Лабораторная работа №8

«Фигуры и геометрические преобразования в Three.js»

Оглавление

Отрисовка различных фигур

Three.js содержит большой набор геометрических фигур, которые можно использовать в своей сцене. Примеры отрисовки различных фигур показаны в программе `geometries.html`. Рассмотрим подробнее некоторые типы фигур и их свойства.

Прямоугольный параллелепипед

`THREE.BoxGeometry` – это очень простая трехмерная геометрия, которая позволяет создавать прямоугольный параллелепипед, задав его ширину, высоту и глубину. Эти три свойства также являются обязательными при создании нового куба, например:

```
new THREE.BoxGeometry(10,10,10);
```

В следующей таблице приведены все свойства, которые можно определить для прямоугольного параллелепипеда:

Свойство	**Описание**
`width`	Ширина прямоугольного параллелепипеда (размер вдоль оси *x*).
`height`	Высота прямоугольного параллелепипеда (размер вдоль оси *y*).
`depth`	Глубина прямоугольного параллелепипеда (размер вдоль оси *z*).
`widthSegments`	Число частей для дискретизации по ширине. Значение по умолчанию: `1`.
`heightSegments`	Число частей для дискретизации по высоте. Значение по умолчанию: `1`.
`depthSegments`	Число частей для дискретизации по глубине. Значение по умолчанию: `1`.

Сфера

С помощью `THREE.SphereGeometry` можно создавать сферические геометрии. Для создания простой сферы используется команда:

```
new THREE.SphereGeometry();
```

Следующие свойства можно использовать для настройки того, как будет выглядеть сферическая геометрия:

Свойство	**Описание**
`radius`	Радиус сферы. Значение по умолчанию: `50`.
`widthSegments`	Количество сегментов, которые будут использоваться по долготе. Большее количество сегментов даст более гладкую поверхность. Значение по умолчанию: `8`, минимальное значение: `3`.
`heightSegments`	Количество сегментов, которые будут использоваться по широте. Значение по умолчанию: `6`, минимальное: `2`.
`phiStart`	Угол, с которого начинается рисование сферы по долготе. Он может принимать значения от 0 до 2π. Значение по умолчанию: `0`.
`phiLength`	Угловой размер, отсчитываемый от `phiStart`, на который будет нарисована сфера по долготе. Значение 2π приведет к отрисовке полной сферы, а π/2 нарисует незамкнутую четвертинку сферы. Значение по умолчанию: 2π.
`thetaStart`	Угол, с которого начинается рисование сферы по широте. Он может принимать значения от 0 до π. Значение по умолчанию: `0`.
`thetaLength`	Угловой размер, отсчитываемый от `thetaStart`, на который будет нарисована сфера по широте. Значение π приведет к отрисовке полной сферы, а π/2 нарисует верхнюю половину сферы. Значение по умолчанию: π.

Цилиндр

Цилиндрические геометрии можно создавать с помощью объекта `THREE.CylinderGeometry`. По умолчанию, конструктор этого объекта не содержит обязательных аргументов, т.е. обычный цилиндр можно создать, вызвав:

```
new THREE.CylinderGeometry();
```

Чтобы изменить внешний вид цилиндра по умолчанию, используются следующие свойства:

Свойство	**Описание**
`radiusTop`	Радиус верхнего основания цилиндра. Значение по умолчанию: `20`.
`radiusBottom`	Радиус нижнего основания цилиндра. Значение по умолчанию: `20`.
`height`	Высота цилиндра. Значение по умолчанию: `100`.
`radialSegments`	Количество сегментов по угловой координате. Значение по умолчанию: `8`. Чем больше сегментов, тем более гладким будет цилиндр.
`heightSegments`	Количество сегментов по высоте цилиндра. Значение по умолчанию: `1`.
`openEnded`	Отключает создание торцевых граней цилиндра. Значение по умолчанию: `false`.
`thetaStart`	Угол, с которого начинается рисование сферы, по умолчанию = 0 (положение на три часа)
`thetaLength`	Угловой размер, отсчитываемый от `thetaStart`, на который будет нарисован цилиндр. По умолчанию установлено значение 2*Pi, что соответствует полному цилиндру.

Конус

`THREE.ConeGeometry` во многом аналогичен `THREE.CylinderGeometry`. Он использует все те же свойства, за исключением того, что позволяет вам устанавливать только один радиус вместо отдельных значений `radiusTop` и `radiusBottom`:

Свойство	**Описание**
`radius`	Радиус основания конуса. Значение по умолчанию — 20.
`height`	Высота цилиндра. Значение по умолчанию: `100`.
`radialSegments`	Количество сегментов по угловой координате. Значение по умолчанию: `8`. Чем больше сегментов, тем более гладким будет цилиндр.
`heightSegments`	Количество сегментов по высоте цилиндра. Значение по умолчанию: `1`.
`openEnded`	Отключает создание торцевых граней цилиндра. Значение по умолчанию: `false`.
`thetaStart`	Угол, с которого начинается рисование сферы, по умолчанию = 0 (положение на три часа)
`thetaLength`	Угловой размер, отсчитываемый от `thetaStart`, на который будет нарисован цилиндр. По умолчанию установлено значение 2*Pi, что соответствует полному цилиндру.

Тор

Для создания простого тора с помощью объекта `THREE.TorusGeometry` не требуется никаких обязательных аргументов. В следующей таблице перечислены аргументы, которые можно указать при создании этой геометрии:

Свойство	**Описание**
`radius`	Радиус тора r_{axial} (см. лекции). Значение по умолчанию: `100`.
`tube`	Радиус трубки r (см. лекции). Значение по умолчанию: `40`.
`radialSegments`	Количество сегментов по углу φ (см. лекции). Значение по умолчанию: `8`.
`tubularSegments`	Количество сегментов по углу θ (см. лекции). Значение по умолчанию: `6`.
`arc`	Размер по углу θ (см. лекции). Значение по умолчанию: 2π (полный круг).

Многогранник

Для создания многогранника необходимо указать вершины и грани. Например, мы можем создать простой тетраэдр так:

```
const vertices = [1, 1, 1,
                  -1, -1, 1,
                  -1, 1, -1,
                  1, -1, -1];
const indices = [2, 1, 0,
                 0, 3, 2,
                 1, 3, 0,
                 2, 3, 1];
new THREE.PolyhedronGeometry(vertices, indices, radius, detail)
```

Пример отрисовки этой фигуры содержится в файле `polyhedron.html`.

Когда вы создаете многогранник, вы можете передать следующие четыре свойства:

Свойство	**Описание**
`vertices`	Точки, составляющие многогранник.
`indices`	Грани, которые необходимо создать из вершин.
`radius`	Размер многогранника. По умолчанию равен `1`.
`detail`	С помощью этого свойства вы можете добавить к многограннику дополнительную детализацию. Если вы установите значение `1`, каждый треугольник в многограннике будет разделен на 4 меньших треугольника. Если вы установите значение `2`, каждый из этих 4 меньших треугольников будет снова разделен на 4 меньших треугольника и так далее.

Трубка

THREE.TubeGeometry создает трубку, которая вытягивается вдоль 3D-сплайна. Вы задаете путь, используя последовательность вершин, а THREE.TubeGeometry создает на их основе трубку. Код, необходимый для создания трубки, очень прост:

```
const points = ... // array of THREE.Vector3 objects

const tubeGeometry = new TubeGeometry(

      new THREE.CatmullRomCurve3(points),

      tubularSegments,

      radius,

      radiusSegments,

      closed

)
```

Пример отрисовки этой фигуры содержится в файле tube.html.

Сначала нам нужно задать массив вершин (переменная points) типа THREE.Vector3. Затем нам нужно определить гладкую кривую через определенные нами точки. Мы можем сделать это, просто передав массив вершин конструктору THREE.CatmullRomCurve3.

Перечислим все аргументы THREE.TubeGeometry:

Свойство	**Описание**
path	Путь, по которому должна следовать трубка.
tubularSegments	Количество сегментов, используемых по длине трубки. Значение по умолчанию: 64. Чем длиннее путь, тем больше сегментов следует указывать.
radius	Радиус трубки. Значение по умолчанию — 1.
radiusSegments	Количество сегментов, которые будут использоваться для аппроксимации округлой формы трубки. Значение по умолчанию — 8. Чем больше это значение, тем более круглой будет выглядеть трубка.
closed	Если для этого параметра установлено значение true, начало и конец трубки будут соединены. Значение по умолчанию false.

Параметрическая поверхность

С помощью `THREE.ParametricGeometry` вы можете создать поверхность на основе параметрического уравнения. Параметры поверхности *u* и *v* должны изменяться в диапазоне от 0 до 1.

Рассмотрим пример:

```
const radialWave = (u, v, optionalTarget) => {
      const result = optionalTarget || new THREE.Vector3()
      const r = 20
      const x = Math.sin(u) * r
      const z = Math.sin(v / 2) * 2 * r + -10
      const y = Math.sin(u * 4 * Math.PI) + Math.cos(v * 2 *Math.PI)
      return result.set(x, y, z)
}
const geom = new THREE.ParametricGeometry(radialWave, 120, 120);
```

Пример отрисовки этой фигуры содержится в файле `parametric-geometries.html`.

Аргументы, которые можно передавать в `THREE.ParametricGeometry`, перечислены в следующей таблице:

Свойство	**Описание**
`function`	Функция, которая определяет положение каждой вершины на основе передаваемых значений *u* и *v*.
`slices`	Количество частей, на которые следует разделить диапазон значений *u*.
`stacks`	Количество частей, на которые следует разделить диапазон значений *v*.

Если, например, мы установим для `slices` значение `5`, а для stacks — `4`, функция будет вызываться со следующими значениями:

```
u:0/5, v:0/4
u:1/5, v:0/4
u:2/5, v:0/4
u:3/5, v:0/4
u:4/5, v:0/4
u:5/5, v:0/4
u:0/5, v:1/4
u:1/5, v:1/4
...
u:5/5, v:3/4
u:5/5, v:4/4
```

Таким образом, чем больше эти значения, тем больше вершин будет считаться и тем более гладкой будет созданная поверхность.

Геометрические преобразования

Когда мы используем геометрии Three.js, то не нужно вручную определять вершины и грани фигур. Three.js самостоятельно создает сетку, содержащую нужную геометрию и один или несколько материалов. После создания сетки, она добавляется на сцену и визуализируется. Есть несколько свойств, которые можно использовать, чтобы изменить положение и ориентацию сетки на сцене. Мы рассмотрим следующий набор свойств и методов:

Свойство/Метод	**Описание**
`position`	Определяет положение объекта на сцене.
`rotation`	Задает вращение объекта вокруг любой из его осей. Three.js также предоставляет специальные функции для вращения вокруг стандартных осей: `rotateX()`, `rotateY()` и `rotateZ()`.
`rotateOnAxis(axis, angle)`	Задает вращение объекта вокруг заданной оси.
`scale`	Это свойство позволяет масштабировать объект вокруг его осей *x*, *y* и *z*.
`translateX(amount)`	Перемещает объект на указанную величину по оси *x*.
`translateY(amount)`	Перемещает объект на указанную величину по оси *y*.
`translateZ(amount)`	Перемещает объект на указанную величину по оси *z*.
`translateOnAxis(axis, distance)`	Перемещает на определенное расстояние вдоль заданной оси.
`visible`	Определяет видимость объекта.

Если открыть файл `transformations.html` в браузере, то будет видно раскрывающееся меню, в котором можно изменять соответствующие параметры.

Свойство position

Начнем со свойства `position`. Мы уже изменяли это свойство у камеры. Теперь с его помощью мы устанавливаем координаты *x*, *y* и *z* у фигуры. С помощью них задается положение относительно родительского объекта. Поскольку обычно мы добавляем объект на сцену, то родителем является сама сцена. Мы можем установить свойство `position` объекта тремя разными способами. Мы можем установить каждую координату напрямую:

```
cube.position.x=10;

cube.position.y=3;

cube.position.z=1;
```

Мы также можем установить их все сразу, как показано ниже:

```
cube.position.set(10,3,1);
```

Свойство `position` – это объект `THREE.Vector3`. Таким образом, мы также можем сделать следующее, чтобы установить значения свойств этого объекта:

```
cube.position=new THREE.Vector3(10,3,1).
```

Свойство rotation

Рассмотрим теперь свойство `rotation`. С помощью него можно задать поворот объекта вокруг одной из его осей. Вы можете установить это значение тремя разными способами:

```
cube.rotation.x = 0.5*Math.PI;

cube.rotation.set(0.5*Math.PI, 0, 0);

cube.rotation = new THREE.Vector3(0.5*Math.PI,0,0);
```

Метод rotateOnAxis (axis : Vector3, angle : Float)

Выполняет поворот с помощью заданного кватерниона:

`axis` – нормированный вектор, задающий ось поворота в системе координат объекта;

`angle` – угол поворота в радианах.

Метод applyQuaternion (quaternion : Quaternion)

Выполняет поворот с помощью заданного кватерниона.

Свойство scale

Следующее свойство – `scale`. С помощью него можно масштабировать объект вдоль определенной оси.

Методы translate

Далее рассмотрим функции перемещения. Они также, как и `position`, позволяют изменять положение объекта, но вместо определения абсолютного положения, мы определяем, куда объект должен переместиться относительно его текущего положения. Например, у нас есть сфера, которая добавляется к сцене, и ее положение установлено на `(1,2,3)`. Затем мы перемещаем фигуру по оси *x*: `translateX(4)`. Теперь ее позиция будет `(5,2,3)`. Если мы хотим вернуть объект в исходное положение, мы вызываем: `translateX(-4)`. В программе `transformations.html` есть вкладка меню под названием translate. Здесь можно поэкспериментировать с этой функцией. Установите значения компонент *x, y* и *z* вектора перемещения и нажмите кнопку translate. Вы увидите, как объект перемещается в новую позицию на основе этих трех значений.

Свойство visible

Последнее свойство, которое можно изменять в меню в правом верхнем углу – это свойство `visible`. Если вы нажмете на одноименный пункт меню, то увидите, что фигура становится невидимой. Когда вы нажмете на него в следующий раз, фигура вновь станет видимой.

Пример: анимация вращающегося куба и отскакивающего мяча

В программе `8.html` в функции `renderScene` добавлен код, который вращает красный куб вокруг всех его осей. Для этого мы увеличиваем свойство `rotation` для всех трех осей на `0.02` каждый раз, когда вызывается функция `renderScene`.

Далее мы рисуем плоскость. Это делается в два этапа. Сначала мы определяем, какие размеры будет иметь прямоугольная плоскость с помощью команды `THREE.PlaneGeometry(60,20)`. В данном случае она имеет ширину `60` и высоту `20`. Нам также нужно сообщить Three.js, как эта плоскость будет выглядеть (например, задать ее цвет и прозрачность). В Three.js мы делаем это, создавая объект материала. Мы создадим матовый материал (`THREE.MeshLambertMaterial`) с цветом `0xAAAAAA`. Затем мы объединим эти два объекта в объект `Mesh` с именем `plane`. Прежде чем мы добавим плоскость в сцену, нам нужно

поставить ее в нужное положение. Мы делаем это, сначала поворачивая её на `90` градусов вокруг оси *x*, а затем мы определяем её положение на сцене, используя свойство `position`.

Схожим образом на сцену добавляются куб и сфера.

Теперь добавим точечный источник света в сцену. `THREE.SpotLight` освещает нашу сцену с позиции (`spotLight.position.set(-40, 40, -15)`).

Рендеринг теней требует больших вычислительных мощностей, и по этой причине тени по умолчанию отключены в Three.js. Однако включить их очень просто.

Первое изменение, которое нам нужно сделать, это сообщить рендереру, что нам нужны тени. Мы делаем это, устанавливая для свойства `shadowMapEnabled` значение `true`. Далее нам нужно явно определить, какие объекты отбрасывают тени и на какие объекты тени накладываются. В нашем примере мы хотим, чтобы сфера и куб отбрасывали тени на плоскость земли. Мы делаем это, устанавливая соответствующие свойства для этих объектов.

Отскакивающий мяч

Чтобы отобразить отскакивающий мяч, мы снова добавляем пару строк кода в нашу функцию `renderScene`.

В кубе мы изменяли свойство `rotation`; для сферы мы будем изменять ее свойство `position`. Мы хотим, чтобы сфера отскакивала от одной точки сцены к другой по плавной кривой (см. рис. 1).

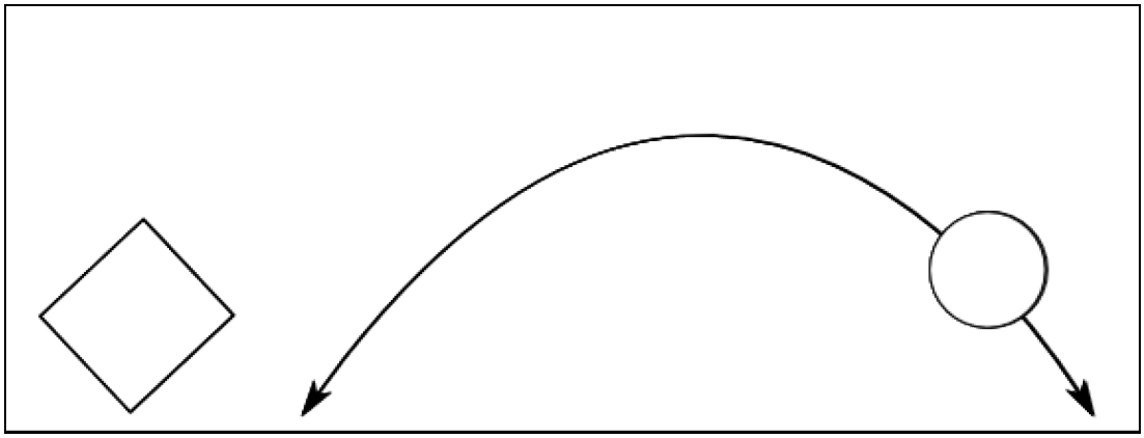

Рис. 1. Траектория отскакивающего мяча

Для этого нам нужно изменить её положение по оси *x* и положение по оси *y*. Функции `Math.cos` и `Math.sin` помогут нам создать плавную траекторию с использованием переменной `step`. Приращение `step += 0.04` определяет скорость прыгающего мяча.

Использование пользовательского интерфейса

Будем использовать dat.GUI, чтобы добавить в наш пример простой пользовательский интерфейс, который позволит нам изменять следующее:

- скорость прыгающего мяча;
- скорость вращения куба.

В JavaScript-объекте `controls` мы определяем два свойства – `rotationSpeed` и `bouncingSpeed` и их значения по умолчанию. Затем мы передаем этот объект в новый объект `dat.GUI` и определяем диапазоны изменения этих двух свойств от `0` до `0.5`.

Далее нам нужно убедиться, что в нашем цикле `renderScene`, мы напрямую ссылаемся на эти два свойства, чтобы при внесении изменений через пользовательский интерфейс dat.GUI это немедленно влияло на скорость вращения и движения наших объектов.

Вращение объекта вокруг точки в пространстве

Когда мы поворачиваем объект, используя его свойство `rotation`, объект поворачивается вокруг своего собственного центра. Однако в некоторых случаях нам может потребоваться повернуть объект вокруг другого объекта. Например, при моделировании солнечной системы требуется вращать Луну вокруг Земли.

Пример: вращение параллелепипеда вокруг сферы

Откройте в браузере файл rotate-around-point-in-space.html.

С помощью элементов управления справа вы можете настраивать вращения красного параллелепипеда. Например, изменяя свойства RotationSpeedX, RotationSpeedY и RotationSpeedZ, вы можете вращать параллелепипед вокруг центра сферы. При данных настройках, нагляднее всего будет изменять свойство RotationSpeedY.

Как это сделать...

Для вращения объекта вокруг другого объекта требуется выполнить несколько дополнительных шагов по сравнению с вращением, которое мы показали ранее:

1. Давайте сначала создадим синюю сферу в центре. Далее вокруг нее мы будем вращать красный параллелепипед.

2. Следующим шагом является определение отдельного объекта, который мы будем использовать в качестве центра поворота для нашего параллелепипеда:

```
// add an object as pivot point to the sphere
pivotPoint = new THREE.Group();
sphereMesh.add(pivotPoint);
```

Объект `pivotPoint` – это объект класса `THREE.Group`. Этот объект может быть создан без определения геометрии и материала. Мы не добавляем его в сцену, а добавляем его к сфере, которую мы создали на шаге 1. Если сфера изменит свое положение или ориентацию, это повлияет и на объект `pivotPoint`.

3. Создаем красный параллелепипед, и вместо того, чтобы добавлять его в сцену, мы добавляем его к только что созданному объекту `pivotPoint`:

Теперь мы можем вращать объект `pivotPoint`, и куб будет следовать за вращением `pivotPoint`. Мы делаем это, обновляя свойство `pivotPoint.rotation` в функции рендеринга.

Как это работает...

Когда создается объект `THREE.Mesh`, то обычно он просто добавляется в сцену `THREE.Scene` и существует независимо от остальных объектов. Однако здесь мы использовали возможность добавления в `THREE.Mesh` дочерних элементов с помощью объекта `THREE.Group`. Поэтому, когда родительский объект поворачивается или перемещается, это также влияет на дочерние элементы.

Интересным моментом использованного подхода является то, что теперь мы можем сделать несколько интересных вещей:

- Мы можем повернуть сам блок, изменив свойство `cube.rotation`, как мы это делали ранее при вращении объекта вокруг его собственной оси.
- Мы также можем повернуть параллелепипед вокруг сферы, изменив свойство `rotation` у сферы, поскольку мы добавили `pivotPoint` как дочерний элемент сетки сферы.
- Мы можем даже комбинировать все вращения, изменяя свойства `rotation` у `pivotPoint`, сферы и куба и создавать очень интересные эффекты.

8.1 Задание для самостоятельной работы

Нарисуйте фигуры из вашего домашнего задания

8.2 Задание для самостоятельной работы

Добавьте операции геометрических преобразований над созданными фигурами, которые используются в ваших домашних заданиях

8.3 Задание для самостоятельной работы

С помощью библиотеки dat.GUI добавьте пользовательский интерфейс для интерактивного изменения параметров геометрических фигур и преобразований.

8.4 Задание для самостоятельной работы

Создайте анимацию движения объектов из домашнего задания