ к процедурам зависящих от реализации расширений:

PROC `wglGetProcAddress`(LPCSTR *lpszProc*);

Выделение битового образа переднего буфера и его освобождение:

HBITMAP `CreateDIBitmap`(HDC *hdc*, CONST BITMAPINFOHEADER **lpbmih*, DWORD *fdwInit*, CONST VOID **lpbInit*, CONST BITMAPINFO **lpbmi*, UINT *fuUsage*);

HBITMAP `CreateDIBSection`(HDC *hdc*, CONST BITMAPINFO **pbmi*, UINT *iUsage*, VOID **ppvBits*, HANDLE *hSection*, DWORD *dwOffset*);

BOOL `DeleteObject`(HGDIOBJ *hObject*);

Перестановка рабочего и фонового буферов:

BOOL `SwapBuffers`(HDC *hdc*);

BOOL `wglSwapLayerBuffers`(HDC *hdc*, UINT *fuPlanes*);

Обнаружение цветовой палитры для вышележащих и нижележащих слоев:

int `wglGetLayerPaletteEntries`(HDC *hdc*, int *iLayerPlane*, int *iStart*, int *cEntries*, CONST COLORREF **pcr*);

int `wglSetLayerPaletteEntries`(HDC *hdc*, int *iLayerPlane*, int *iStart* int *cEntries*, CONST COLORREF **pcr*);

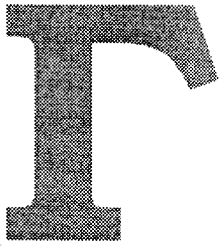
BOOL `wglRealizeLayerPalette`(HDC *hdc*, int *iLayerPlane*, BOOL *bRealize*);

Использование растрового или векторного шрифта:

BOOL `wglUseFontBitmaps`(HDC *hdc*, DWORD *first*, DWORD *count*, DWORD *listBase*);

BOOL `wglUseFontOutlines`(HDC *hdc*, DWORD *first*, DWORD *count*, DWORD *listBase*, FLOAT *deviation*, FLOAT *extrusion*, int *format*, LPGLYPHMETRICSFLOAT *lpgmf*);

Основы GLUT: инструментальный набор пакета разработчика OpenGL



Это приложение описывает подмножество инструментального пакета разработчика OpenGL от Марка Килгарда (Mark Kilgard) (GLUT), который полностью документирован в его книге, OpenGL Programming for the X Window System (Addison–Wesley, 1996). GLUT стала популярной библиотекой среди OpenGL-разработчиков благодаря тому, что стандартизует управление окнами и событиями и упрощает их. Библиотека GLUT была надстроена над различными реализациями OpenGL, включая версии для X Window System и Microsoft Windows.

Это приложение разбито на следующие основные разделы

- Инициализация и создание окон
- Управление окнами и событиями
- Загрузка карты цвета
- Инициализация и отображение трехмерных объектов
- Управление фоновыми процессами
- Запуск программ

О том, где взять исходный код GLUT, см. раздел «Как получить примеры программ» в предисловии.

С помощью GLUT ваше приложение структурирует обработку событий, опираясь на функции обратной связи. (Этот метод направлен на работу с пакетом Xt Toolkit, также известным как X Intrinsics, включающим в себя набор специальных компонентов — виджетов (widgets).) Например, сначала вы открываете окно и регистрируете процедуры обратной связи для конкретных событий. Затем вы организуете главный цикл. В этом цикле, если происходят события, вызываются на выполнение зарегистрированные для них функции обратной связи. Когда такая функция заканчивает работу, вы снова возвращаетесь в главный цикл.

Инициализация и создание окон

Перед открытием окна вам нужно задать его характеристики. Будет ли оно с однократной или двойной буферизацией? Как будут храниться цвета — в компонентах RGBA или как индексы цвета? Где на экране должно отображаться окно? Ответы на все эти вопросы дают команды `glutInit()`, `glutInitDisplayMode()`, `glutInitWindowSize()` и `glutInitWindowPosition()`, вызываемые перед `glutCreateWindow()`, которая непосредственно высвечивает окно.

void `glutInit`(int *argc*, char ***argv*);

Должна быть вызвана перед любой другой подпрограммой GLUT, поскольку именно эта команда инициализирует библиотеку GLUT. Эта команда умеет также обрабатывать параметры командой строки, но в зависимости от конкретной оконной системы. Для X Window она разбирает ключи `-iconic`, `-geometry` и `-display`. (То есть параметры, предназначенные для `main()`.)

void `glutInitDisplayMode`(unsigned int *mode*);

Задает режимы отображения (такие как RGBA или индексация цвета, однократная или двойная буферизация) для окон, создаваемых вызовом `glutCreateWindow()`. Вы можете также задать связанные с окном буферы глубины, трафарета и/или накопления. Аргумент маски *mask* является поразрядной (через OR) комбинацией `GLUT_RGBA` или `GLUT_INDEX`, `GLUT_SINGLE` или `GLUT_DOUBLE` и других признаков для буферов: `GLUT_DEPTH`, `GLUT_STENCIL` или `GLUT_ACCUM`. Так, для двойной буферизации в режиме RGBA совместно с буферами трафарета и глубины используйте `GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL`. Значение по умолчанию — `GLUT_RGBA | GLUT_SINGLE` (режим RGBA, однократная буферизация).

void `glutInitWindowSize`(int *width*, int *height*);

void `glutInitWindowPosition`(int *x*, int *y*);

Присваивают окнам, созданным с помощью `glutCreateWindow()`, начальные размеры и позицию. Аргументы (*x*, *y*) указывают положение угла окна относительно полного экрана. Ширина *width* и высота *height* задают размеры окна в пикселах. Начальные размеры и позиция окна являются рекомендациями и могут быть впоследствии переопределены.

int glutCreateWindow(char *name);

Открывает окно с ранее заданными характеристиками (режимы отображения, ширина, высота и т. д.). Строка *name* может быть выведена в заголовке окна, если оконная система это поддерживает. Окно не показывается до момента входа в главный цикл glutMainLoop(), поэтому не спешите что-либо отображать в него.

Возвращаемое значение представляет собой уникальный целочисленный идентификатор окна. Эти идентификаторы используются для управления множеством окон и выводом в них (согласно контексту) по указаниям программы.

Управление окнами и событиями

После создания окна, но до входа в главный цикл, вы должны зарегистрировать функции обратной связи с помощью следующих вызовов.

void glutDisplayFunc(void (*func)(void));

Указывает функцию, которая будет вызвана при потребности перерисовать содержимое окна. Это обычно нужно в момент первого открытия окна, при его повторном всплытии или связано с воздействием других окон, или при явном вызове glutPostRedisplay().

void glutReshapeFunc(void (*func)(int *width*, int *height*));

Задает функцию, вызываемую на выполнение в момент изменения размеров окна или его положения. Аргумент *func* — это указатель на функцию, принимающую два аргумента — новые ширину и высоту окна. Обычно процедура *func* вызывает команду glViewport() с целью согласовать новые размеры с содержимым экрана, и это переопределяет матрицу проекции так, чтобы коэффициент сжатия проецируемого изображения соответствовал окну просмотра, и пропорции были соблюдены. Если glutReshapeFunc() не вызывается или она дерегистрирована нулевым указателем, вызывается функция переотрисовки по умолчанию, которая, в свою очередь, делает вызов glViewport(*0*, *0*, *width*, *height*).

void glutKeyboardFunc(void (*func)(unsigned char *key*, int *x*, int *y*));

Задает функцию *func*, вызываемую в момент нажатия клавиши, генерирующей ASCII-код *key*. Аргументы *x* и *y* указывают положение мыши (в системе координат окна) в момент нажатия клавиши.

void glutMouseFunc(void (*func)(int *button*, int *state*, int *x*, int *y*));

Задает функцию *func*, вызываемую при нажатии или отпускании кнопки мыши. Кнопка *button* — это GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON или GLUT_RIGHT_BUTTON. Аргумент состояния *state* равен GLUT_UP или GLUT_DOWN, в зависимости от того, нажимается или отпускается кнопка. Аргументы *x* и *y* указывают положение указателя мыши (в оконных координатах) в момент возникновения события.

void `glutMotionFunc`(void (**func*)(int *x*, int *y*));
Задает функцию *func*, вызываемую при перемещении указателя мыши в пределах окна, когда нажаты одна или более кнопок мыши. Аргументы *x* и *y* указывают положение (в оконных координатах) указателя мыши в момент возникновения события.

void `glutPostRedisplay`(void);
Помечает текущее окно как требующее переотрисовки. При первой возможности будет вызвана функция обратной связи, зарегистрированная `glutDisplayFunc()`.

Загрузка карты цвета

Если вы используете режим индексации цвета, для вас может оказаться неожиданностью, что в OpenGL нет средства для загрузки таблицы цвета или таблицы подстановки цвета. Это сделано сознательно, поскольку для каждой оконной системы процесс загрузки карты цвета сугубо индивидуален. Библиотека GLUT предоставляет процедуру общего характера для загрузки в память одного индекса цвета или RGB-значения — `glutSetColor()`.

void `glutSetColor`(GLint *index*, GLfloat *red*, GLfloat *green*, GLfloat *blue*);
Загружает индекс в карте цветов *(index)* с заданными значениями красного *(red)*, зеленого *(green)* и синего *(blue)* компонентов. Эти значения нормализуются к диапазону [0,0; 1,0].

Инициализация и отображение трехмерных объектов

Во многих примерах программ в этом руководстве используются трехмерные модели, иллюстрирующие различные аспекты воспроизведения объектов. Следующие процедуры рисования включены в библиотеку GLUT с целью не писать повторно соответствующий код в каждой программе. Эти процедуры отображают свои графические объекты в режиме непосредственного вывода. Каждая трехмерная модель имеет две разновидности: каркасную (wireframe), без нормалей к поверхности, и сплошную, с тонированием и нормалями. Используйте сплошную версию, если моделируете освещение. Генерирует координаты текстуры только модель чайника (teapot).

void `glutWireSphere`(GLdouble *radius*, GLint *slices*, GLint *stacks*);
void `glutSolidSphere`(GLdouble *radius*, GLint *slices*, GLint *stacks*);
void `glutWireCube`(GLdouble *size*);
void `glutSolidCube`(GLdouble *size*);
void `glutWireTorus`(GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *nsides*, GLint *rings*);

void `glutSolidTorus`(GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *nsides*, GLint *rings*);
void `glutWireIcosahedron`(void);
void `glutSolidIcosahedron`(void);
void `glutWireOctahedron`(void);
void `glutSolidOctahedron`(void);
void `glutWireTetrahedron`(void);
void `glutSolidTetrahedron`(void);
void `glutWireDodecahedron`(GLdouble *radius*);
void `glutSolidDodecahedron`(GLdouble *radius*);
void `glutWireCone`(GLdouble *radius*, GLdouble *height*, GLint *slices*, GLint *stacks*);
void `glutSolidCone`(GLdouble *radius*, GLdouble *height*, GLint *slices*, GLint *stacks*);
void `glutWireTeapot`(GLdouble *size*);
void `glutSolidTeapot`(GLdouble *size*);

Управление фоновыми процессами

Вы можете указать функцию, которая будет выполняться при отсутствии каких-либо событий (например, во время простоя (idle) главного цикла) — `glutIdleFunc()`. Это особенно полезно для продолжительной анимации или другой фоновой обработки.

void `glutIdleFunc`(void (**func*)(void));
Задает функцию *func*, вызываемую по факту отсутствия событий. При передаче в нее значения `NULL` фоновое выполнение блокируется.

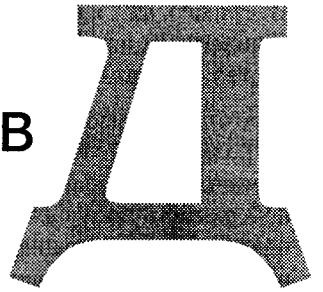
Запуск программ

По завершении всех настроечных действий GLUT-программа входит в цикл обработки событий, `glutMainLoop()`.

void `glutMainLoop`(void);

Начинает цикл обработки событий GLUT, не завершающийся сам по себе. При возникновении событий происходит вызов зарегистрированных функций обратной связи.

Вычисление векторов нормалей



В этом приложении рассказывается, как вычислить векторы нормалей к поверхности. Перед тем как воспользоваться средствами OpenGL для моделирования освещения, вам следует определить нормали, как это описано в главе 5. Тех, кому нужно освежить знания о нормалях и связанных с ними командах OpenGL, отсылаем к разделу «Векторы нормали» главы 2. В данном приложении объясняются тонкости их применения. Оно разбито на следующие основные разделы:

- Нахождение нормалей к аналитическим поверхностям
- Нахождение нормалей к полигональным поверхностям

Поскольку нормали перпендикулярны поверхности, вы можете найти нормаль к конкретной точке поверхности, сначала найдя плоскость, касательную к поверхности в этой точке. Нормаль — это вектор, перпендикулярный плоскости. Например, для идеальной сферы направление нормали совпадает с вектором, направленным из центра сферы к точке ее поверхности. Для других типов поверхностей нормали определяются в зависимости от описания этих поверхностей.

Повторный вызов, сглаживающий искривленные поверхности, аппроксимирует их большим количеством небольших плоских многоугольников. Если векторы, перпендикулярные этим многоугольникам, используются в качестве нормалей, поверхность получается «граненой» (faceted), так как направления нормалей меняются не непрерывно, а дискретно при переходах через границы многоугольников. Однако во многих случаях существует точное математическое описание поверхности, и истинные нормали могут быть вычислены в любой точке. Использование истинных нормалей улучшает воспроизведение, как показано на рис. Д.1. Даже если вы не имеете математического описания, вы можете получить лучшие результаты, чем ячеистые фигуры на рисунке. В двух основных разделах этого приложения говорится о том, как вычислить векторы нормалей в следующих двух случаях:

- **Нахождение нормалей к аналитическим поверхностям.** Объясняет, как воспользоваться имеющимся математическим описанием поверхности.
- **Нахождение нормалей к полигональным поверхностям.** Здесь рассматривается случай, когда в вашем распоряжении имеются только многоугольники, аппроксимирующие поверхность.

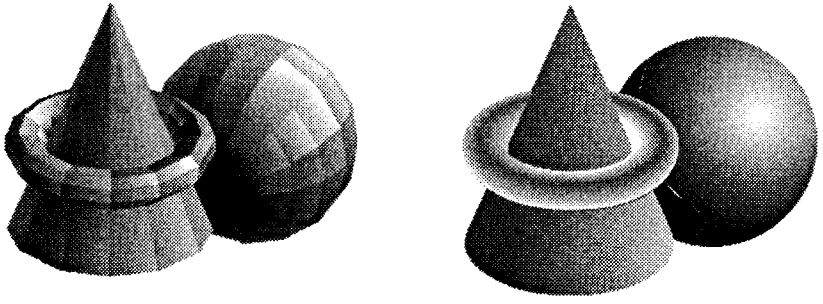


Рис. Д.1. Отображение на основе полигональных нормалей в сравнении с истинными нормалями

Нахождение нормалей к аналитическим поверхностям

Аналитическая поверхность — это гладкая, дифференцируемая поверхность, которая описывается математическим уравнением (или системой уравнений). Во многих случаях легче всего найти нормали к таким поверхностям, если вы имеете описание следующего вида:

$$\mathbf{V}(s, t) = [\ \mathbf{X}(s, t)\ \mathbf{Y}(s, t)\ \mathbf{Z}(s, t)\ ],$$

где s и t принадлежат некоторой области изменения, а **X**, **Y** и **Z** — дифференцируемые функции двух переменных. Для вычисления нормали найдите частные производные

$$\frac{\partial V}{\partial s} \text{ и } \frac{\partial V}{\partial t},$$

которые являются тангенсами углов наклонов векторов к поверхности относительно s- и t-направлений. Векторное произведение

$$\frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t}$$

перпендикулярно обоим векторам и, следовательно, к поверхности. Далее показано, как вычислить векторное произведение двух векторов. (Не забывайте про возможность вырождения векторного произведения к нулевой длине.)

$$[v_x\ v_y\ v_z] \times [w_x\ w_y\ w_z] = [(v_y w_z - w_y v_z)\ (v_x w_z - w_x v_z)\ (v_x w_y - w_x v_y)].$$

Вероятно, вам потребуется нормализовать результирующий вектор. Чтобы нормализовать вектор $[x\ y\ z]$, вычислите его длину:

$$\textit{длина} = \sqrt{x^2 + y^2 + z^2},$$

и поделите каждый компонент вектора на эту длину.

В качестве примера возьмем аналитическую поверхность

$$\mathbf{V}(s, t) = [\ s^2\ t^3\ 3 - st\ ]$$

Отсюда мы имеем

$$\frac{\partial V}{\partial s} = [2s\ 0\ -t],\ \frac{\partial V}{\partial t} = [0\quad 3t^2\ -s] \text{ и } \frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t} = [3t^3\ 2s^2\ 6st^2\ ].$$

Например, когда $s = 1$ и $t = 2$, соответствующая точка поверхности — (1, 8, 1), а вектор (−24, 2, 24) — это перпендикуляр к поверхности в данной точке. Длина вектора равна 34, соответственно, единичный вектор нормали — (−24/34, 2/34, 24/34) = (−0,70588, 0,058823, 0,70588).

Для аналитических поверхностей, описанных неявно, наподобие $\mathbf{F}(x, y, z) = 0$, решение задачи более сложное. Здесь вы должны решить ее относительно одной из переменных, скажем, $z = \mathbf{G}(x, y)$, и привести к явной форме, представленной выше:

$$\mathbf{V}(s, t) = [\ s\ t\ \mathbf{G}(s, t)\ ].$$

Затем продолжите, как описано ранее.

Если вы не можете получить уравнение поверхности в явной форме, воспользуйтесь тем фактом, что вектор нормали задается градиентом

$$\nabla F = \left[\frac{\partial F}{\partial x}\ \frac{\partial F}{\partial y}\ \frac{\partial F}{\partial z}\right],$$

вычисленным для определенной точки (x, y, z). Вычисление градиента дело несложное, но нахождение точки, лежащей на поверхности, труднее. Как пример

неявно заданной аналитической функции рассмотрим уравнение сферы с единичным радиусом и центром в начале координат:

$$x^2 + y^2 + z^2 - 1 = 0.$$

Это значит, что уравнение

$$\mathbf{F}(x, y, z) = x^2 + y^2 + z^2 - 1$$

может быть решено для z в виде

$$z = \pm\sqrt{1 - x^2 - y^2}.$$

Таким образом, нормаль может быть вычислена в явной форме

$$\mathbf{V}(s, t) = [s\ t\ \sqrt{1 - s^2 - t^2}\ ],$$

как описано ранее.

Если бы не удалось получить решение относительно z, можно было бы воспользоваться градиентом

$$\nabla\mathbf{F} = [2x\ 2y\ 2z\ ].$$

В нашем случае найти точку поверхности не сложно — например, (2/3, 1/3, 2/3) принадлежит поверхности. Опираясь на градиент, находим нормаль в этой точке — (4/3, 2/3, 4/3). Вектор единичной нормали — (2/3, 1/3, 2/3), как и ожидалось, имеет координаты, совпадающие с координатами точки..

Нахождение нормалей к полигональным поверхностям

Как упоминалось ранее, часто нужно найти нормали к поверхности, описанной полигонами — многоугольниками, чтобы создать впечатление гладкой, а не граненой поверхности. В большинстве случаев простейший путь сделать это (хотя, может, и не самый эффективный) — вычислить вектор нормали для каждой грани и усреднить соседние нормали. Усреднение нормалей для соседних граней — обычная практика. На рис. Д.2 показана поверхность и ее полигональная аппроксимация. Конечно, если многоугольники представляют точную математически поверхность, а не просто аппроксимированную, как для куба или алмазного резца, усреднение не рекомендуется. Вычислите нормаль к каждой грани, как описано ниже, и используйте это значение для каждой из вершин грани.

Чтобы найти нормаль к плоскому многоугольнику, возьмите любые три вершины, v_1, v_2 и v_3, многоугольника, не лежащие на прямой линии. Векторное произведение

$$[v_1 - v_2] \times [v_2 - v_3]$$

перпендикулярно многоугольнику. (Обычно требуется нормализовать результирующий вектор.) Затем вам нужно усреднить нормали для смежных граней, избегая присвоения излишних весов. Например, на рис. Д.2, где n_1, n_2, n_3 и n_4 — нормали для четырех многоугольников с общей вершиной P, вычислите $n_1 + n_2 + n_3 + n_4$ и нормализуйте результат. (Он может быть лучше, если веса нормалей будут про-

порциональны размерам углов в точке пересечения.) Полученный вектор можно использовать как нормаль в точке P.

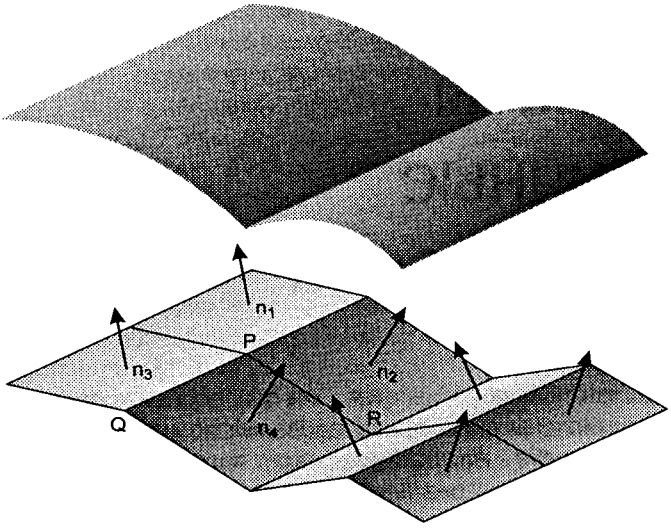
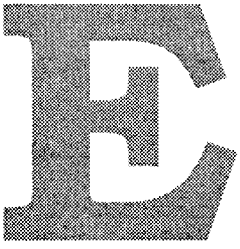


Рис. Д.2. Усреднение векторов нормалей

Иногда требуется подстроить этот метод к текущей ситуации. Так, для границы поверхности (такой как точка Q на рис. Д.2) вам лучше выбрать нормаль, отталкиваясь от знания, как должна выглядеть поверхность. Бывает, что лучше усреднить нормали многоугольников также и на границах. В иных моделях присутствуют гладкие части и острые углы (точка R ребра на рис. Д.2 — характерный пример). В этом случае нормали по обеим сторонам складки не требуют усреднения. Вместо этого многоугольники с одной стороны складки должны быть нарисованы с одной нормалью, а с соседней стороны — с другой.

Однородные координаты и матричные преобразования



В этом приложении затрагивается тема однородных координат. Здесь также перечислены виды преобразований по матрице, используемые для поворота, масштабирования, параллельного переноса, перспективная и ортогональная проекции. Эти понятия вводятся и обсуждаются в главе 3. Заинтересованных в углубленном изучении отсылаем к любой книге по трехмерной компьютерной графике, например Computer Graphics: Principles and Practice, авторов Фоли (Foley), ван Дама (van Dam), Фейнера (Feiner) и Хаггеса (Hughes) (Addison–Wesley, 1990); или к материалам по проективной геометрии — например, The Real Projective Plane, Коксетер Х. М. С. (H. S. M. Coxeter), 2-е издание (Cambridge University Press, 1961). В последующем обсуждении под однородными координатами мы всегда будем понимать трехмерные однородные координаты, хотя в терминологии проективной геометрии это понятие применимо к пространствам любой размерности. Данное приложение разбито на следующие основные разделы:

- Однородные координаты
- Матричные преобразования

Однородные координаты

Команды OpenGL обычно имеют дело с двух- и трехмерными вершинами, но фактически всегда они обрабатываются на внутреннем уровне как трехмерные «однородные» вершины, скрывающие под собой четыре координаты. Каждый вектор-столбец $(x, y, z, w)^T$ представляет однородную вершину, если хотя бы один из элементов ненулевой. Если вещественное число a — не нуль, $(x, y, z, w)^T$ и $(ax, ay, az, aw)^T$ представляют одну однородную вершину. (Это просто похоже на пропорцию: $x/y = (ax)/(ay)$.) Точка трехмерного Евклидова пространства $(x, y, z)^T$ становится однородной вершиной с координатами $(x, y, z, 1.0)^T$, а двухмерная точка $(x, y)^T$ — вершиной $(x, y, 0.0, 1.0)^T$.

Пока значение параметра w отлично от нуля, однородная вершина $(x, y, z, w)^T$ соответствует трехмерной точке $(x/w, y/w, z/w)^T$. Если $w = 0{,}0$, она не соответствует никакой Евклидовой точке, это идеализированная «бесконечно удаленная» точка. Для понимания последнего термина рассмотрим точку (1, 2, 0, 0), заметив, что последовательность точек (1, 2, 0, 1), (1, 2, 0, 0.01) и (1, 2.0, 0.0, 0.0001) соответствует Евклидовым точкам (1, 2), (100, 200) и (10 000, 20 000). Эта последовательность представляет точки, в прогрессии, устремленные в бесконечность по закону $2x = y$. Таким образом, вы можете считать точку (1, 2, 0, 0) «бесконечно удаленной» в данном направлении.

ПРИМЕЧАНИЕ

OpenGL не обрабатывает корректно однородные отсеченные координаты со значениями w < 0. Чтобы ваш код был совместим со всеми системами OpenGL, используйте только неотрицательные значения w.

Преобразование вершин

Преобразования вершин (такие как вращение, смещение (translation), масштабирование и сдвиговая деформация (shearing)) и проекций (таких как перспективная и ортогональная) могут быть все представлены применением соответствующей 4 × 4 матрицы к координатам данной вершины. Если v представляет однородную вершину и $\mathbf{M}$ — 4 × 4 матрица преобразования, то $\mathbf{M}v$ — это образ v после преобразования ее по $\mathbf{M}$. (В программах компьютерной графики обычно используются невырожденные (nonsingular) преобразования — другими словами, матрица `M` может быть инвертирована. Это не факт, но невырожденные преобразования чреваты проблемами.)

После преобразования все вершины отсекаются (clipped) так, чтобы x, y и z попали в диапазон $[-w, w]$ (в предположении, что $w > 0$). Заметьте, что этот диапазон соответствует Евклидову пространству $[-1.0, 1.0]$.

Преобразование нормалей

Векторы нормалей не преобразуются так, как это делается с вершинами или позициями векторов. С точки зрения математики лучше не думать о векторах нор-

малей, а думать о плоскостях, им перпендикулярным. Тогда правила преобразования векторов нормалей описываются как правила для перпендикулярных плоскостей.

Однородная плоскость — это вектор-строка (a, b, c, d), где хотя бы один из элементов a, b, c и d отличен от нуля. Если q — ненулевое вещественное число, (a, b, c, d) и (qa, qb, qc, qd) представляют одну плоскость. Точка $(x, y, z, w)^T$ находится на этой плоскости (a, b, c, d), когда $ax + by + cz + dw = 0$. (Если $w = 1$, получаем стандартную Евклидову плоскость.) Среди элементов (a, b, c, d), представляющих Евклидову плоскость, хотя бы один должен быть ненулевым. Если же они все равны нулю, вектор $(0, 0, 0, d)$ соответствует «бесконечно удаленной» плоскости, содержащей все «бесконечно удаленные» точки.

Если $\mathbf{p}$ — однородная плоскость и v — однородная вершина, утверждение «v лежит на плоскости $\mathbf{p}$» записывается математически как $\mathbf{p}v = 0$, где $\mathbf{p}v$ является обычным произведением матриц. Если $\mathbf{M}$ — невырожденное преобразование (то есть это 4×4 матрица, которой обратна матрица $\mathbf{M}^{-1}$), тогда запись $\mathbf{p}v = 0$ эквивалента выражению $\mathbf{p}\mathbf{M}^{-1}\mathbf{M}\mathbf{v} = 0$, так что $\mathbf{M}v$ лежит в плоскости $\mathbf{p}\mathbf{M}^{-1}$. Отсюда $\mathbf{p}\mathbf{M}^{-1}$ — это образ плоскости при $\mathbf{M}$-преобразовании вершины.

Если вам больше нравится думать о векторах нормали как о векторах, вместо того, чтобы думать о плоскостях, перпендикулярных к ним, будем считать $\mathbf{v}$ и $\mathbf{n}$ векторами, такими, что $\mathbf{v}$ перпендикулярен $\mathbf{n}$. То есть $\mathbf{n}^T\mathbf{v} = 0$. Таким образом, для произвольного невырожденного преобразования $\mathbf{M}$ мы имеем $\mathbf{n}^T\mathbf{M}^{-1}\mathbf{M}\mathbf{v} = 0$, и это означает, что $\mathbf{n}^T\mathbf{M}^{-1}$ — транспонирование преобразованного вектора нормали. Итак, преобразованный вектор нормали — $(\mathbf{M}^{-1})\mathbf{T}\mathbf{n}$. Другими словами, векторы нормалей преобразуются транспонированием матрицы обратного преобразования, которое преобразует точки. Вот закрутили!

Матричные преобразования

Хотя любая невырожденная матрица $\mathbf{M}$ представляет допустимое преобразование проекции, несколько специальных случаев особенно полезны. Эти матрицы описаны в следующих подразделах.

Параллельный перенос

Вызов `glTranslate*`(x, y, z) генерирует матрицу $\mathbf{T}$, где

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & y \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{и} \quad \mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Масштабирование

Вызов `glScale*`(x, y, z) генерирует матрицу **S**, где

$$\mathbf{S} = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{и} \quad \mathbf{S}^{-1} = \begin{bmatrix} \frac{1}{x} & 0 & 0 & 0 \\ 0 & \frac{1}{y} & 0 & 0 \\ 0 & 0 & \frac{1}{z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Заметьте, что $\mathbf{S}^{-1}$ определена, только если x, y и z все ненулевые.

Поворот

Вызов `glRotate*`(a, x, y, z) генерирует матрицу **R**, исходя из следующего:

Пусть $\mathbf{v} = (x, y, z)^T$ и $\mathbf{u} = \mathbf{v}/\|\mathbf{v}\| = (x', y', z')^T$.

Также введем матрицы

$$\mathbf{S} = \begin{bmatrix} x & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix} \quad \text{и} \quad \mathbf{M} = uu^T + (\cos a)(1 - uu^T) + (\sin a)\,\mathbf{S}$$

Тогда

$$\mathbf{R} = \begin{bmatrix} m & m & m & 0 \\ m & m & m & 0 \\ m & m & m & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

где m обозначает элементы 3 × 3 матрицы **M**, определенной страницей выше. Матрица **R** определена всегда. Если $x = y = z = 0$, то **R** — единичная матрица. Вы можете получить обратную **R** матрицу $\mathbf{R}^{-1}$ подстановкой $-a$ вместо a или путем транспонирования.

Команда `glRotate*()` генерирует матрицу для поворота относительно произвольной оси. Часто поворот выполняется относительно одной из осей координат; ему соответствуют следующие матрицы:

`glRotate*`(a, 1, 0, 0): $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

`glRotate*`$(a, 0, 1, 0)$: $\begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$,

`glRotate*`$(a, 0, 0, 1)$: $\begin{bmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

Как и раньше, обратные матрицы получаются транспонированием.

Перспективная проекция

Вызов `glFrustum`(l, r, b, t, n, f) генерирует матрицу **R**, где

$$\mathbf{R} = \begin{bmatrix} \frac{2n}{r-1} & 0 & -\frac{r+1}{r-1} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \text{ и}$$

$$\mathbf{R}^{-1} = \begin{bmatrix} \frac{r-1}{2n} & 0 & 0 & \frac{r+1}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -\frac{(f-n)}{2fn} & \frac{f+n}{2fn} \end{bmatrix}.$$

Матрица **R** определена, пока $l \neq r$, $t \neq b$ и $n \neq f$.

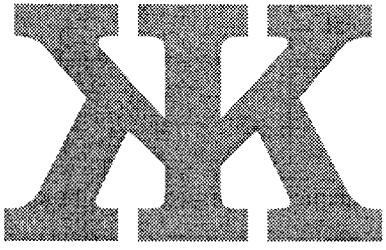
Ортогональная проекция

Вызов `glOrtho`(l, r, b, t, n, f) генерирует матрицу **R**, где

$$\mathbf{R} = \begin{bmatrix} \frac{2}{r-1} & 0 & 0 & \frac{r+1}{r-1} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ и } \mathbf{R}^{-1} = \begin{bmatrix} \frac{r-1}{2} & 0 & 0 & \frac{r+1}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{-2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матрица **R** определена, пока $l \neq r$, $t \neq b$ и $n \neq f$.

Приемы программирования



В этом приложении описаны некоторые приемы и руководящие принципы, которые вы можете найти очень полезными для себя. Имейте в виду, что эти приемы основаны на замыслах разработчиков OpenGL, а не на каком-либо опыте общения с действующими приложениями и реализациями. Данное приложение содержит следующие основные разделы:

- Правильное обращение с OpenGL
- Приемы увеличения производительности
- Советы по применению GLX

Правильное обращение с OpenGL

- Выполняйте проверку ошибок как можно чаще. Вызывайте `glGetError()` хотя бы раз на каждую прорисовку сцены с целью своевременного обнаружения ошибок.
- Не рассчитывайте на определенное поведение реализации OpenGL в случае ошибки — оно может измениться в следующих версиях. Например, OpenGL 1.1 игнорирует матричные операции между `glBegin()` и `glEnd()`, но в последующих версиях это может быть не так. С другой стороны, семантика ошибок OpenGL может измениться по линии совместимости снизу вверх (upward).
- При необходимости сжать всю геометрию до уровня плоскости используйте матрицу проекции. Если используется модельно-видовая матрица, средства OpenGL, оперирующие в системе координат наблюдателя (такие как освещение и определенные приложением плоскости отсечения), могут потерпеть крах.
- Не делайте экстенсивных изменений отдельно взятой матрицы. Так, не стоит анимировать вращение последовательными вызовами `glRotate*()` с инкрементом угла. Лучше используйте `glLoadIdentity()` для инициализации матрицы для каждого кадра и тогда уже вызывайте `glRotate*()` с желаемым значением угла.
- Рассчитывайте на многократное прохождение базы данных изображений для генерации одних и тех же пиксельных фрагментов, только если это поведение гарантируется правилами инвариантности, установленными для гибких реализаций OpenGL. (См. приложение 3 на предмет правил инвариантности.) В противном случае могут быть сгенерированы отличающиеся наборы фрагментов.
- Не ждите сообщений об ошибках при определении списка отображения. Команды внутри списка отображения генерируют ошибки во время его выполнения.
- Размещайте в конусе узла зрения ближайшую плоскость объекта как можно дальше от точки наблюдения для оптимизации операций с буфером глубины.
- Вызывайте `glFlush()` для принудительного выполнения ранее выданных команд OpenGL. Не полагайтесь на `glGet*()` или `glIs*()` для «прочистки» выполнения задержанных операций воспроизведения. Эти команды оптимизируют работу с данными, но они не гарантируют завершение всех ждущих своей очереди команд.
- Отключайте генерирование полутонов для не требующих этого изображений (например, при вызове `glCopyPixels()`).
- Используйте в полной мере буфер накопления. Так, если аккумулируются четыре изображения, масштабируйте каждое в размер одной четвертой от максимально позволенного.
- Если нужна точная двухмерная растеризация, вы должны осторожно задать и ортогональную проекцию, и вершины растеризуемых примитивов. Ортого-

нальная проекция должна быть задана с целочисленными координатами, например: `gluOrtho2D(0, width, 0, height)`, где *width* и *height* — ширина и высота окна просмотра. Чтобы растеризация была предсказуема, для данной проекционной матрицы позиции вершин многоугольников и пикселов изображений должны быть заданы целочисленными координатами. Например, `glRecti(0, 0, 1, 1)` правильно закрашивает левый нижний угол окна просмотра, а `glRasterPos2i(0, 0)` верно позиционирует изображение в левый нижний угол окна просмотра. Тем не менее, вершины точек, вершины линий и позиции битовых образов должны помещаться в позицию, соответствующую половине целого. Так, прямая, рисуемая от (*x1*, 0.5) до (*x2*, 0,5), будет с уверенностью воспроизведена в нижнем ряду пикселов окна просмотра, а точка, нарисованная с координатами (0.5, 0.5), с надежностью окрасит тот же пиксел, что и команда `glRecti(0, 0, 1, 1)`.

Оптимальный компромисс, когда все примитивы принимают целочисленные позиции, то есть гарантируется предсказуемая растеризация, заключается в смещении координат x и y на 0.375, как показано в следующем фрагменте кода. Такой перенос удерживает границы многоугольников и растровых изображений на «безопасной» дистанции от центров пикселов, тогда как перемещаемые вершины прямых практически примыкают к центрам пикселов.

```
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, width, 0, height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.375, 0.375, 0.0);
/* воспроизведение всех примитивов с целочисленными координатами */
```

- Не злоупотребляйте отрицательными значениями координаты w вершины и отрицательными значениями координаты q текстуры. OpenGL может не отсечь такие координаты правильно и допустить ошибку интерполяции при тонировании примитивов.

- Не полагайтесь на точность операций, основываясь на типах данных аргументов команд OpenGL. Например, если вы используете `glRotated()`, не следует надеяться, что конвейер на стадии геометрической обработки обеспечит заданную для чисел с плавающей запятой двойную точность во время операции. Возможно, что значения аргументов `glRotated()` будут преобразованы к другому типу данных перед обработкой.

Приемы увеличения производительности

- Используйте `glColorMaterial()`, если быстро изменяется только одно свойство материала (например, в каждой вершине). Используйте `glMaterial()` для редких изменений или в условиях быстрого изменения нескольких свойств.

- Используйте `glLoadIdentity()` для инициализации матриц, а не загружайте собственную копию единичной матрицы.
- Используйте специальные вызовы для матриц, такие как `glRotate*()`, `glTranslate*()` и `glScale*()`; вряд ли реализованные вами вращение, перенос и масштабирование с помощью `glMultMatrix()` окажутся быстрее.
- Используйте функции запроса при потребности всего лишь в нескольких параметрах состояния для собственных вычислений. Если запрашиваемые параметры принадлежат одной группе, используйте `glPushAttrib()` и `glPopAttrib()` для их сохранения и восстановления.
- Используйте списки отображения для инкапсуляции заведомо «долгоиграющих» изменений состояния.
- Используйте их же для инкапсуляции при визуализации неподвижных объектов, которая будет выполняться многократно.
- Используйте объекты текстур для инкапсуляции данных текстур. Поместите все вызовы `glTexImage*()` (включая мини-текстуры — mipmap), требуемые для полной спецификации текстуры, и связанные `glTexParameter*()` вызовы (устанавливающие свойства текстур) в объект текстуры. Закрепите этот объект текстуры для выбора текстуры.
- Если ситуация позволяет, используйте `gl*TexSubImage()` для замены существующего изображения текстуры или его части, это обходится «дешевле», чем тратить время на удаление изображений и создание новых.
- Если ваша реализация OpenGL поддерживает высокопроизводительный рабочий набор резидентных текстур, попытайтесь сделать все свои текстуры резидентными — то есть попробуйте уместить их в высокопроизводительной памяти. При необходимости сожмите их по размеру или измените разрешение для внутреннего формата. Если такое сжатие приводит, увы, к потере четкости, вы можете присвоить некоторым таким неприемлемым текстурам пониженный приоритет, и тогда при необходимости они будут вытолкнуты из набора в пользу других текстур.
- Используйте вычислители даже для простейших мозаичных разбиений поверхности, чтобы сократить сетевой трафик в клиент-серверном окружении.
- Пользуйтесь единичными нормалями, если это возможно, тем самым избегая издержек на нормализацию посредством `GL_NORMALIZE`. Постарайтесь забыть про `glScale*()` при моделировании освещения, поскольку для него практически всегда проводится нормализация (нужно включить режим `GL_NORMALIZE`).
- Пользуйтесь командой `glShadeModel()` в режиме `GL_FLAT`, если не требуется плавного затушевывания.
- По возможности очищайте весь кадр одним вызовом `glClear()`. Не тратьте процессорное время на очистку маленьких участков буферов; применяйте эту команду для полной или почти полной очистки.
- Эффективнее один раз вызвать `glBegin(GL_TRIANGLES)` для рисования множества независимых треугольников, нежели чем множество раз вызвать

`glBegin(GL_TRIANGLES)` для отдельных треугольников или использовать `glBegin(GL_POLYGON)`. Даже если нужно нарисовать всего один треугольник, задавайте параметр `GL_TRIANGLES`, но не `GL_POLYGON`. Таким же образом используйте один вызов `glBegin(GL_QUADS)` вместо повторных `glBegin(GL_POLYGON)`. И более того, используйте один вызов `glBegin(GL_LINES)` для рисования множества независимых отрезков, а не обращайтесь к `glBegin(GL_LINES)` столько раз, сколько имеется отрезков.

- Некоторые реализации OpenGL используют преимущества хранения данных вершин в массивах вершин. Используйте массивы вершин для сокращения издержек на вызовы функций. В других реализациях можно поднять производительность за счет пакетной обработки или повторной обработки вершин.

- В общем, используйте векторные формы команд для передачи предварительно вычисленных данных, и используйте скалярные формы для передачи значений, вычисляемых незадолго до вызова.

- Не транжирьте впустую ресурсы системы на излишние переключения режимов, такие как установка одного и того же значения цвета перед выводом каждой вершины плоского многоугольника.

- Убедитесь, что заблокировали затратные операции растеризации и операции над фрагментами при отображении и копировании изображений. Ведь если вы попросите, OpenGL применит текстуры даже к растровым данным.

- Если это непринципиально, обрабатывайте многоугольники не в двух режимах — для переднего и заднего фронтов, а в одном.

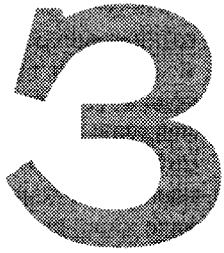
Советы по применению GLX

- Используйте `glXWaitGL()` вместо `glFinish()`, чтобы вынудить команды визуализации в X Window следовать за командами GL.

- Аналогично, лучше использовать `glXWaitX()`, нежели чем `XSync()`, чтобы заставить команды GL выполняться после команд X Window.

Будьте осторожны с `glXChooseVisual()`, так как булевские значения сравниваются точно. Поскольку некоторые реализации не экспортируют объекты визуализации со всеми комбинациями булевских характеристик, вам следует несколько раз вызвать `glXChooseVisual()` с различными булевскими значениями перед тем, как разочароваться в успехе. Например, при отсутствии доступных объектов визуализации с однократной буферизацией проверьте наличие их аналогов (с теми же характеристиками) с двойной буферизацией. Какой-нибудь может оказаться доступен, и это облегчит ваше положение.

OpenGL и инвариантность



OpenGL не дает гарантии, что изображения, произведенные в разных реализациях системы, будут в точности до пиксела соответствовать друг другу. Тем не менее в отдельных случаях для одной и той же реализации удается получить точное соответствие. В этом приложении описаны правила, позволяющие добиться инвариантности при определенных условиях.

Очевидный и самый фундаментальный случай — это повторяемость результатов. Правильно согласованная реализация OpenGL вырабатывает одни и те же результаты каждый раз при выполнении определенной последовательности команд в одинаковых начальных условиях. Хотя такая повторяемость полезна для тестирования и диагностики, для разработчиков программ она часто ничего не значит, поскольку трудно воспроизвести тождественные начальные условия. Например, эти требования не выполняются для повторного отображения сцены после обмена местами рабочего и фонового буферов. Таким образом, одной повторяемости мало, чтобы гарантировать устойчивость для изображения с двойной буферизацией.

Простой и полезный алгоритм, рассчитанный на инвариантность, — это стирание линии посредством ее рисования с цветом фона. Этот алгоритм работает, только если совпадают результаты растеризации линии в одном и том же фрагменте (x, y), полученные как для случая цвета переднего плана, так и для цвета фона. OpenGL требует, чтобы координаты фрагментов, сгенерированных растеризацией, были инвариантны относительно содержимого буфера кадров, задействованных для рисования буферов цвета, матриц, отличных от находящихся в вершине стека, параметров отсечения, всех масок записи, всех значений очистки, текущего цвета, индекса, нормали, координат текстуры и признаков границ, текущего цвета растровых данных, индекса цвета растра и координаты текстуры для растра, а также свойств материала. Дополнительно требуется, чтобы сгенерированные фрагменты были сами одинаковы, с точностью до значений цвета, для различного содержимого буфера кадров, включенных буферов цвета, для матриц не с вершины стека, для других параметров отсечения, масок записи или значений очистки.

Помимо того, OpenGL предлагает, но не требует, чтобы генерация фрагментов была инвариантна относительно режима использования матриц, глубины стека матриц, параметров альфа-теста (отличного от активного), параметров трафарета (отличного от включенного), параметров теста глубины (отличного от активного), параметров смешивания (отличных от активного), логических операций (не тех, которые включены) и параметров хранения пикселов и операций при их передаче. Поскольку инвариантность относительно нескольких включений не рекомендуется, вы должны использовать другие параметры для отключения функций, если требуется инвариантное воспроизведение. Например, для того чтобы воспроизведение было инвариантно как при смешивании, так и без него, установите параметр `GL_ONE` для смешивания и `GL_ZERO` для отключения смешивания, а не вызывайте `glDisable(GL_BLEND)`. Аналогично поступайте для альфа-теста, тестов глубины и трафарета и для всех логических операций.

Наконец, OpenGL требует, чтобы арифметика операций над фрагментами, таких как смешивание и тест глубины, была инвариантна в применении к любой переменной состояния OpenGL, за исключением состояний, имеющих непосредственное отношение к операциям. Так, на арифметику смешивания могут воздействовать только те параметры OpenGL, которые определяют источник и приемник, и отвечают за включение смешивания. Смешивание инвариантно к изменению любых других параметров состояния. Эта инвариантность сохраняется для теста отсечения, альфа-теста, тестов трафарета и глубины, смешивания, генерации полутонов, логических операций и маскирования записи в буферы.

При соблюдении всех этих требований OpenGL может гарантировать, что изображения, полученные с помощью различных буферов цвета, одновременно или по раздельности, с использованием одной последовательности команд, будут идентичны с точностью до пиксела. Это верно как для всех буферов цвета в буфере кадров, так и для неэкранных буферов цвета, но при переходе от буфера кадров к неэкранным буферам инвариантность не гарантируется.

Словарь терминов

GLX — оконный интерфейс OpenGL для X Window System.

IRIS GL — собственная графическая библиотека Silicon Graphics, разрабатывавшаяся в 1982–1992 годы. OpenGL ведет происхождение из IRIS GL.

MIP-текстура (mipmap) — версия с низким разрешением карты текстуры, используемая для текстур геометрических примитивов, экранное разрешение которых отлично от разрешения исходной карты текстуры.

NURBS — неоднородный рациональный B-сплайн (Non-Uniform Rational B-Spline). Это наиболее общий способ задания параметрических кривых и поверхностей. (См. подпрограммы GLU NURBS в главе 12.)

RGBA — красный (Red), зеленый (Green), синий (Blue), альфа- (Alpha) компоненты.

VRML — сокращение от Virtual Reality Modeling Language (язык моделирования виртуальной реальности), объявленный (согласно спецификации VRML) «универсальным описательным языком для имитаторов с множественным участием (multiparticipant)». VRML разработан специально с целью позволить людям путешествовать по трехмерным мирам, расположенным в Мировой Паутине (World Wide Web). Первые версии VRML — это подмножества файлового формата Open Inventor file (OIF) с добавками, дающими возможность передвигаться по ссылкам Интернета.

X Window System — оконная система, используемая на многих компьютерах, оснащенных OpenGL. GLX — это название расширения OpenGL для системы X Window. (См. приложение B.)

Альфа-компонент (alpha) — четвертый компонент цвета. Альфа-компонент никогда не отображается непосредственно, а обычно используется для смешивания цветов. В OpenGL принято, что альфа-компонент соответствует понятию непрозрачности (opacity), обратному понятию прозрачности (transparency), то есть если значение альфа-компонента равно 1.0, это полная непрозрачность, и наоборот, величина альфа-компонента 0.0 означает полную прозрачность.

Анимация (animation) — многократное воспроизведение сцены с плавным изменением точки наблюдения и/или позиций объектов, настолько быстрое, что достигается иллюзия движения. Анимация OpenGL практические всегда выполняется с двойной буферизацией.

Бит (bit) — двоичный разряд. Может принимать только два значения: 0 или 1. Двоичные числа — это конструкции из одного или более битов.

Битовая плоскость (bitplane) — прямоугольный массив битов, один к одному соответствующих пикселам. Буфер кадров представляет собой «колоду» из битовых плоскостей.

Битовый образ (bitmap) — растровое изображение, прямоугольный массив битов. Также примитив, воспроизводимый командой `glBitmap()`, в этом случае данные битового образа используются в качестве маски.

Буфер (buffer) — группа битовых плоскостей, хранящая единый компонент (такой как глубина или зеленый цвет) или один индекс (например, индекс цвета или трафарета). Иногда красный, зеленый, синий и альфа-буферы рассматриваются как единый буфер цвета, а не как отдельные буферы.

Буфер глубины (depth buffer) — память, хранящая значения глубины для каждого пиксела. С целью удаления невидимых поверхностей в буфере глубины фиксируются значения глубины объектов, ближайших к наблюдателю. Сохраненное значение сравнивается со значением глубины каждого вновь поступающего на обработку фрагмента, и перед воспроизведением фрагмент должен подвергаться тесту глубины.

Буфер кадров (framebuffer) — совокупность всех буферов окна или контекста. Иногда в него включается вся память аппаратного графического видеоускорителя.

Буфер накопления (accumulation buffer) — память (битовые плоскости), используемая для накопления серии изображений, сгенерированных в буфере цвета. Использование буфера накопления может существенно улучшить качество изображения, но, соответственно, при этом замедлить время визуализации. Буфер накопления используется для таких эффектов, как глубина резкости, размытие при движении и сглаживание сцены.

Буфер трафарета (stencil buffer) — память (битовые плоскости), используемая для дополнительного пофрагментного тестирования, в паре с *буфером глубины* (depth buffer). Для заливки областей может быть применен тест трафарета, кэпинг (capping) сложных геометрических объектов и наложение полупрозрачных многоугольников.

Векторное произведение (outer product) — процесс, вырабатывающий матрицу из двух векторов. Для матрицы **A**, являющейся произведением векторов **u** и **v**, $\mathbf{A}_{ij} = \mathbf{u}_i\mathbf{v}_j$.

Векторы нормали (normal vectors) — см. *нормаль* (normal).

Вершина (vertex) — точка в трехмерном пространстве.

Видимый объем (view volume) — кусок пространства в оконных координатах, значения которых удовлетворяют следующим трем условиям: $-w \leq x \leq w$, $-w \leq y \leq w$ и $-w \leq z \leq w$. Графические примитивы, выходящие за пределы видимого объема, отсекаются.

Визуализация, воспроизведение изображения (rendering) — преобразование примитивов, заданных в координатах объекта, в изображение в буфере кадров. Визуализация — это первостепенная операция OpenGL, то, ради чего OpenGL существует.

Вогнутость (concave) — Не *выпуклость* (convex).

Воздушная перспектива (depth-cuing) — техника визуализации, при которой цветовое представление частей изображения зависит от расстояния до точки наблюдения.

Выбор (selection) — процесс определения, какой из примитивов пересекает двухмерную область или трехмерный объем, заданные матричными преобразованиями.

Выбор с помощью указателя мыши (picking) — форма *выбора* (selection), при которой используется прямоугольная область (обычно рядом с позицией указателя) с целью выяснить, какой из примитивов выбран.

Выполнение (execute) — команда OpenGL выполняется, когда она вызывается в непосредственном режиме или как часть списка отображения.

Выпуклость (convex) — многоугольник считается выпуклым, если никакая прямая в плоскости многоугольника не пересекает его более чем дважды.

Выпуклый каркас (convex hull) — наименьшая выпуклая область, охватывающая заданную группу точек. В двух измерениях такой каркас образуется растягиванием «резинового обруча» вокруг точек так, чтобы все они попали внутрь «обруча».

Выравнивание относительно окна (window-aligned) — когда говорят про выровненные относительно окна отрезки линий или ребра многоугольников, вкладывают тот смысл, что они параллельны сторонам окна. (В OpenGL окно имеет прямоугольную форму, с горизонтальными и вертикальными границами.) Когда говорят про шаблон многоугольника, подразумевают, что шаблон зафиксирован относительно начала координат окна.

Вычисление (evaluated) — OpenGL-процесс генерации координат вершин объекта и параметров для ранее определенных уравнений Безье.

Гамма-коррекция (gamma correction) — функция, применяемая для корректировки приведения цветов, сохраненных в буфере кадров, в соответствие с нелинейным характером чувствительности глаз (и иногда и монитора) к линейным изменениям интенсивности цвета.

Геометрическая модель (geometric model) — координаты вершин объекта и параметры, его характеризующие. Заметьте, что OpenGL не предписывает определенный синтаксис геометрическим моделям, в отличие от синтаксиса и семантики воспроизведения геометрических моделей.

Геометрический объект (geometric object) — см. *геометрическая модель* (geometric model).

Гистограмма (histogram) — «база данных» распределения значений пикселов в изображении. В процессе гистограммирования подсчитывается количество уникальных значений пикселов.

Глубина (depth) — z-координата окна.

Графический примитив (geometric primitive) — точка, линия и многоугольник.

Группа (group) — каждый пиксел или изображение в памяти клиента представляется группой из одного, двух, трех или четырех элементов. Таким образом, в контексте изображения в памяти клиента группа и пиксел — одно и то же.

Группа атрибутов (attribute group) — набор связанных параметров состояния, которые OpenGL может сохранять или восстанавливать одновременно.

Двойная буферизация (double-buffering) — контексты OpenGL с наличием переднего и заднего буферов цвета являются дважды буферизуемыми. Плавная анимация достигается при отображении только в задний буфер (невизуализуемый) и выполнении обмена содержимым переднего и заднего буферов — их перестановке. См. описание `glutSwapBuffers()` в приложении Г.

Деколь (decal) — метод вычисления значений цвета для приложений, работающих с текстурами. Цвета текстуры замещают цвета фрагмента, или, если включено альфа-смешивание, они смешиваются с цветами фрагмента, используя только значение альфа-компонента.

Деление перспективы (perspective division) — деление x, y и z на w, выполняемое в координатах отсечения (clip coordinates).

Список отображения (display list) — именованный список команд OpenGL. Списки отображения всегда сохраняются на сервере с целью уменьшить сетевой трафик в клиент-серверных средах. За счет предварительной подготовки к отображению содержимое такого списка может быть обработано более эффективно в сравнении с тем же набором команд OpenGL, выполняемых непосредственно. Это чрезвычайно важно для затратных по вычислениям команд, таких как команды NURBS или тесселяция.

Дрожание (jittering) — псевдослучайное отклонение положения объектов в сцене, используемое в паре с буфером накопления для получения спецэффектов.

Задние поверхности (back faces) — См. *поверхности* (faces).

Зазубрины, ступеньки (jaggies) — артефакты зубчатого (aliased) рендеринга. Края таких примитивов выглядят неровно. Например, близкая к горизонтальной линия после визуализации представляет собой набор из горизонтальных отрезков, а не плавную, «непрерывную» линию.

Закрашивание Гуро (Gouraud shading) — плавная интерполяция цветов по площади многоугольника или вдоль отрезка прямой, выполняемая между присвоенными вершинам цветами. Также называется *плавным затенением* (smooth shading).

Зубчатость — см. *ступенчатость (aliasing)*.

Изображение (image) — прямоугольный массив пикселов в памяти клиента или в буфере кадров.

Индекс (index) — отдельное значение, интерпретируемое как абсолютное, а не нормализованное в конкретный диапазон (как компоненты). Индексы цвета — это «имена» цветов, «разыменовываемые» аппаратным обеспечением через карту цветов. Индексы обычно маскируются, а не ограничиваются к диапазону. Например, индекс 0xf7 маскируется к 0x7, когда записывается в 4-битовый буфер (цвета или трафарета). Индексы цвета и трафарета всегда обрабатываются как индексы, а не как компоненты.

Индекс цвета (color index) — номер, представляющий цвет как бы по «имени», а не его фактическое значение. OpenGL обращается с индексами цвета как с «непрерывными» числами (подобными числам с плавающей запятой), выполняя над ним такие операции, как интерполяция и генерация полутонов. Тем не менее цветовые индексы хранятся в буфере кадров практически всегда как целые числа. Индексы с плавающей запятой преобразуются посредством округления к ближайшему целому.

Индексы (indices) — индекс во множественном числе. (Выбор между формами indices или indexes может быть предметом обсуждения, но авторы предпочитают вариант -ices.)

Интерполяция (interpolation) — вычисление промежуточных значений (таких как цвет или глубина) между граничными пикселами (такими как вершины прямоугольников или концы прямой).

Источник направленного света (directional light source) — см. *бесконечно удаленный источник освещения* (infinite light source).

Каркас, проволочный каркас (wireframe) — представление объекта только отрезками линий. Обычно эти отрезки соответствуют ребрам многоугольника.

Карта цветов (color map) — таблица преобразования индексов в RGB-значения, доступные для данного монитора. Каждый индекс цвета читается из буфера цвета, преобразуется в RGB-триаду по карте цветов и отправляется на экран.

Клиент (client) — компьютер, на котором выдаются команды OpenGL. Клиент может быть соединен по сети с другим компьютером, где эти команды выполняются, или же команды могут выдаваться и обрабатываться на одном компьютере. См. также *сервер* (server).

Компоненты (components) — отдельные «непрерывные» (например, вещественные) значения, представляющие интенсивность или количественную характеристику. Обычно нулевые значения компонентов соответствуют минимальной интенсивности, а равные единице — максимальной интенсивности, хотя иногда используются и другие диапазоны. Поскольку значения компонентов интерпретируются как нормализованные, они независимы от действительной размерности. Например, RGB-триада (1, 1, 1) — это белый, невзирая на то, сколько выделяется значению компонента в буфере цвета — 4, 8 или 12 бит. Выходящие за пределы диапазона компоненты обычно ограничиваются нормализованным диапазоном (clamped), а не обрезаются (truncated) и интерпретируются как-либо

иначе. Так, RGB-триада (1.4, 1.5, 0.9) будет ограничена в (1.0, 1.0, 0.9) перед тем, как обновит собой буфер цвета. Значения красного, зеленого, синего, альфа-канала и глубины всегда обрабатываются как компоненты, а не как индексы.

Контекст (context) — полный набор параметров состояния OpenGL. Заметьте, что контекст буфера кадров не является частью состояния OpenGL, а определяет конфигурацию буфера кадров.

Координаты объекта (object coordinates) — система координат, соответствующая объекту до применения к нему любых преобразований OpenGL.

Координаты отсечения (clip coordinates) — система координат, которая «следует» за преобразованиями по матрице проекции и предшествует обработке перспективы. Отсечение видимого объема выполняется в координатах отсечения, но не в заданных приложением (application-specific clipping).

Коррекция перспективы (perspective correction) — дополнительные вычисления координат текстуры, с целью исправления искажений текстур в перспективной проекции.

Лицевые поверхности (front faces) — см. *поверхности* (faces).

Логические операции (logical operation) — булевские математические операции над цветом входящих фрагментов (RGBA или индексы) и значениями цвета (RGBA или индексы), уже помещенными в соответствующее место буфера кадров. Примеры: И (AND), ИЛИ (OR), исключающее ИЛИ (XOR), NAND и INVERT.

Локальный источник света (local light source) — источник освещения с точно заданной позицией. Такой источник испускает радиальный свет. Его другие названия — *точечный* (point) или *позиционный* (positional) источник света. *Прожектор* (spotlight) — специальный вид локального источника света.

Массив вершин (vertex array) — блок с данными вершин (координаты вершины, координаты текстуры, нормали к поверхности, RGBA-цвета, индексы цвета и признаки границ), сохраненный в массиве и впоследствии используемый для задания множества геометрических примитивов одной командой OpenGL.

Материал (material) — свойство поверхности, привлекаемое к расчету ее освещенности.

Матрица (matrix) — двухмерный массив значений. Матрицы OpenGL всегда имеют размеры 4 × 4, в памяти клиента они представлены 1 × 16 одномерными массивами.

Матрица проекции (projection matrix) — матрица 4 × 4, предназначенная для преобразования точек, линий, многоугольников и растровых позиций из системы координат наблюдателя в координаты отсечения.

Матрица текстур (texture matrix) — матрица с размерами 4 × 4, служащая для преобразования координат текстур из своей собственной системы координат в координаты, используемые для интерполяции и подстановки текстур.

Матрица цвета (color matrix) — матрица с размерами 4 × 4, используемая для преобразования RGBA-значений пикселов, которые передаются в команды рас-

ширения Imaging Subset. Матрицы цвета особенно полезны при преобразованиях цветового пространства (для разных моделей представления цвета).

Матрицы (matrices) — матрица во множественном числе. См. *индексы* (indices).

Метод минимакса (minmax) — процесс вычисления максимального и минимального значений пикселов в изображении с помощью расширения Imaging Subset.

Многоугольник (polygon) — почти плоская поверхность, ограниченная ребрами, которые заданы вершинами. Каждый треугольник в сетке треугольников есть многоугольник, как и каждый четырехугольник в четырехугольной сетке. Прямоугольник, который тоже представляет собой случай многоугольника, задается командой `glRect*()`.

Модельно-видовая матрица, макетная матрица (modelview matrix) — матрица с размерами 4 × 4, по которой выполняются преобразования точек, линий, многоугольников и растровых позиций из координат объекта в координаты наблюдателя.

Модуль текстуры (texture unit) — при мультитекстурировании некоторая часть изображений текстур, соответствующая одному шагу обработки отдельного изображения текстуры. Модуль текстуры служит для поддержания состояния текстуры на определенном проходе текстурирования, а именно изображения текстуры, параметры фильтра, среды, генерации координат и стека матриц. Мультитекстурирование заключается в применении последовательности модулей текстуры.

Модуляция (modulate) — метод вычисления значений цвета в приложении, работающем с текстурами, при котором цвета текстуры и фрагментов комбинируются.

Монитор (monitor) — устройство, отображающее изображение в буфере кадров.

Мультитекстурирование (multitexturing) — процесс применения нескольких изображений текстур к одному примитиву. Изображения применяются одно за другим в порядке очереди выполнения операций над текстурами.

Наложение текстур (texture mapping) — процесс применения изображения (текстуры) к примитиву. Наложение текстур часто используется для придания реалистичности сцене. Например, можно применить картинку к многоугольникам, представляющим стену здания.

Невыпуклость (nonconvex) — многоугольник является невыпуклым, когда в плоскости многоугольника находится линия, пересекающая его более чем дважды.

Нормализация (normalized) — чтобы нормализовать вектор нормали, разделите каждый из ее компонентов на квадратный корень из суммы их квадратов. Затем, если представить нормаль в виде вектора, исходящего из вершины координат (nx', ny', nz'), длина этого вектора будет единичной.

$$factor = \sqrt{nx^2 + ny^2 + nz^2};$$

$$nx' = nx / factor;$$

$$ny' = ny / factor;$$

$$nz' = nz / factor.$$

Нормаль (normal) — трехкомпонентное уравнение плоскости, определяющее угловую ориентацию, но не положение, плоскости или поверхности.

Обратная связь (feedback) — механизм OpenGL, позволяющий возвращать информацию о геометрических и растровых примитивах, подлежащих растеризации в окно, обратно приложению. При активном режиме обратной связи пикселы сцены не обновляются.

Фоновое освещение (ambient) — фоновое освещение не имеет определенного направления и равномерно распределяется в пространстве. Фоновое освещение, попадающее на поверхность, поступает со всех направлений. Свет отражается от объекта независимо от положения и ориентации поверхности, его интенсивность одинакова во всех направлениях.

Объект (object) — объектно-координатная модель, воспроизводимая как набор примитивов.

Объект текстуры (texture object) — именованная кэш-память, хранящая данные текстуры, такие как размер массива изображения, связанные MIP-карты и параметры текстуры: ширина, высота, толщина границы, внутренний формат, размерность компонентов, фильтры уменьшения/увеличения, цвет границы и приоритет, присвоенный текстуре.

Однократная буферизация (single-buffering) — контексты OpenGL, не имеющие задних буферов, являются однократно буферизуемыми. Вы можете использовать эти контексты для анимации, но избегайте нарушающих визуальное равновесие «вспышек», когда выполняете рендеринг.

Однородные координаты (homogeneous coordinates) — набор из $n+1$ координат, представляющих точки в n-мерном пространстве проецирования. Точки проективного пространства можно трактовать как точки Евклидова пространства в совокупности с некоторыми точками в бесконечности. Координаты названы однородными, поскольку масштабирование по каждой из них с некоторым ненулевым коэффициентом не изменит самой точки, к которой относятся эти координаты. Однородные координаты полезны в вычислениях проективной геометрии, и тем самым в компьютерной графике, где сцены должны проецироваться в окно.

Окно (window) — подобласть буфера кадров, обычно прямоугольная, все пикселы которой сконфигурированы в буфере одинаково. Контекст OpenGL в один момент времени имеет дело только с одним окном.

Оконные координаты (window coordinates) — система координат окна. Важно различать пикселы, дискретные по природе, и оконные координаты, являющиеся непрерывными. Например, пиксел (0, 0) в левом нижнем углу окна имеет оконные координаты (0.5, 0.5, z), соответствующие центру этого пиксела. Заметьте, что оконные координаты включают в себя глубину — z, тоже являющуюся непрерывным компонентом.

Ортогональная проекция (orthographic) — проекция без вклада перспективы (параллельная), как на некоторых технических рисунках.

Освещение (lighting) — процесс вычисления цвета в вершинах, основанный на источниках света, свойствах материала и цветовых моделях.

Ослабление (attenuation) — свойство освещения, характеризующее, как интенсивность освещения убывает с расстоянием.

Отбраковка (culling) — процесс исключения невидимых передних или задних поверхностей рисуемого многоугольника.

Отражательная способность (shininess) — экспоненциальная функция, связанная с *отраженной составляющей* (specular) освещения. Управление отражением выполняется показателем степени, с которой затухает отраженный свет.

Отраженный свет (specular) — зеркальное отражение характеризуется отражением от всех полированных поверхностей и позицией наблюдателя. Максимальная отражающая способность имеет место, когда угол между наблюдателем и направлением отражения равен нулю. Зеркальные материалы отбрасывают свет с большей интенсивностью в направлении прихода лучей, с затуханием согласно степени экспоненциальной функции (shininess).

Отсечение (clipping) — исключение из рассмотрения частей геометрических примитивов, находящихся снаружи полупространства, образованного плоскостью отсечения. Точки снаружи просто отбрасываются. Части линий или многоугольников, попадающие во внешнее полупространство, не рассматриваются, и вместо них по необходимости генерируются вершины для получения полного примитива в пределах отсеченного полупространства. Геометрические примитивы и текущая позиция растра (когда она задана) всегда отсекаются относительно шести полупространств, определяемых левой, правой, нижней, верхней, ближней и дальней плоскостями видимого объема. Приложения могут задавать дополнительные плоскости отсечения (application-specific clipping), применяемые в системе координат наблюдателя.

Отсечение (scissoring) — тест фрагментов на отсечение. Фрагменты снаружи прямоугольника отсечения отбрасываются.

Отсечение, задаваемое приложением (application-specific clipping) — отсечение примитивов относительно плоскостей в системе координат наблюдателя; плоскости задаются приложением с помощью команды `glClipPlane()`.

Память клиента (client memory) — основная память (хранящая параметры состояния) клиентского компьютера.

Параметры (parameters) — значения, передаваемые в виде аргументов командам OpenGL. Иногда параметры передаются по ссылке на команду OpenGL.

Перестановка байтов (byte swapping) — изменение порядка следования байтов в переменной определенного типа (как правило, целочисленной, то есть `int`, `short` и др.)

Пиксел (pixel) — элемент картинки. Биты для позиции (x, y) во всех битовых плоскостях буфера кадров вместе составляют один пиксел (x, y). Для изображения в памяти клиента пиксел является одной группой элементов. В оконных координатах OpenGL каждый пиксел соответствует области экрана 1.0×1.0. Координаты левого нижнего угла пиксела — (x, y), а правого верхнего угла — $(x + 1, y + 1)$.

Пирамида видимого объема (frustum) — видимый объем сцены после применения перспективы.

Плавное затенение (smooth shading) — См. *закрашивание Гуро* (Gouraud shading).

Плоское затенение (flat shading) — примитивы окрашиваются равномерно одним цветом, в отличие от плавного (smoothly) затенения. См. *закрашивание Гуро* (Gouraud shading).

Поверхности (faces) — стороны многоугольника. Каждый многоугольник имеет две поверхности: переднюю и заднюю, но в окне показывается только одна из них. Узнать, какая из них видна в окне, можно после проецирования многоугольника в окно. Если ребра многоугольника ориентированы по часовой стрелке, видна одна; а если против часовой стрелки — видна другая. Соотнесение поверхностей направлению обхода — задача OpenGL-программиста.

Позиционный источник света (positional light source) — см. *локальный источник света* (local light source).

Полупространства (half-spaces) — плоскость рассекает пространство на два полупространства.

Преобразования (transformations) — трансформации в пространстве. Преобразования OpenGL ограничены проективными преобразованиями с матрицей 4 × 4. К ним относятся вращение, параллельный перенос, масштабирование (непропорциональное) относительно координатных осей, преобразования перспективы и комбинации перечисленного.

Признак границы (edge flag) — логическое значение для вершины, фиксирующее тот факт, что вершина предшествует краевому ребру. Для пометки ребра как не граничного используется команда `glEdgeFlag*()`. Когда многоугольник рисуется в режиме `GL_LINE`, рисуются только краевые ребра.

Примитив (primitive) — точка, линия, многоугольник, битовый образ или изображение. (Обратите внимание: не только точка, а еще и прямая или многоугольник!)

Примитив изображения (image primitive) — битовый образ или изображение.

Прожектор (spotlight) — специальный тип локального источника света, характеризуемый как направлением (куда он указывает), так и положением. Прожектор имитирует световой конус, где интенсивность света изменяется в зависимости от расстояния от центра конуса.

Прямая, линия (line) — прямой участок между двумя вершинами. В отличие от математических линий, прямые OpenGL ограничены по толщине и длине. Полосы-линии составляются из других отрезков, которые тоже являются линиями.

Прямоугольник (rectangle) — четырехугольник, ребра которого через одно параллельны в координатах объекта. Многоугольники задаются командой `glRect*()` всегда как прямоугольники; другие четырехугольники могут быть прямоугольниками.

Рабочий набор (working set) — на компьютерах со специальным аппаратным обеспечением, ускоряющим работу с текстурами, это группа объектов текстур,

резидентных на текущий момент. Производительность работы с текстурами в пределах рабочего набора выше относительно текстур вне этого набора.

Размытие, генерация полутонов (dithering) — техника для наращивания имеющегося диапазона цветов изображения за счет их «пространственного» размывания. Смежные пикселы при этом получают различные значения цвета; при просмотре издали эти цвета кажутся смешанными в один промежуточный цвет. Этот метод напоминает полутоновую печать, применяемую в черно-белых публикациях для достижения оттенков серого.

Размытие при движении (motion blurring) — техника, использующая буфер накопления для имитации размытия, появляющегося на пленке для движущегося объекта или в случае съемки статичного объекта с перемещением фотоаппарата. Без размытия при движении анимация может оказаться отрывистой.

Разъемный фильтр (separable filter) — тип 2D-*фильтра свертки* (convolution filter), который может быть представлен двумя одномерными векторами. Такой фильтр вычисляется как *векторное произведение* (outer product).

Распаковка (unpack) — процесс преобразования пикселов из формата приложения во внутренний формат OpenGL.

Рассеянное освещение (diffuse) — рассеянные освещение и отражение подчиняются направлению распространения света от источника. Интенсивность света, падающего на поверхность, изменяется согласно углу между ориентацией объекта и направлением света. Диффузные материалы равномерно рассеивают свет во всех направлениях.

Расслоение, чередование (interleaved) — метод сохранения массивов вершин, посредством которого разнородные данные (то есть вершины, нормали, координаты текстур и др.) группируются вместе для быстрого извлечения.

Растеризация (rasterization) — преобразование спроектированных точки, линии или многоугольника, или пикселов битового образа или полноцветного изображения во фрагменты, каждый из которых соответствует пикселу в буфере кадров. Заметьте, что все примитивы растеризуются, не только геометрические — точки, линии и многоугольники.

Расширение (extension) — набор средств, функционально дополняющих ядро OpenGL. Расширения могут не поддерживаться отдельными реализациями OpenGL, в отличие от функций ядра, которые поддержаны для каждой реализации. Есть расширения различных типов, в зависимости от того, в каком направлении простирается их функциональность. Общий принцип, что расширения поддерживаются только для реализаций OpenGL от отдельных компаний, названия которых включаются в виде аббревиатуры в имена функций и констант. Если в создании расширения принимают участие несколько компаний, их названия заменяются на буквы `EXT`. Например, `GL_EXT_polygon_offset`. Если расширение OpenGL просмотрено и одобрено ассоциацией OpenGL Architecture Review Board (или ARB, для краткости), его название изменяется, соответственно, еще раз. Пример одобренного ARB расширения — `GL_ARB_multitexture`.

Расширение ARB Imaging Subset — набор расширений OpenGL для обработки прямоугольников пикселов. Операции Imaging Subset включают в себя подстановку цветов по таблице, матричные преобразования цвета, свертки и статистическую обработку изображений.

Режим RGBA (RGBA mode) — контекст OpenGL соответствует режиму RGBA, если буферы цвета хранят красный, зеленый, синий компоненты и альфа-компонент цвета, а не индексы цвета.

Режим индексации цвета (color-index mode) — такой контекст OpenGL, когда буферы цвета хранят индексы цвета, а не красный, зеленый, синий и альфа-компоненты.

Режим локального просмотра (local viewer mode) — режим модели освещения OpenGL, при котором вычисляется точный вектор от вершины к наблюдателю. Из соображений производительности локальный режим не включен, и при вычислении освещения используется приближение к действительному вектору.

Режим непосредственного выполнения (immediate mode) — выполнение команд OpenGL напрямую, а не через список отображения. Какого-либо бита, отвечающего за включение и выключение непосредственного режима, не существует; это имеет отношение к тому, как используется OpenGL.

Резидентная текстура (resident texture) — изображение текстуры, кэшируемое в специальной памяти с высокой производительностью. Если реализация OpenGL не имеет такой памяти, все изображения текстур считаются резидентными.

Самосвечение (emission) — свет от самоизлучающего объекта. Интенсивность эмиссии материала никак не связана с любыми внешними источниками освещения.

Сервер (server) — компьютер, на котором выполняются команды OpenGL. Это может быть другой компьютер, а не тот, где эти команды вызваны. См. *клиент* (client).

Сеть (network) — соединение двух и более компьютеров с целью передачи данных между ними.

Си (C) — язык программирования от Бога.

Си++ (C++) — объектно-ориентированный язык программирования для неверующих в божественную достаточность C.

Сингулярная (вырожденная) матрица (singular matrix) — матрица, которую нельзя обратить. Геометрически такая матрица представляет преобразование, сжимающее точки хотя бы одной линии в одну точку.

Система координат (coordinate system) — в n-мерном пространстве набор n линейно независимых векторов, закрепленных в некоторой точке (называемой началом координат — origin). Группа координат указывает точку в пространстве (или вектор из начал координат) тем, как далеко нужно следовать вдоль каждого вектора для достижения точки (вершины вектора).

Система координат наблюдателя (eye coordinates) — система координат, которая «следует» за преобразованиями по модельно-видовой матрице (modelview) и

предшествует преобразованиям по матрице проекции. Освещение и заданные приложением плоскости отсечения применяются в этой системе координат.

Смешивание (blending) — сведение двух цветовых компонентов в один, обычно выполняемое как их линейная интерполяция.

Смещение многоугольника (polygon offset) — техника для изменения значений глубины многоугольника, где дополнительные геометрические примитивы рисуются с идентичными геометрическими координатами.

Стерео (stereo) — создание впечатления трехмерного изображения посредством вычисления своего изображения для каждого глаза. Стереоизображения требуют специального оборудования, такого как синхронизированные мониторы или специальные очки. Некоторые реализации OpenGL поддерживают стереоэффект за счет расслоения цветовых данных на левый и правый буферы.

Ступенчатость (aliasing) — техника визуализации, при которой пикселам присваивается цвет отображаемого примитива, невзирая на то, покрывает примитив целый пиксел или его часть. Результатом этого являются неровные, ступенчатые края — зазубрины (jaggies).

Таблица подстановки цветов (color-lookup table) — одномерное «изображение», согласно которому замещаются компоненты RGBA для пикселов изображения.

Тексел (texel) — элемент текстуры в памяти текстур. Тексел представляет цвет, примененный к соответствующему фрагменту.

Текстура-посредник (proxy texture) — местозаполнитель для изображения текстуры, используемый для определения, достаточно ли ресурсов для поддержки изображения текстуры с заданными размерами и разрешением внутреннего формата.

Текстуры (textures) — одномерные или двухмерные изображения, используемые для изменения цвета фрагментов, полученных при растеризации.

Текущая матрица (current matrix) — матрица для преобразования координат из одной системы отсчета в другую. В OpenGL имеется три текущие матрицы: модельно-видовая матрица (modelview) преобразует координаты объекта (указанные программистом) в систему координат наблюдателя; матрица перспективы преобразует координаты в системе наблюдателя в отсеченные (clip) координаты; матрица текстур преобразует заданные или сгенерированные координаты текстуры согласно значениям матрицы. Каждая текущая матрица является верхним элементом стека матриц. Каждым из трех стеков матриц можно манипулировать с помощью команд OpenGL.

Текущая позиция растра (current raster position) — позиция в оконных координатах, указывающая положение растеризуемых примитивов изображения. Текущая растровая позиция и другие текущие параметры растра изменяются вызовом `glRasterPos()`.

Тесселяция, мозаичное разбиение (tessellation) — сведение части аналитической поверхности к сетке из многоугольников или части аналитической кривой к последовательности линий.

Тонирование, затенение (shading) — процесс интерполяции цвета в пределах внутренней области многоугольника или на отрезке между его вершинами, выполняемый при растеризации.

Точечный источник света (point light source) — см. *локальный источник света* (local light source).

Точка (point) — точное положение в пространстве, представляемое видимой точкой конечного диаметра.

Точка обзора (viewpoint) — начало координат в системах координат наблюдателя и отсечения, в зависимости от контекста. Когда говорится об освещении, точка наблюдения относится к координатам наблюдателя. При разговоре о проецировании подразумевается система координат отсечения. С типовой матрицей проецирования оба начала координат совпадают.

Треугольник (triangle) — многоугольник с тремя ребрами. Треугольники всегда выпуклые.

Туман (fog) — метод визуализации, позволяющий имитировать такие атмосферные эффекты, как дымка, туман и смог, посредством перетекания цветов объекта в фоновый цвет, основываясь на расстоянии от наблюдателя. Туман также способствует передаче пространственного расположения — глубине восприятия объектов сцены.

Удаление невидимых линий (hidden-line removal) — метод, определяющий, какие части каркасного объекта будут видимы. Линии, составляющие каркас, рассматриваются как ребра прозрачных поверхностей, которые могут заслонить собой другие, дальше отстоящие от наблюдателя ребра.

Удаление невидимых поверхностей (hidden-surface removal) — техника, определяющая, какие части непрозрачного, тонированного объекта, будут видимы, а какие — заслонены другими. Тест координаты на глубину с использованием буфера глубины в качестве хранилища значений глубины является наиболее общим методом удаления невидимых поверхностей.

Удаленный в бесконечность источник света (infinite light source) — направленный источник освещения. Радиально излучаемый им свет падает на все объекты параллельными лучами.

Упаковка (pack) — процесс преобразования цветов пикселов из буфера в запрашиваемый приложением формат.

Уровень детализации (level of detail) — процесс создания множества копий объекта или изображения с различным разрешением. См. также *MIP-текстуры* (mipmap).

Устранение ступенчатости (antialiasing) — техника рендеринга, при которой пикселам присваиваются цвета, основанные на части площади пикселов, которые покрыты воспроизводимым примитивом. При сглаживании уменьшаются или устраняются неровности, характерные для техники зубчатого воспроизведения.

Фильтр свертки (convolution filter) — одномерное или двухмерное изображение, используемое для вычисления взвешенного среднего для пикселов другого изображения.

Фильтрация (filtering) — процесс комбинирования пикселов или текселов, имеющий цель получить версию входного изображения или текстуры с более высоким или более низким разрешением.

Фрагмент (fragment) — фрагменты генерируются в процессе растеризации примитивов. Каждый фрагмент соответствует отдельному пикселу и характеризуется цветом, глубиной и в иных случаях координатами текстур.

Четырехугольник (quadrilateral) — многоугольник с четырьмя ребрами.

Шрифт (font) — группа графически представленных символов, используемая для отображения текстовых строк. Символы могут быть буквами латиницы, математическими символами, азиатскими идеограммами, египетскими иероглифами и т. д.

Штриховка, прерывание (stipple) — одномерный или двухмерный двоичный шаблон, в котором генерация фрагментов для нулевых значений не выполняется. В отношении линий — одномерных объектов — речь идет о прерывании. Многоугольники — двухмерные объекты — штрихуются с фиксированной ориентацией относительно окна.

Элемент (element) — отдельный компонент или индекс.

Ядро (kernel) — см. *свертка ядро* (convolution kernel).

Свертка «ядро» (convolution kernel) — см. *фильтр свертки* (convolution filter).

Яркость (luminance) — воспринимаемая глазом освещенность (brightness) поверхности. Часто связывается со средневзвешенной суммой красного, зеленого и синего компонентов.

Алфавитный указатель

A

AGL, 34, 562
- инициализация, 563
- использование
  - Apple-шрифтов, 564
- обработка ошибок, 565
- перестановка буферов, 564
- подключение, 35
- прототипы, 565
- управление визуализацией, 563

aglChoosePixelFormat(), 565
aglConfigure(), 566
aglCopyContext(), 565
aglCreateContext(), 565
aglDescribePixelFormat(), 565
aglDescribeRenderer(), 565
aglDestroyContext(), 565
aglDestroyPixelFormat(), 565
aglDestroyRendererInfo(), 565
aglDevicesOfPixelFormat(), 565
aglDisable(), 566
aglEnable(), 566
aglErrorString(), 566
aglGetCurrentContext(), 565
aglGetDrawable(), 566
aglGetError(), 566
aglGetInteger(), 566
aglGetVersion(), 565
aglGetVirtualScreen(), 566
aglIsEnabled(), 566
aglNextPixelFormat(), 565
aglNextRendererInfo(), 565
aglQueryRendererInfo(), 565
aglResetLibrary(), 566
aglSetCurrentContext(), 565
aglSetDrawable(), 565
aglSetFullScreen(), 566
aglSetInteger(), 566
aglSetOffScreen(), 565
aglSetVirtualScreen(), 566
aglSwapBuffers(), 566
aglUpdateContext(), 565
aglUseFont(), 566
ARB, 498
Architecture Review Board, 498
ASCII-код, 253, 267–268

B

bilinear sampling, 346
bitmap, 262
bitplane, 388
blending, 403

C

capping, 401
ChoosePixelFormat(), 572
CreateDIBitmap(), 573
CreateDIBSection(), 573
cube map texture, 369

D

decal, 355, 397, 508
DeleteObject(), 573
depth textures, 383
depth-cuing, 226
DescribePixelFormat(), 572
dithering, 403

E

equalization, 295

F
foreshortening, 121
framebuffer, 389

G
GetVersion(), 572
GetVersionEx(), 572
glAccum(), 405
glActiveTexture(), 372
glActiveTexture*(), 372
glAlphaFunc(), 396
glAreTexturesResident(), 351
glArrayElement, 60, 79
glBegin, 27, 57
glBindTexture(), 318, 348
glBitmap(), 262–263
glBitmap*(), 265
  пример использования, 268
glBlendColor, 203
glBlendEquation, 204
glBlendFunc, 202
glBlendFuncSeparate, 203
glCallList, 60
glCallList(), 250
glCallLists, 60
glCallLists(), 252, 267
glClear, 27, 48
glClear(), 392
glClearAccum, 48
glClearAccum(), 392
glClearColor, 27, 48, 158
glClearColor(), 392
glClearDepth, 48
glClearDepth(), 392
glClearIndex, 48, 158, 229
glClearIndex(), 392
glClearStencil, 48
glClearStencil(), 392
glClientActiveTexture*(), 375
glClipPlane, 134
glColor, 60, 79, 156
glColor*(), 266, 380
glColor3, 156
glColor3f, 27, 49
glColor4, 156
glColorMask(), 394
glColorMaterial, 189
glColorPointer, 77
glColorSubTable(), 298
glColorTable(), 295, 298
glColorTableParameter*(), 296
glCompressedTexImage1D(), 334
glCompressedTexImage2D(), 334
glCompressedTexImage3D(), 334
glCompressedTexSubImage1D(), 334
glCompressedTexSubImage2D(), 334
glCompressedTexSubImage3D(), 334
glConvolutionFilter1D(), 303
glConvolutionFilter2D(), 299
glConvolutionParameter*(), 303
glCopyColorSubTable(), 298
glCopyColorTable(), 297
glCopyConvolutionFilter1D(), 303
glCopyConvolutionFilter2D(), 302
glCopyPixels, применение, 519
glCopyPixels(), 270, 276–277, 282, 287, 294
glCopyTexImage*(), 278–279, 294
glCopyTexImage1D(), 328
glCopyTexImage2D(), 322, 327
glCopyTexSubImage*(), 278–279, 294
glCopyTexSubImage1D(), 328
glCopyTexSubImage3D(), 331
glCullFace, 68
glDeleteLists(), 252, 267
glDeleteTextures(), 350
glDepthFunc(), 402
glDepthMask, 211
glDepthMask(), 394
glDepthRange, 127
glDepthRange(), 264
glDisable, 61
glDisableClientState, 76
glDrawArrays, 84
glDrawBuffer(), 392–393
glDrawElements, 80
glDrawPixels
  применение, 519
glDrawPixels(), 270, 275–276, 282, 293–294, 399
  пример использования, 275
  форматы пикселов и типы данных, 273
glDrawRangeElements, 46, 83
glEdgeFlag, 60, 72, 79
glEdgeFlagPointer, 77
glEdgeFlagv, 72
glEnable, 61
glEnable(), 295
glEnableClientState, 76
glEnd, 27, 58
glEndList(), 248
glEvalCooord, 60
glEvalCoord*(), 442, 445–447
glEvalMesh*(), 446, 448
glEvalPoint, 60
glFeedbackBuffer(), 470, 486–4 7
glFinish, 51

glFlush, 27, 51
glFog, 228
glFogCoord, 60, 79, 232
glFogCoordPointer, 77
glFrontFace, 68
glFrustum, 121
glFrustum(), 412, 588
glGenLists(), 248, 267
glGenTextures(), 318, 347
glGetBooleanv, 62
glGetBooleanv(), 265, 390
glGetClipPlane(), 528
glGetColorTable(), 298, 528
glGetColorTableParameter*(), 298, 528
glGetCompressedTexImage(), 528
glGetConvolutionFilter(), 528
glGetConvolutionParameter*(), 300, 528
glGetDoublev, 62
glGetError(), 496, 528
glGetFloatv, 62
glGetFloatv(), 264, 266
glGetHistogram(), 307, 528
glGetHistogramParameter*(), 528
glGetIntegerv, 62
glGetIntegerv(), 266, 286, 323, 372, 390–391, 396, 398, 476
glGetLight*(), 383, 528
glGetMap*(), 528
glGetMaterial*(), 528
glGetMinmax(), 309, 528
glGetMinmaxParameter*(), 528
glGetPixelMap*(), 528
glGetPointerv, 62
glGetPolygonStipple(), 528
glGetSeparableFilter(), 528
glGetString(), 497, 528
glGetTexEnv*(), 528
glGetTexGen*(), 528
glGetTexImage(), 278, 528
glGetTexLevelParameter*(), 323, 528
glGetTexParameter*(), 529
glHint, 214
glHistogram(), 306
glIndex, 60, 157
glIndex*(), 266
glIndexMask(), 394
glIndexPointer, 77
glInitNames(), 469–471
glInterleavedArrays, 85
glIsEnabled 62
glIsEnabled), 398
glIsList(), 252, 529
glIsTexture.), 348, 529
glLight, 171
glLightModel, 182
glLineStipple, 64
glLineWidth, 63
glListBase(), 252, 267
glLoadIdentity, 107
glLoadIdentity(), 103
glLoadMatrix, 107
glLoadName(), 470–471
glLoadTransposeMatrix, 108
glLogicOp(), 404
glLookAt, 117
glMap*(), 444, 446–447
glMapGrid*(), 446, 448
glMaterial, 60, 185
glMatrixMode, 106
glMatrixMode(), 304
glMinmax(), 308–309
glMultiDrawArrays, 46, 84
glMultiDrawElements, 46, 82
glMultiTexCoord, 60
glMultiTexCoord*(), 372, 374–375
glMultMatrix, 107
glMultTransposeMatrix, 108
glNewList(), 248
glNormal, 60
glNormal3, 73, 79
glNormalPointer, 77
glOrtho, 27, 124
glOrtho(), 588
glPassThrough(), 487–488
glPixelMap*(), 283, 285–286
  параметры, 285
glPixelStore, 69
glPixelStore*(), 263, 280, 291, 331
  параметры, 280
glPixelTransfer*(), 283–284, 286, 304–305
  параметры, 284
glPixelZoom(), 276, 287
  пример использования, 287
glPointParameterf, 234
glPointParameterfv, 234
glPointSize, 62
glPolygonMode, 67
glPolygonOffset, 237
glPolygonStipple, 69
glPopAttrib, 88
glPopClientAttrib, 89
glPopMatrix, 132
glPopMatrix(), 469
glPopName(), 470–471
glPrioritizeTextures(), 352
glPushAttrib, 88
glPushClientAttrib, 89
glPushMatrix, 132

glPushMatrix(), 469
glPushName(), 469–471
glRasterPos*(), 262–264, 266, 270, 471
glReadBuffer(), 273, 392–393
glReadPixels(), 270–271, 277, 282, 294
  форматы пикселов и типы данных, 273
glRect, 55
glRenderMode(), 469–471, 486
glResetHistogram(), 306, 308
glResetMinmax(), 309
glRotate, 111, 116
glRotate*(), 587
glSampleCoverage, 224
glScale, 112
glScale*(), 436, 587
glScissor(), 396
glSecondaryColor, 60, 157
glSecondaryColor3, 79
glSecondaryColor3*(), 381
glSecondaryColorPointer, 77
glSelectBuffer(), 469, 472
glSeparableFilter2D(), 302
glStencilFunc(), 395, 397
glStencilMask(), 394–395
glStencilOp(), 398
glTexCoord, 60, 79
glTexCoord*(), 319, 356–357
glTexCoordPointer, 77
glTexEnv*(), 340, 353, 376
glTexGen*(), 363, 367, 370, 385
glTexImage*(), 278, 294
glTexImage1D(), 327
glTexImage2D(), 318–319, 323, 337, 369
glTexImage3D(), 329
glTexParameter*(), 319, 342, 345, 352, 360–361
glTexSubImage*(), 278, 294, 352
glTexSubImage1D(), 328
glTexSubImage2D(), 325, 335
glTexSubImage3D(), 330
glTranslate, 111, 116
glTranslate*(), 410, 586
GLU, 23, 34, 54, 418, 432
  подключение, 34
gluBeginCurve(), 453, 461
gluBeginPolygon(), 431–432
gluBeginSurface(), 453, 460
gluBeginTrim(), 464
gluBuild1DMipmapLevels(), 344
gluBuild1DMipmaps(), 343
gluBuild2DMipmapLevels(), 344
gluBuild2DMipmaps(), 343
gluBuild3DMipmapLevels(), 344
gluBuild3DMipmaps(), 343
gluCheckExtension(), 499
gluCylinder(), 433–435
gluDeleteNurbsRenderer(), 457
gluDeleteQuadric(), 433
gluDeleteTess(), 419, 430–431
gluDisk(), 433–435
gluEndCurve(), 454, 461
gluEndPolygon(), 431–432
gluEndSurface(), 454, 460
gluEndTrim(), 464
gluErrorString(), 496
gluGetNurbsProperty(), 459, 529
gluGetString(), 419, 498, 529
gluGetTessProperty(), 427, 529
gluLoadSamplingMatrices(), 459
gluLookAt, 117
gluLookAt(), 103
gluNewNurbsRenderer(), 453, 457
gluNewQuadric(), 432–433
gluNewTess(), 419–420, 431
gluNextContour(), 431–432
gluNurbsCallback(), 453, 459, 461
gluNurbsCallbackData(), 462
gluNurbsCurve(), 453, 461
gluNurbsProperty(), 453, 457
gluNurbsSurface(), 453, 460
gluOrtho2D, 124
gluPartialDisk(), 433–434, 436
gluPerspective, 122
gluPerspective(), 384
gluPickMatrix(), 475–476
gluProject, 145
gluPwlCurve(), 464
gluQuadricCallback(), 433
gluQuadricDrawStyle(), 433–434
gluQuadricNormals(), 433–434
gluQuadricOrientation(), 432, 434
gluQuadricTexture(), 433–434
gluScaleImage(), 321
gluSphere(), 433–435
GLUT, 34, 36, 574
  загрузка карты цвета, 577
  задание режима отображения, 575
  запуск программ, 37, 578
  инициализация, 575
  подключение, 35
  создание окна, 36, 575
  трехмерные объекты, 38, 577
  управление окнами и событиями, 36, 576
  фоновые процессы, 578
glutCreateWindow, 36
glutCreateWindow(), 576

glutDisplayFunc, 37
glutDisplayFunc(), 576
gluTessBeginContour(), 428, 432
gluTessBeginPolygon(), 423, 428, 432
gluTessCallback(), 419–421, 431–432
gluTessEndContour(), 428, 432
gluTessEndPolygon(), 428, 432
gluTessNormal(), 427, 430
gluTessProperty(), 419, 424
gluTessVertex(), 428–429, 431
glutIdleFunc(), 578
glutInit, 36
glutInit(), 575
glutInitDisplayMode, 36
glutInitDisplayMode(), 575
glutInitWindowPosition, 36
glutInitWindowPosition(), 575
glutInitWindowSize, 36
glutInitWindowSize(), 575
glutKeyboardFunc, 38
glutKeyboardFunc(), 576
glutMainLoop, 37
glutMainLoop(), 578
glutMotionFunc, 38
glutMotionFunc(), 577
glutMouseFunc, 38
glutMouseFunc(), 576
glutPostRedisplay, 37
glutPostRedisplay(), 577
glutReshapeFunc, 38, 51
glutReshapeFunc(), 576
glutSawpBuffers, 41
glutSetColor, 36, 158, 160, 197
glutSetColor(), 577
glutSolidCone(), 578
glutSolidCube, 39
glutSolidCube(), 577
glutSolidDodecahedron(), 578
glutSolidIcosahedron(), 578
glutSolidOctahedron(), 578
glutSolidSphere, 39
glutSolidSphere(), 577
glutSolidTeapot(), 578
glutSolidTetrahedron(), 578
glutSolidTorus(), 578
glutWireCone(), 578
glutWireCube, 39
glutWireCube(), 577
glutWireDodecahedron(), 578
glutWireIcosahedron(), 578
glutWireOctahedron(), 578
glutWireSphere, 39
glutWireSphere(), 577
glutWireTeapot(), 578
glutWireTetrahedron(), 578
glutWireTorus(), 577
gluUnProject, 143
gluUnProject4, 145
glVertex, 27, 56, 60, 79
glVertex*(), 264
glVertexPointer, 77, 79
glViewport, 126
glWindowPos*(), 262, 264, 471
GLX, 34, 556
  инициализация, 557
  обработка событий, 559
  очистка поверхностей, 560
  перестановка буферов, 560
  подключение, 35
  прототипы, 560
    устаревшие, 562
  синхронизация выполнения, 559
  управление визуализацией, 561
glXChooseFBConfig(), 560
glXChooseVisual(), 562
glXCopyContext(), 561
glXCreateContext(), 562
glXCreateGLXPixmap(), 562
glXCreateNewContext(), 561
glXCreatePbuffer(), 561
glXCreatePixmap(), 561
glXCreateWindow(), 561
glXDestroyContext(), 561
glXDestroyGLXPixmap(), 562
glXDestroyPbuffer(), 562
glXDestroyPixmap(), 562
glXDestroyWindow(), 562
glXGetClientString(), 560
glXGetConfig(), 390, 562
glXGetCurrentContext(), 561
glXGetCurrentDisplay(), 561
glXGetCurrentDrawable(), 561
glXGetCurrentReadDrawable(), 561
glXGetFBConfigAttrib(), 561
glXGetFBConfigs(), 560
glXGetProcAddress(), 561
glXGetSelectedEvent(), 561
glXGetVisualFromFBConfig(), 561
glXIsDirect(), 561
glXMakeContextCurrent(), 561
glXMakeCurrent(), 562
glXQueryContext(), 561
glXQueryExtension(), 560
glXQueryExtensionsString(), 560
glXQueryServerString(), 560
glXQueryVersion(), 560
glXSelectEvent(), 561
glXSwapBuffers, 41

glXSwapBuffers(), 561
glXUseXFont(), 562
glXWaitGL(), 561
glXWaitX(), 561

H
hit records, 469

I
imaging pipeline, 276
Imaging Subset, 293

J
jittering, 406

L
least recently used, LRU, 352

M
mapping, 285
Microsoft Windows, 34, 569
mipmap, 336
MIP-отображение, 335
MIP-текстуры, 335
mosaicing, 340
multisampling, 220

N
Non-Uniform Rational B-Spline, 440
NURBS, 23, 440, 453
  и тесселяция, 461
NURBS-объекты, 457
  обработка ошибок, 459
  создание NURBS-кривой или поверхности, 460
  создание и удаление, 457
  управление свойствами, 457

O
OpenGL, 23
OpenGL Utility Library, GLU, 418
OS/2 Warp, 566

P
PGL, 34, 566
  инициализация, 567
  использование
    логических шрифтов OS/2, 568
    палитр индексов цвета, 568
  перестановка буферов, 568
  подключение, 35
  прототипы, 568
  синхронизация выполнения, 568
  управление визуализацией, 567
pglChooseConfig(), 568
pglCopyContext(), 569
pglCreateContext(), 569
pglDestroyContext(), 569
pglGetCurrentContext(), 569
pglGetCurrentWindow(), 569
pglGrabFrontBitmap(), 569
pglIsIndirect(), 569
pglMakeCurrent(), 569
pglQueryCapability(), 568
pglQueryConfigs(), 568
pglQueryVersion(), 568
pglReleaseFrontBitmap(), 569
pglSelectColorIndexPalette(), 569
pglSwapBuffers(), 569
pglUseFont(), 569
pglWaitGL(), 569
pglWaitPM(), 569
picking, 475
pixel-transfer operations, 283
point sampling, 346
proxies, 298

R
reference plane, 363
rendering, 26
RGB, 149
RGBA, 151
  туман, 229

S
selection hit, 469
SetPixelFormat(), 572
shadow map, 383
SwapBuffers(), 573
sweep selection, 485

T
table lookup, 285
tessellation, 418
texture units, 371
trilinear sampling, 346

V
vertices, 26, 52

W
WGL, 34, 569
  визуализация OpenGL в битовый образ, 571
  доступ к расширениям OpenGL, 571
  инициализация, 570
  использование шрифтов, 572
  обращение к функциям OpenGL, 570

WGL (*продолжение*)
перестановка буферов, 571
подбор палитры цветов, 571
подключение, 35
прототипы, 572
синхронизация выполнения, 571
управление визуализацией, 570
wglCopyContext(), 572
wglCreateContext(), 572
wglCreateLayerContext(), 572
wglDeleteContext(), 572
wglDescribeLayerPlane(), 572
wglGetCurrentContext(), 572
wglGetCurrentDC(), 572
wglGetLayerPaletteEntries(), 573
wglGetProcAddress(), 572
wglMakeCurrent(), 572
wglRealizeLayerPalette(), 573
wglSetLayerPaletteEntries(), 573
wglShareLists(), 572
wglSwapLayerBuffers(), 573
wglUseFontBitmaps(), 573
wglUseFontOutlines(), 573
winding number, 425
wireframe, 427
working set, 347

X

X Window, 41, 390, 556

Z

z-буфер, 163, 391

А

Акели, Курт, 405
альфа-компонент, 149
альфа-тест, 396
аналитическая поверхность, 580
анимация, 39

Б

байты
перестановка, 282
битовая плоскость, 26, 47, 151, 388
битовый образ, 261–262
и переменные состояния, 266
отображение, 262, 265
цвет, 266
буфер, 48
выбор буфера для записи и чтения, 392
глубины, 210
очистка, 48
буфер выбора, 471
буфер глубины, 48, 391
маскирование, 394
очистка, 392
буфер кадров, 26, 389
задание фильтра свертки, 301
запись пикселов, 275
копирование пикселов, 276
создание таблицы цветов, 297
создание текстуры, 326
буфер маски
удаление невидимых линий, 514
буфер накопления, 48, 391, 405
имитация фотографической глубины резкости, 411
очистка, 392
размытие при движении, 411
сглаживание сцены, 406, 415
управление, 405
буфер трафарета, 48, 391
маскирование, 394
очистка, 392
рисование закрашенных вогнутых многоугольников, 509
буфер цвета, 48, 149, 390
двойная буферизация, 390
маскирование, 394
очистка, 392

В

вектор ориентации, 117
векторное произведение, 302
векторный шрифт, 253
векторы нормали, 73
вершины, 26, 52
преобразование, 585
видеобуфер, 26
видеопамять, 271
видимый объем, 101
Вильямс, Ланс, 336
воздушная перспектива, 226
вторичный свет, 184
вторичный цвет, 60, 380
выбор, 468
буфер выбора, 471
и двойная буферизация, 502
и составные имена, 479
инициализация, 469
рекомендации по написанию программ, 484
с помощью указателя мыши, 475
с учетом глубины, 481
стек имен, 470
выбор части иерархического объекта сцены, 479

выборка, 220
вычислители, 441
двухмерные, 446
запуск, 447
определение, 447
определение равноотстоящих координат, 448
и текстуры, 451
контрольные точки, 442
многочлен Бернштейна, 444
одномерные, 442
запуск, 445
определение, 444
определение равноотстоящих координат, 446

Г

гамма-коррекция, 151
Гарднер, Мартин, 517
геометрический примитив, 26
гистограмма, 306
глубина, 163
глубина текстур, 383
градиентная заливка, 158

Д

данные
выравнивание в памяти, 282
двойная буферизация, 40, 153, 155, 390, 502
двухмерные фильтры свертки, 299
разъемные, 302
деколь, 508
Дирихле области, 516
диск, 435
диффузия изображений, 501, 507

З

захват, 485

И

игра «Жизнь», 517
идентификация версии GLU, 498
идентификация версии OpenGL, 497
идентификация расширений, 498
иерархия списков отображения, 250
изображение, 261, 270
диффузия, 501, 507
зеркальное отражение, 287
команды управления, 270
масштабирование с фильтрацией, 519
минимаксная статистика, 308
перестановка, 519
растажение и сжатие, 286
изображение (*продолжение*)
рисование с учетом глубины, 516
создание устойчивых изображений, 508
статистическая информация, 306
трансформация, 503
элементы изображения, 279
изображения
конвейерная обработка, 276
имитация аэрографа, 519
имитация видео, 519
имитация фотографической глубины резкости, 405
индекс цвета, 149
источник света, 170
движущийся, 178
комбинирование, 176
направленный, 173
неподвижный, 177
позиционный, 173
создание, 171

К

квадратичные объекты
создание и удаление, 433
управление атрибутами, 434
квадратичные поверхности, 432
квадратичные примитивы
диск, 435
сфера, 435
цилиндр, 435
Килгард, Марк, 19, 34, 557, 574
комплект обучающих программ, 20
Конвей, Джон, 517
конвейер рисования, 30
конвейеризация обработки изображений, 276
таблицы цветов, 295
контрольные точки, 442
координаты
нормализованные, 102
однородные, 585
оконные, 102
туман, 231
усеченные, 101
кривая Безье, 441
кубические текстуры, 369
кэпинг, 401

Л

Лессер, Дитер, 441
линии, 26, 53
прерывистые, 64
логические операции, 404

М

маскирование буферов, 394
маскирование области экрана
неправильной формы, 399
массив идентификаторов, 469
масштабирование
изображений, 286
текстур, 321
масштабирование по матрице, 112, 587
материал
цвет, 166
матрица
модельно-видовая, 108
текущая, 106
стек, 130
текстур, 381
матрица преобразования
текущая, 103
матрица проекций, 105
матрица цвета, 304
матричные преобразования, 586
масштабирование, 587
ортогональная проекция, 588
параллельный перенос, 586
перспективная проекция, 588
поворот, 243, 587
Мегахед, Эйб, 511
минимакс, 308
многоугольник, 53
отклонение, 236
многоугольники
выпуклые, 54
лицевые, 67
оборотные, 68
простые, 54
фактурные, 69
многочлен Берштейна, 444
множественное отображение, 336
модель
полутоновая, 156
цветовая, 156
модель закрашивания, 158
модель освещения, 167
выбор, 182
модельно-видовая матрица, 104
мозаичное представление, 418
мультивыборка, 220
покрытие, 223
мультитекстурирование, 77, 371

Н

наложение теней, 512
нахождение перекрывающихся
областей, 511
невидимые поверхности
удаление, 163
непосредственный режим, 31, 46, 240, 248
нормали к аналитическим
поверхностям, 580
нормали к полигональным
поверхностям, 582
нормаль, 73
к поверхности, 92
нормализованная, 74
преобразование, 585
текущая, 73
Нэйт, Робинс, 20

О

области Дирихле, 516
обработка ошибок, 495
обратная связь, 420, 486, 489
данные, определяемые пользователем,
423
и NURBS, 461
использование маркеров, 488
массив обратной связи, 487
объект визуализации, 557
объекты текстур, 347
объем видимости, 121
одиночная буферизация, 155
одномерные фильтры свертки, 303
однородные координаты, 585
окно просмотра, 125
округлые точки, 63, 215, 234, 507
операции над фрагментами, 33, 395
альфа-тест, 396
дрожание, 406, 415
клиширование, 403
логические операции, 404
размытие, 152, 403
смешивание, 201, 403
тест глубины, 220, 402–403
тест на отсечение, 395
тест трафарета, 397
операции при передаче пикселов, 283
опорная плоскость, 363
ортогональная проекция, 105, 123, 588
освещение
двухстороннее, 183
излучаемый цвет, 166
излучение, 187
коэффициент ослабления, 174
математика, 193
общее фоновое, 182, 165
отраженный или зеркальный
свет, 166
прожекторы, 174

освещение (*продолжение*)
рассеянное, 166
режим индексации цвета, 196
математика, 197
фоновое, 165
очистка буферов, 48, 392

П

палитра цветов, 149
параллельный перенос по матрице, 586
параметры состояния, 29, 61, 528
и списки отображения, 257
первичный свет, 184
перекрытие полупрозрачных многоугольников, 402
перспективная проекция, 105, 121, 588
Пигл, Лес, 441
пиксел, 26
чтение в оперативную память, 271
пикселы
буфер глубины, 391
буфер накопления, 391
буфер трафарета, 391
буфер цвета, 390
внутренний формат в текстурах, 353
запись в буфер кадров из оперативной памяти, 275
копирование внутри буфера кадров, 276
операции при передаче данных, 283
производительность операций, 293
размер отводимой памяти, 390
режимы хранения, 280
табличное преобразование, 285
упаковка и распаковка, 279
форматы для упакованных данных, 273
чтение и отображение прямоугольных областей, 289
плавное затенение, 415
плоскости отсечения, 134
поверхности второго порядка, 432
поворот по матрице, 243, 587
полигон, 26, 53
полупрозрачность, 500
порядок следования байтов, 280
послойное отображение, 505
правило намотки, 425
преобразование вершин, 585
преобразование нормалей, 585
преобразования
видовые, 103, 115
вращение, 111
имитационные, 142
преобразования (*продолжение*)
масштабирование, 112
модельные, 104
модельно-видовые, 108
обратные, 142
окна просмотра, 125
перемещение, 111
проблемы, 128
проекционные, 104
проецирования, 120
прерывистые линии, 64
применение текстурирования, 515
пример
NURBS-поверхность, 454
автоматическое генерирование координат текстур, 364
воспроизведение сцены с источником света в точке наблюдения, 384
выбор, 472
выбор мышью с учетом глубины, 481
выбор с помощью указателя мыши, 476
генерация координат кубических текстур, 370
гистограмма, 307
дрожание объектов сцены, 407
дрожание с помощью ортогональной проекции, 410
замещение части текстуры, 325
имитация глубины резкости, 413
инициализация модулей текстуры, 373
интерполирующие функции комбинирования, 379
использование двухмерного фильтра свертки, 300
использование матрицы цвета, 305
использование правила намотки, 425
использование таблицы цветов, 296
использование текстуры-посредника, 324
использование теста трафарета, 399
использования векторных шрифтов, 253
квадратичные объекты, 436
кривая Безье для четырех контрольных точек, 442
минимальное и максимальное значения пикселов, 309
определение многоугольника тесселяции, 429
освещенные и затененные поверхности Безье, 449
параметры состояния, 258

пример (*продолжение*)
переключение шаблонов штриховки, 259
поверхность Безье, 447
прикрепление объектов текстур, 349
применение MIP-текстур, 337
применение битовых образов, 262
применение вычислителей для текстур, 451
применение функций обратной связи NURBS-тесселятор, 463
регистрация и применение функций обратной связи NURBS-тесселятора, 462
регистрация функций обратной связи для тесселяции, 421
режим обратной связи, 489
рисование, копирование и масштабирование пикселов, 287
сглаживание сцены, 408
создание составных имен при выборе, 479
создание стека имен, 470
текстура при рисовании шахматной доски, 317
трехмерное текстурирование, 329
усечение NURBS-поверхности, 465
функции обратной связи для вершин и объединения, 422
примитивы, 23
битовый образ, 262
изображение, 270
квадратичные, 435
линия, 52
многоугольник, 52
точка, 52
программы
aaindex.c, 218
aargb.c, 216
accanti.c, 410
accpersp.c, 407–408
alpha.c, 208
alpha3D.c, 211
bezcurve.c, 442
bezmesh.c, 449
bezsurf.c, 447
blendeqn.c, 204
checker.c, 317
clip.c, 135
colormat.c, 190
colormatrix.c, 305
colortable.c, 296
combiner.c, 379
программы (*продолжение*)
convolution.c, 300
cube.c, 102
cubemap.c, 370
dof.c, 413
double.c, 42
drawf.c, 262
feedback.c, 489
fog.c, 226
fogcoord.c, 232
fogindex.c, 230
font.c, 268
hello.c, 37
histogram.c, 307
image.c, 275, 287
light.c, 168
lines.c, 65
material.c, 188
minmax.c, 309
mipmap.c, 337
model.c, 113
movelight.c, 179
multisamp.c, 221
multitex.c, 373
mvarray.c, 83
pickdepth.c, 481
picksquare.c, 476
planet.c, 138
pointp.c, 235
polyoff.c, 238
polys.c, 70
quadric.c, 436
robot.c, 140
select.c, 472
shadowmap.c, 384
smooth.c, 158
stencil.c, 399
stroke.c, 254
surface.c, 454
surfpoints.c, 462–463
tess.c, 421–422, 429
tesswind.c, 425
texgen.c, 364
texsub.c, 325
textbind.c, 349
texture3d.c, 329
texturesurf.c, 451
torus.c, 241
trim.c, 465
unproject.c, 143
varray.c, 78
проекция
ортогональная, 105
перспективная, 105
прозрачность, 397

производительность
при выводе пикселов, 293
пространственное разрешение, 403

P

рабочий набор, 347
равноотстоящие координаты, 446
размытие при движении, 411
растеризация, 25, 33, 150
растровый объект
текущие координаты, 264
расширения
AGL, 562
GLX, 556
Imaging Subset, 293
PGL, 566
WGL, 569
идентификация, 498
режим
RGBA, 149, 151
индексированные цвета, 149, 153, 196
переключение, 155
режимы передачи пикселов
установка, 284
режимы хранения пикселов, 280
для трехмерных текстур, 331
рендеринг, 26
рисование закрашенных вогнутых многоугольников, 509
Робинс, Нэйт, 115, 119, 125, 142, 181, 347, 356–357, 362, 382
свойства материала, 189
Роджерс, Дэвид, 432
Россиньяк, Ярек, 511

C

самосвечение, 187
свертки, 298
масштабирование и отклонение, 303
обработка границ, 304
создание из буфера кадров, 301
свойства материала
определение, 185
сглаживание, 213
многоугольники, 224
мультивыборка, 220
режим RGBA, 215
режим индексации цвета, 218
сглаживание символов, 505
сглаживание сцены, 406, 415
система координат
локальная, 110
наблюдателя, 101
фиксированная, 110
смешивание цветов
выражения, 204
источник, 202
получатель, 202
трехмерное, 210
составные имена, 479
список отображения, 31, 46, 240–241
выполнение, 250
допустимые команды, 249
и параметры состояния, 257
и шрифты, 266
иерархический, 250
именование, 247
индексация, 252
инкапсуляция изменений, 258
конец, 243, 248
начало, 243, 248
составной, 252
стек имен, 469
создание, 470
стек матриц текстур, 381
сфера, 435
сферические текстуры, 367

T

таблицы подстановки, 285
цвета, 295
таблицы цветов
внутренний формат, 296
задание, 295
замещение данных, 297
извлечение данных, 298
создание из буфера кадров, 297
таблицы-посредники, 298
табличное преобразование пикселов, 285
тексел, 312
текстуры, 312
MIP-отображение, 335
автоматическое генерирование MIP-текстур, 343
внутренний формат, 320
загрузка сжатых текстур, 334
задание параметров обработки текстуры, 361
замещение одномерных текстур, 328
и вычислители, 451
координаты текстур, 356
автоматическая генерация, 362
координата q, 382
подбор координат, 357
установка, 356
копирование, 326
кубические, 369

текстуры (*продолжение*)
масштабирование, 321
модули текстур, 371–372
модули текстуры
установка текущего модуля, 372
мультитекстурирование, 371
автоматическая генерация координат, 375
задание координат текстур, 374
обработка границ, 335
объекты текстур, 347
именование, 347
приоритеты, 352
связывание, 349
создание, 348
удаление, 350
одномерные, 327
описание, 319
переопределение, 324
переопределение сжатых текстур, 334
переопределение трехмерной текстуры, 330
плиточное размещение, 335
порядок наложения, 315
применение вторичного цвета, 380
размножение и усечение, 359
обработка границ, 359
резидентные, 351
запрос статуса, 351
с учетом глубины, 383
сжатие изображений, 333
складывание мозаики, 340
создание двухмерной текстуры, 322
создание из буфера кадров, 326
создание одномерных текстур, 327
создание трехмерных текстур, 329
стек матриц текстур, 381
стратегии резидентности, 351
сферические, 367
текстуры-посредники, 323
теневые карты, 383
создание, 383
трехмерные, 329
увеличение и миниатюризация, 345
управление уровнем детализации, 341
уровень детализации, 337, 339
допустимые значения, 342
установка текущей функции текстурирования, 376
фильтрация, 344
билинейная выборка, 346
выборка по точке, 346
трехлинейная выборка, 346
формат и тип данных, 320
текстуры (*продолжение*)
функции текстур, 353
аддитивное смешивание, 355
комбинирование функций, 375
модуляция цвета, 355
эффект наложения окружения, 367
текущая позиция растра, 263
текущий контекст, 248
теневая карта, 383
тесселяция, 418
и NURBS, 461
нормали, 427
обратная совместимость, 431
описание многоугольника, 428
правила намотки в конструктивной блочной геометрии, 425
правило намотки, 425
производительность, 430
процедуры обратной связи, 420
свойства, 424
запрос значений, 427
создание объекта разбиения, 419
удаление объекта разбиения, 430
тест глубины, 220, 402
установка функции сравнения, 402
тест маски
перекрытие полупрозрачных многоугольников, 402
штриховка, 402
тест на отсечение, 395
тест трафарета, 397
кэпинг, 401
тест на отсечение, 396
Тиллер, Уэйн, 441
точка, 26, 53
трансформация изображений, 503
туман, 225
координаты, 231
режим RGBA, 229
режим индексации цвета, 229
уравнения, 228

У

угловой коэффициент глубины, 237
удаление невидимых линий, 513
указатель, 485
упакованные данные
форматы пикселов, 273

Ф

Фарин, Джеральд, 441
фильтр свертки
двухмерный, 299
фильтрация текстур, 344

фоновый свет, 165
фрагмент, 150
функции обратной связи
  begin, 420, 462
  color, 462
  combine, 420
  edgeFlag, 420
  end, 420, 462
  error, 420
  normal, 462
  texCoord, 462
  vertex, 420, 462

Х

Хосчек, Джозеф, 441
Хэберли, Пауль, 405

Ц

цвет
  битового образа, 266
  вторичный, 380
    без освещения, 381
    отраженный, 381
  вычисление, 149
  индекс, 149
  карта цветов, 149, 153
  минимальное и максимальное значения, 308
цвет (*продолжение*)
  модуляция текстурой, 355
  определение
    в режиме RGBA, 156
    режим индесации цвета, 157
  переводные картинки, 397
  представление компонентов цвета, 279
  распределение значений в изображении, 306
  смешивание цветов, 149
  стек матриц цвета, 305
  таблицы цветов, 295
цветовая палитра, 153
цветовой буфер, 149
цветовосприятие, 147
цилиндр, 435

Ш

Шнейдер, Бенгт-Олаф, 511
шрифт
  описание, 268
шрифты
  и списки отображения, 266
  сглаживание символов, 505

Э

элемент изображения, 279

OpenGL 4-е издание

Руководство по программированию

OpenGL — это мощный программный интерфейс, применяемый для получения высококачественных, программно генерируемых изображений и интерактивных приложений, использующих двух- и трехмерные объекты, а также растровые изображения. Четвертое издание книги предоставляет всеобъемлющую и исчерпывающую информацию как о языке OpenGL, так и о его библиотеке инструментов, описывает новые возможности и наиболее важные приложения, одобренные OpenGL Architecture Review Board (ARB). Здесь вы найдете подробное описание заложенной в OpenGL функциональности и множества базовых методов компьютерной графики, таких как построение и воспроизведение трехмерных моделей, интерактивный просмотр объектов с различных точек наблюдения, использование тонирования, освещения и эффектов текстурирования. Представлено углубленное описание дополнительных методов компьютерной графики, включая наложение текстур, сглаживание, «туман» и имитацию других атмосферных эффектов, а также сплайны, конвейерную обработку изображений. Внимание также уделено и другим ключевым темам, таким как повышение производительности программ, расширения OpenGL и создание кросс-платформных приложений.

Четвертое издание книги было дополнено в соответствии с последними обновлениями языка OpenGL, который теперь поддерживает наложение «кубических» текстур, мультитекстурирование, многовыборочное сглаживание, текстурирование с учетом глубины и теней, дополнительные пользовательские режимы текстурирования. Представлены фрагменты программных расширений от ARB и много новых примеров программ.

Тема: **Программированиие/OpenGL** | Уровень пользователя: **опытный**

ПИТЕР®

197198, Санкт-Петербург, а/я 619
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 712-27-05, piter@kharkov.piter.com

ISBN 5-94723-827-6

9 785947 238273

www.piter.com — вся информация о книгах и веб-магазин