

# Лабораторная работа №6

## «Введение в Three.js»

### Оглавление

Three.js .....	1
Использование dat.GUI для создания пользовательского интерфейса .....	1
Инициализация сцены .....	2
THREE.Line.....	3
THREE.LineBasicMaterial .....	3
THREE.LineDashedMaterial.....	3
Задание цвета с помощью объекта THREE.Color.....	4
THREE.PlaneGeometry .....	4
THREE.CircleGeometry.....	5
THREE.RingGeometry.....	5
THREE.ShapeGeometry .....	5
6.1 Задание для самостоятельной работы .....	7
6.2 Задание для самостоятельной работы .....	9

### Three.js

Three.js (<http://threejs.org>) – это библиотека для создания и анимации 3D-сцен. В следующем списке показаны некоторые возможности, которые доступны в Three.js:

- Создание простых и сложных 3D-геометрий.
- Анимация и перемещение объектов в 3D-сцене.
- Применение текстур и материалов.
- Использование различных источников света для освещения сцены.
- Загрузка объектов из программ 3D-моделирования.
- Добавление расширенных эффектов постобработки.
- Работа с собственными шейдерами.
- Создание облаков точек.

С помощью Three.js можно создать сцены любой сложности, от простых 3D-моделей до фотореалистичных сцен в реальном времени (<https://threejs.org/examples/>):

### Использование dat.GUI для создания пользовательского интерфейса

Библиотека dat.GUI (сайт <https://github.com/dataarts/dat.gui>) позволяет очень легко создавать простые компоненты пользовательского интерфейса, с помощью которых можно изменять переменные в коде. Будем использовать dat.GUI для изменения различных атрибутов, а также для переключения видимости фигур.

В основной части нашего JavaScript-кода нужно настроить объекты, которые будут содержать свойства, которые мы будем изменять, используя `dat.GUI`. Например:

```
const controls = {  
    opacityRed: 1.0  
};
```

В этом объекте JavaScript мы определили одно свойство и его значение по умолчанию. Затем мы передаем этот объект в новый `dat.GUI` и определяем диапазон изменения этого свойства от 0 до 1:

```
const gui = new dat.GUI();  
gui.add(controls, 'opacityRed', 0.0, 1.0);
```

Метод `listen()` заставляет интерфейс отслеживать изменение показываемой переменной (если она изменяется в программе, а не пользователем).

Для того, чтобы изменения значений в `dat.GUI` сразу влияли на показываемую сцену, требуется запускать функцию перерисовки `render()` в цикле с помощью функции `requestAnimationFrame(render)`.

## Инициализация сцены

Создадим в `Three.js` нашу первую сцену и познакомимся с функциями создания двумерных геометрий. Поскольку основное предназначение библиотеки `Three.js` – это работа с 3D-графикой, то для просмотра любой графики потребуется создать камеру. Добавим также на сцену оси. С помощью них можно увидеть, где отображаются объекты. Ось *x* окрашена в красный цвет, ось *y* – в зеленый цвет, а ось *z* – в синий цвет.

Для этого, мы должны создать объекты `scene`, `camera`, и `renderer`. Объект `scene` – это контейнер, который используется для хранения всех объектов, которые мы хотим визуализировать. Без объекта `THREE.Scene`, `Three.js` не сможет ничего нарисовать. Объект `camera` определяет, что мы увидим при рендеринге сцены. Объект `renderer` отвечает за вычисление того, как объект сцены будет отрисовываться в браузере. Мы создадим рендер, использующий библиотеку `WebGL`.

С помощью функции `setClearColor` мы устанавливаем цвет фона и сообщаем модулю визуализации, какой размер сцены необходимо визуализировать с помощью функции `setSize`.

Затем, мы создаем объект `axes` и используем функцию `scene.add`, чтобы добавить оси в нашу сцену.

Для просмотра двумерной графики достаточно будет создать камеру, реализующую ортогональную проекцию. Для этого используется метод `THREE.OrthographicCamera`, который имеет параметры `left`, `right`, `top`, `bottom`, `near`, `far`, задающие соответствующие границы области просмотра.

Для позиционирования камеры, используются атрибуты *x*, *y* и *z*. Чтобы направить камеру в центр сцены, используется функция `lookAt`. По умолчанию считается, что центр сцены расположен в точке (0,0,0).

Для рисования объекта требуется задать его геометрию. Далее нужно определить, как она будет выглядеть (например, какой у нее будет цвет). В `Three.js` это делается с помощью создания материалов. Затем геометрия и материал используются для создания сетки. Далее нам нужно добавить сетку в сцену, как мы уже это сделали с осями.

Наконец, мы говорим рендереру отрисовать сцену с использованием предоставленного объекта камеры.

Создавать геометрии можно либо «вручную», используя объект `THREE.BufferGeometry`, либо использовать любую готовую геометрию, уже существующую в `Three.js`.

## THREE.Line

`THREE.Line` создает сетку для рисования одной ломаной линии на основе вершин, записанных в `THREE.BufferGeometry`.

В программе мы сначала создаем новый экземпляр `THREE.BufferGeometry`, затем задаем координаты *x*, *y* и *z* вершин и записываем их в массив `Float32Array`, затем назначаем вершины атрибуту позиции:

```
geometry.setAttribute('position', new THREE.BufferAttribute(
  arrayOfVertices, 3 ));
```

Для каждой вершины мы также вычисляем значение цвета, которое используем для установки атрибута цвета вершины. Эти значения мы записываем в массив цветов `THREE.BufferGeometry`. Чтобы использовать эти значения для установки цвета, нужно присвоить для свойства `vertexColors` материала линии значение `true`.

`Three.js` предоставляет два разных материала для оформления линий: `THREE.LineBasicMaterial` и `THREE.LineDashedMaterial`.

## THREE.LineBasicMaterial

Это материал, который можно использовать в геометрии `THREE.Line` для создания цветных линий. Наиболее важным свойством этого материала является `color`: он определяет цвет линии. Если вы укажете `vertexColors`, то свойство `color` материала будет проигнорировано. Свойство `visible` определяет, будет ли материал видимым.

Большинство свойств материала можно изменить во время выполнения. Однако для изменения некоторых из них (например, `vertexColors`), необходимо еще установить для свойства `NeedUpdate` значение `true`.

В программу добавлен графический интерфейс управления, который можно использовать для изменения рассматриваемых свойств материала (в данном случае цвета, видимости и свойства `vertexColors`).

## THREE.LineDashedMaterial

Следующий материал лишь немного отличается от `THREE.LineBasicMaterial`. С помощью `THREE.LineDashedMaterial` мы можем не только раскрашивать линии, но и создавать эффект штриховой линии, указывая размеры штрихов и интервалов. Этот материал имеет те же свойства, что и `THREE.LineBasicMaterial`, а также три дополнительных, которые вы можете использовать для определения ширины штриха и ширины промежутков между штрихами:

- `DashSize`: размер штриха.
- `GapSize`: это размер промежутка между штрихами.
- `Scale`: масштабирует `DashSize` и `GapSize`. Если масштаб меньше 1, `DashSize` и `GapSize` увеличиваются, тогда как если масштаб больше 1, `DashSize` и `GapSize` уменьшаются.

Для корректной работы шриховой линии, нужно вызвать функцию `ComputeLineDistances()`, которая используется для определения расстояния между вершинами, составляющими линию. Если этого не сделать, промежутки будут отображаться неправильно.

## Задание цвета с помощью объекта `THREE.Color`

Прежде чем мы перейдем к рассмотрению следующего типа геометрии, рассмотрим различные способы задания цвета с помощью объекта `THREE.Color`. В `Three.js` цвет может задаваться как:

- шестнадцатеричная строка (`"#0c0c0c"`);
- шестнадцатеричное значение (`0x0c0c0c`). Этот способ является предпочтительным;
- указав отдельные значения (0,3, 0,5, 0,6) компонент RGB в диапазоне от 0 до 1.

Объект `THREE.Color` имеет следующие функции:

Свойство	Описание
<code>set(value)</code>	Устанавливает значение цвета в соответствии с заданным шестнадцатеричным значением. Это значение может быть строкой, числом или существующим экземпляром <code>THREE.Color</code> .
<code>setHex(value)</code>	Устанавливает значение цвета в соответствии с заданным числовым шестнадцатеричным значением.
<code>setRGB(r, g, b)</code>	Устанавливает значение цвета на основе предоставленных значений компонент RGB. Диапазон значений от 0 до 1.
<code>setStyle(style)</code>	Устанавливает значение цвета в формате задания цветов CSS. Например, вы можете использовать такие конструкции как <code>"rgb(255, 0, 0)"</code> , <code>"#ff0000"</code> , <code>"#f00"</code> или даже <code>"red"</code> .
<code>fromArray(array)</code>	Работает также как и метод <code>setRGB</code> , но теперь значения RGB могут быть представлены в виде массива чисел.
<code>getHex()</code>	Возвращает шестнадцатеричное значение цвета.
<code>getStyle()</code>	Возвращает значение цвета в формате CSS.

## `THREE.PlaneGeometry`

Объект `THREE.PlaneGeometry` можно использовать для создания прямоугольника.

Создать объект `THREE.PlaneGeometry` очень просто и делается следующим образом:

```
new THREE.PlaneGeometry(width, height, widthSegments, heightSegments)
```

Объяснение этих свойств показано в следующем списке:

- `width`: это ширина прямоугольника.
- `height`: это высота прямоугольника.
- `widthSegments`: это количество сегментов, на которые следует разделить ширину. По умолчанию это 1.
- `heightSegments`: это количество сегментов, на которые следует разделить высоту. По умолчанию это 1.

В качестве материала будем использовать простой материал (`THREE.MeshBasicMaterial`), который характеризуется цветом.

В графическом интерфейсе можно управлять свойствами этой геометрии и материала. Также можно переключать свойство каркаса материала.

## THREE.CircleGeometry

С помощью этой геометрии можно создать круг (или часть круга).

Рассмотрим свойства, определяющие внешний вид круга:

- `radius`: радиус круга. Значение по умолчанию — 50.
- `segments`: определяет количество сегментов, используемых для создания круга. Значение по умолчанию — 8, минимальное число — 3. Большее количество влияет на гладкость круга, но увеличивает количество полигонов.
- `thetaStart`: угол, с которого начинается рисование круга. Это значение может находиться в диапазоне от 0 до  $2 * \text{PI}$ , значение по умолчанию — 0.
- `thetaLength`: угловой размер круга. Значение по умолчанию —  $2 * \text{PI}$ .

Следующий пример создает полный круг радиусом 3, разделенный на 12 сегментов:

```
new THREE.CircleGeometry(3, 12)
```

Если вы хотите создать половину круга, можно использовать следующий вызов:

```
new THREE.CircleGeometry(3, 12, 0, Math.PI);
```

Круг начинается со значения по умолчанию 0 и рисуется только наполовину, поскольку мы указываем `thetaLength` равным `Math.PI`.

## THREE.RingGeometry

`THREE.RingGeometry` отличается от `THREE.CircleGeometry`, отверстием в центре круга.

Свойства `THREE.RingGeometry`:

- `innerRadius`: радиус отверстия. Если для этого свойства установлено значение 0, отверстие отображаться не будет. Значение по умолчанию — 0.
- `outerRadius`: внешний радиус круга. Значение по умолчанию — 50.
- `thetaSegments`: количество диагональных сегментов, которые будут использоваться для создания круга. Значение по умолчанию — 8.
- `phiSegments`: это количество сегментов, которые необходимо использовать по длине кольца. Значение по умолчанию — 8.
- `thetaStart` и `thetaLength` — аналогичны `THREE.CircleGeometry`.

## THREE.ShapeGeometry

`THREE.ShapeGeometry` позволяет создавать произвольные 2D-фигуры. Эта функциональность напоминает функциональность рисования сложных фигур библиотеки `<canvas>`.

Конструктор `THREE.ShapeGeometry` принимает в качестве аргумента объект `THREE.Shape` или массив объектов `THREE.Shape`. Помимо `THREE.Shape`, также можно определить, насколько плавными будут граничные кривые. Значение по умолчанию — 12.

Объект `THREE.Shape` создает контур фигуры, используя линии, кривые и сплайны. С помощью свойства `holes` можно создавать отверстия в фигуре.

Теперь давайте посмотрим на список функций рисования, которые можно использовать для создания `THREE.Shape`:

- `moveTo(x, y)`: перемещение позиции рисования в точку с координатами  $x$  и  $y$ .
- `lineTo(x, y)`: рисует прямолинейный отрезок от текущей позиции (например, заданной функцией `moveTo`) до заданных координат  $x$  и  $y$ .
- `quadraticCurveTo(aCPx, aCPy, x, y)`: задает квадратичную кривую с использованием одной дополнительной точки (с координатами  $aCPx$  и  $aCPy$ ), к которой кривая будет притягиваться и заканчиваться в указанной конечной точке (с координатами  $x$  и  $y$ ). Начальной точкой является текущая позиция пути.
- `bezierCurveTo(aCPx1, aCPy1, aCPx2, aCPy2, x, y)`: рисует кубическую кривую по заданным точкам. Кривая рисуется на основе двух промежуточных точек (с координатами  $aCPx1$ ,  $aCPy1$  и  $aCPx2$ ,  $aCPy2$ ), и конечной точки с координатами ( $x$  и  $y$ ). Начальной точкой является текущая позиция пути.

На следующем рисунке поясняются различия между этими двумя кривыми:

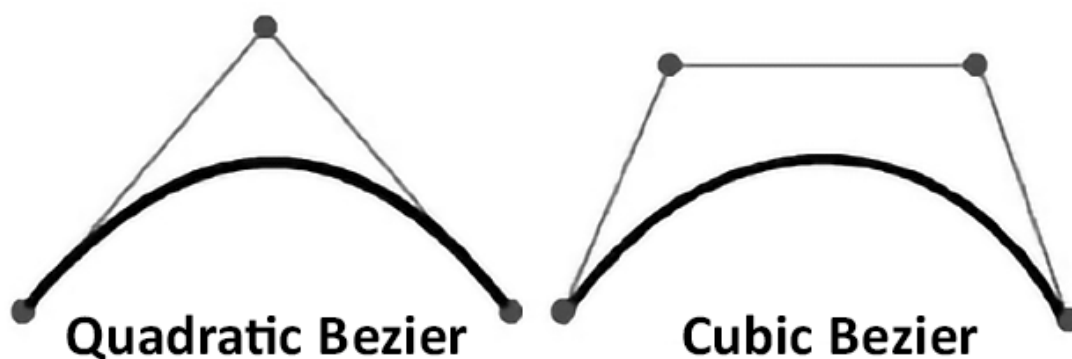


Рис. 1. Квадратичная и кубическая Безье

- `splineThru(pts)`: эта функция рисует плавную линию через предоставленный массив координат. Этот массив должен быть массивом объектов `THREE.Vector2`. Отправной точкой является текущее положение пути.
- `arc(aX, aY, aRadius, aStartAngle, aEndAngle, aClockwise)`: рисует круг (или часть круга). Круг начинается с текущего положения пути. Здесь  $aX$  и  $aY$  используются как смещения от текущей позиции,  $aRadius$  устанавливает радиус круга, а  $aStartAngle$  и  $aEndAngle$  определяют размер рисуемой части круга. Логическое свойство `aClockwise` определяет, рисуется ли круг по часовой стрелке или против часовой стрелки.
- `absArc(aX, aY, aRadius, aStartAngle, aEndAngle, aClockwise)`:

См. описание функции `arc`. Позиция является абсолютной, а не относительной текущей позиции.

- `ellipse(aX, aY, xRadius, yRadius, aStartAngle, aEndAngle, aClockwise)` : см. описание функции `arc`. В качестве дополнения с помощью функции эллипса мы можем отдельно устанавливать длины полуосей `xRadius` и `yRadius`.

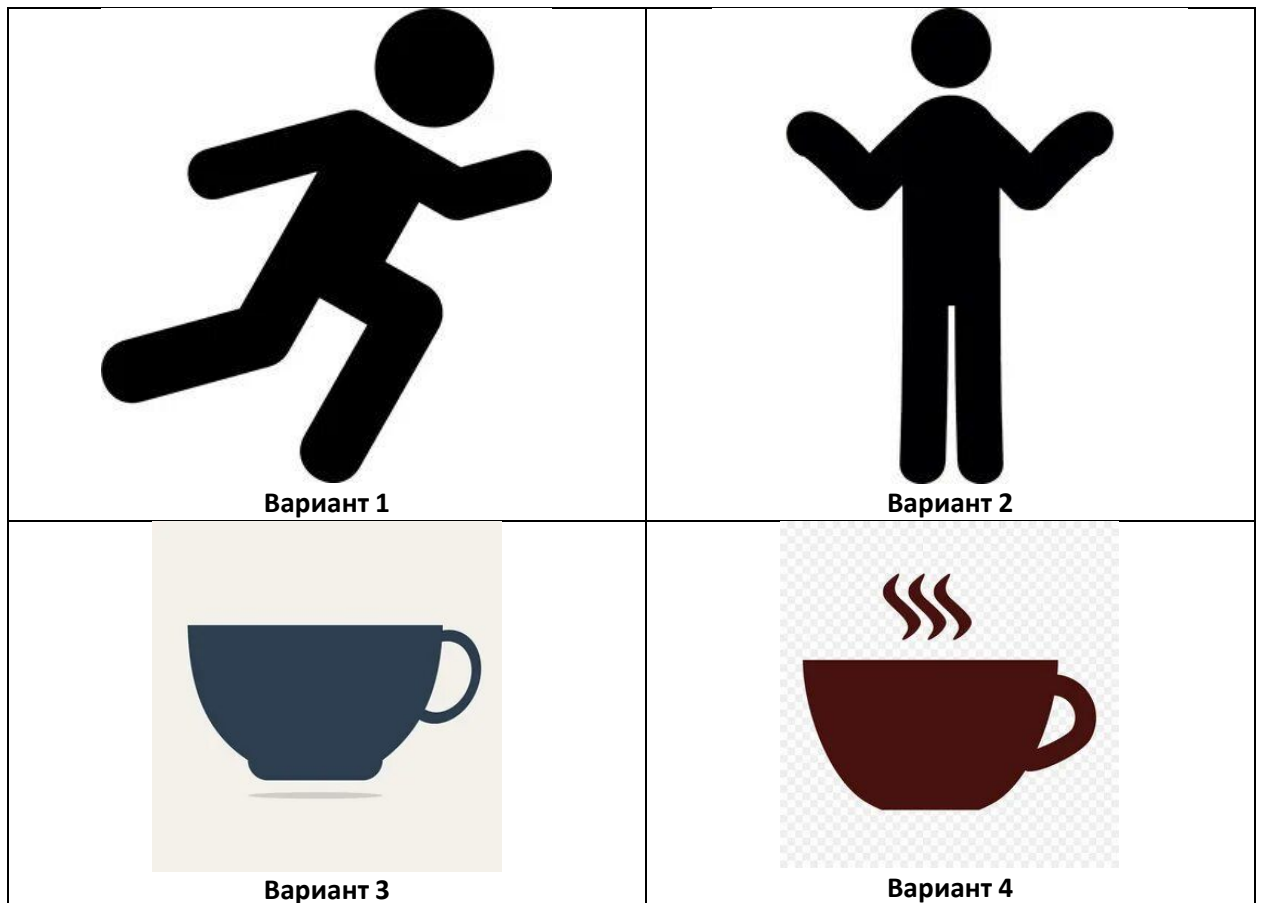
- `absEllipse(aX, aY, xRadius, yRadius, aStartAngle, aEndAngle, aClockwise)`: см. описание функции `ellipse`. Позиция является абсолютной, а не относительной текущей позиции.

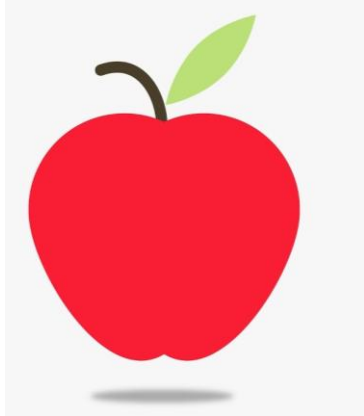
- `fromPoints(vectors)`: если вы передадите в эту функцию массив координат точек `THREE.Vector2` (или `THREE.Vector3`), `Three.js` создаст ломаную линию, проходящую через эти точки.

- `holes`: Свойство `holes` содержит массив объектов `THREE.Shape`. Каждый из объектов в этом массиве отображается как отверстие.

## 6.1 Задание для самостоятельной работы

Создайте рисунок с помощью методов `Three.js`.

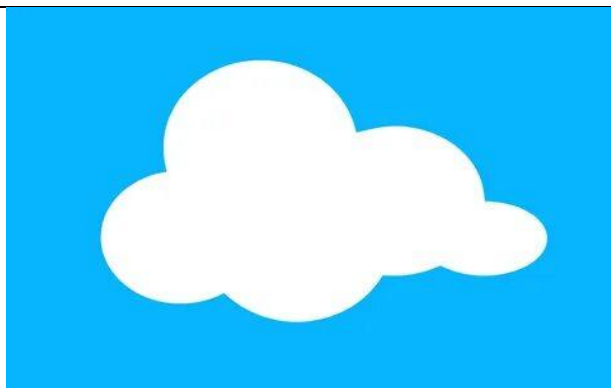




Вариант 5



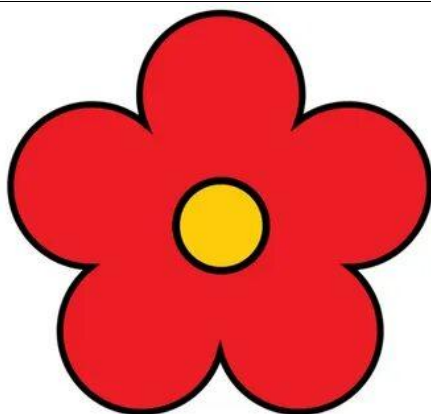
Вариант 6



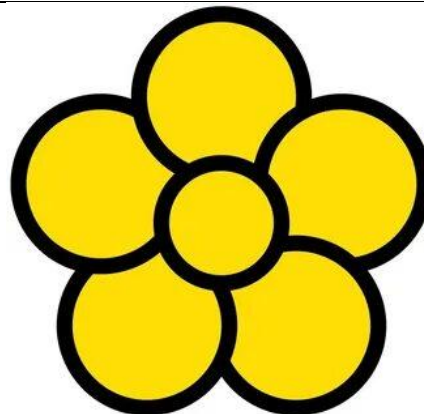
Вариант 7



Вариант 8



Вариант 9



Вариант 10




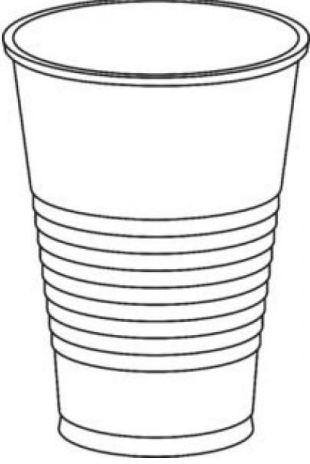
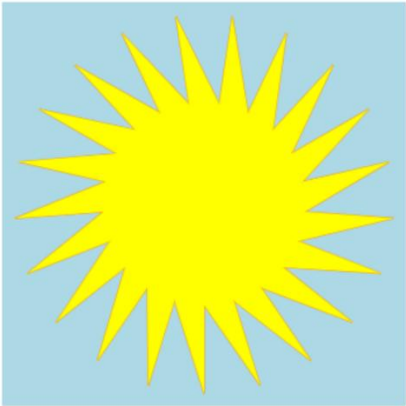



Вариант 11



Вариант 12



 <p>Вариант 13</p>	 <p>Вариант 14</p>
 <p>Вариант 15</p>	 <p>Вариант 16</p>
 <p>Вариант 17</p>	 <p>Вариант 18</p>

## 6.2 Задание для самостоятельной работы

С помощью dat.GUI создайте интерфейс для изменения атрибутов созданного рисунка.