МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
БАШКИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

А.Н. Вильданов

3D-моделирование на WebGL с помощью библиотеки Three.js

Учебное пособие

Уфа
РИЦ БашГУ
2014

УДК 004.925 (07)
ББК 32.97я7
В46

Рецензенты:
проф., д-р физ.-мат. наук **Р.Ф. Маликов** (БГПУ, г. Уфа);
отдел вычислительной математики
(Институт математики с ВЦ УНЦ РАН, г. Уфа).

Вильданов А.Н.

В46 3d-моделирование на WebGL с помощью библиотеки Three.js: учебное пособие / А.Н. Вильданов. − Уфа: РИЦ БашГУ, 2014. – 113 с.

ISBN 987-5-7477-3560-6

Пособие посвящено молодой, но перспективной технологии построения трехмерной графики в браузере WebGL. В пособии развернуто описываются средства библиотеки Three.js для создания трехмерных геометрических объектов, анимации и элементов интерактивности.

Данное пособие рекомендовано в качестве дополнительной литературы по дисциплине «Основы web-программирования» для студентов направления подготовки «Прикладная математика и информатика». Также может служить пособием при написании курсовых и дипломных работ, связанных с трехмерной графикой.

УДК004.925 (07)
ББК 32.97я7

ISBN 987-5-7477-3560-6

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ АВТОРА

Предлагаемое пособие предназначено раскрыть возможности 3D-моделирования с помощью технологии WebGL, опираясь на библиотеку Three.js. Желание написать эту книгу вызвано, во-первых, новизной и перспективностью самой технологии, а во-вторых, малым или, наверно, даже полным отсутствием литературы на русском языке, посвященной этой технологии.

В самом пособии о «чистом» WebGL сказано мало. Целиком и полностью книга посвящена методам библиотеки Three.js. В этом автор не видит ничего предосудительного, так как считает, что программист не должен сильно отвлекаться на проблемы типа «как изобразить сферу», а должен больше времени уделять логике работы программы и реализации ее основной идеи.

Книга снабжена, как обычно, иллюстрациями, а также программным кодом, реализующим основные детали рассматриваемых примеров.

Пособие рассчитано на студентов, имеющих базовые навыки программирования, опыт работы с HTML, CSS и с языком JavaScript. При этом совсем не обязательно знать JavaScript полностью. Главное, чтобы человек представлял, для чего нужен язык JavaScript, а также не ленился самостоятельно выяснять назначение той или иной незнакомой ему стандартной функции.

Свои отзывы, пожелания и замечания можете присылать автору по адресу alvild@gmail.com.

ВВЕДЕНИЕ

Что такое WebGL?

С развитием Web стал неизбежно нуждаться в 3D возможностях.

WebGL (Web-based Graphics Library[1]) – программная библиотека, предназначенная для создания интерактивной трехмерной графики в веб-браузерах.

За счёт использования низкоуровневых средств поддержки OpenGL часть кода на WebGL может выполняться непосредственно на видеокартах, что дает выигрыш по быстродействию.

Под этим же словом «WebGL» обычно подразумевают и саму технологию создания трехмерной графики с помощью одноименной библиотеки. Технология WebGLразрабатывается промышленным консорциумом Khronos Group, который специализируется на выработке открытых стандартов интерфейсов программирования в области создания и воспроизведения динамической графики и звука на широком спектре платформ и устройств. В консорциум входят более 100 компаний.

Построение графики происходит на объекте canvas. Отрисовка в WebGL осуществляется с помощью так называемых вершинных и пиксельных шейдеров.

Шейдеры – это функции, написанные на специальном языке программирования GLSL, которые выполняются графической картой и обрабатывают данные вершин и пикселей, определяя окончательные параметры изображения объекта. Они могут включать в себя описание поглощения и рассеяния света, наложение текстуры, отражение и преломление, затенение, и т.д.

Работа с шейдерами – это достаточно трудоемкий процесс. Ко всему прочему, нужно еще описать каждую вершину, каждое ребро, каждую грань, нормали к поверхности, их цвет, положение и пр. Для повышения скорости разработки можно использовать один из нескольких фреймворков WebGL – Sylvester, glUtils.js, webgl-utils.js, Three.js, и др. В данном пособии выбор пал на активно развивающуюся библиотеку с открытым исходным кодом Three.js.

[1] Дословно можно перевести как: «Библиотека для графики в Web»

Поддержка браузерами

– Google Chrome – WebGL включён по умолчанию во все версии начиная с 9;

– Mozilla Firefox – WebGL был включён во все платформы, у которых есть нужная графическая карта с актуальными драйверами, начиная с версии 4.0.;

– Opera – WebGL реализован в версии Opera 12.00, но отключен по умолчанию;

– Safari – поддерживает WebGL, но поддержка отключена по умолчанию;

– Internet Explorer – компания Microsoft уже официально подтвердила поддержку WebGL в IE11;

– Браузер Яндекс создан компанией «Яндекс» на базе браузера с открытым исходным кодом Chromium, поэтому также поддерживает WebGL.

Что такое Three.js?

Three.js –это библиотека JavaScript, содержащая набор готовых классов для создания и отображения интерактивной компьютерной 3D графики в WebGL.

Библиотека Three.js облегчает работу с WebGL. При использовании Three.js отпадает необходимость в написании шейдерных процедур (но эта возможность остается), и появляется возможность оперировать с более привычными и удобными понятиями сцены, света и камеры, объектами и их материалами.

Библиотека Three.js также поддерживает отображение готовых трёхмерных моделей формата Collada (который обеспечивает совместимость моделей таких программ, как Maya, 3ds Max, Blender, Unreal engine, и т.д.)

Над библиотекой работает большое количество разработчиков. Главным идеологом и разработчиком является программист из Барселоны Рикардо Кабельо (Ricardo Cabello), творческий псевдоним Mr. Doob.

Для использования библиотеки нужно скачать с сайта threejs.org файл three.min.js[2] (собственно, сама библиотека, содержащая минимальный набор функций для работы с WebGL), и подключить к вашему проекту.

[2] URL: http://threejs.org/build/three.min.js

Также на странице сайта threejs.org доступно для скачивания все содержимое сайта с примерами, исходниками и документацией.

Не забывайте отлавливать ошибки в приложении. Если вы видите пустой экран, то стопроцентно возникла ошибка, и ее нужно найти. Например, в браузере Google Chrome для этого следует зайти в Меню – Инструменты – Консоль Javascript и выбрать вкладку Console (обычно по умолчанию).

1. ПЕРВЫЕ ШАГИ С THREE.JS

Создадим папку webgl для нашего первого рабочего проекта. В ней будут располагаться файл главной страницы index.html, папка js для библиотеки three.min.js и файл с логикой нашего приложения mainapp.js (рис.):

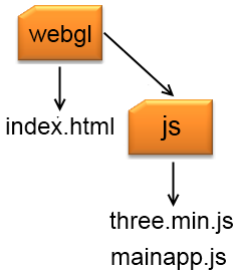

Рис.1. Структура нашего приложения

На главной странице index.html нужно подключить наши библиотеки и создать раздел, в котором будет создан «холст» canvas для отрисовки. Для обращения к этому разделу присвоим ему, например, идентификатор «MyWebGLApp». Тогда главная страница может иметь вид:

```
<!DOCTYPEhtml>
<html>
<head>
    <title>Фигуры WebGL</title>
    <meta charset="utf-8">
    <style>
        body {
            margin: 0;
            padding: 0;
             overflow: hidden;
      }
    </style>
<script type="text/javascript" src="js/three.min.js"></script>
<script type="text/javascript" src="js/mainapp.js"></script>
</head>
<body>
    <div id="MyWebGLApp"></div>
</body>
</html>
```

Здесь мы убрали полосу прокрутки страницы при помощи стилевой команды «overflow: hidden;».

Скачайте файл three.min.js и разместите его в папке js. Затем создайте новый пустой текстовый файл и сохраните под названием mainapp.js. Весь дальнейший код мы будем писать именно в этом файле. Используйте для этого любой подходящий, наиболее удобный для вас редактор.

Работу с 3D графикой WebGL с помощью Three.js можно условно разбить на следующие этапы:

– добавление сцены;
– добавление камеры;
– добавление света (освещения);
– добавление графических объектов на сцену;
– создание объекта визуализации;
– рендеринг (визуализация);
– анимация (движение объектов, их взаимодействие).

1.1. Добавление сцены

Именно на сцену мы будем добавлять все созданные нами объекты. Объявим переменную:

```
varscene;
```

Переменная scene должна быть глобальной. Создается сцена просто:

```
scene = newTHREE.Scene();
```

Для добавления объектов на сцену или удаления их со сцены используются методы add и remove:

```
scene.add( object );
…
scene.remove( object );
```

1.2. Добавление камеры

Камера, по сути – это «глаз», который смотрит на нашу сцену. Простейший тип камеры – это «перспективная» камера, которая воспринимает все объекты в перспективной проекции. Т.е., например, чем дальше объект находится от нас, тем он кажется нам меньше. Объявим глобальную переменную и создадим камеру:

```
varcamera;
…
camera = newTHREE.PerspectiveCamera(45,
          window.innerWidth / window.innerHeight, 1, 10000 );
```

Первый аргумент – это FOV (field of View – поле (угол) зрения), в примере – 45.

Второй аргумент – это пропорция – соотношение сторон (обычно ширина экрана в пикселях делится на высоту, например, 1.33 = 4:3).

Третий и четвертый аргументы – минимальное и максимальное расстояние от камеры, которое попадает в рендеринг. Так, очень далекие точки не будут отрисовываться вообще.

У камеры можно указать положение:

```
camera.position.set(0, -20, 100);
```

и направление обзора:

```
camera.lookAt(newTHREE.Vector3(10, -100, 100));
```

(камера направлена на конец вектора с координатами (10, -100, 100)).

Можно «следить» за объектами:

```
camera.lookAt (object.position);
```

По умолчанию камера всегда смотрит в «центр» холста с координатами (0,0,0). При изменении каких-либо параметров камеры нужно затем вызвать метод:

```
camera.updateProjectionMatrix();
```

С нашей перспективной камерой трудно достаточно полно оценить «трехмерность» объектов на сцене. Хочется иметь возможность рассматривать объекты с разных сторон, вблизи или издалека.

Для этого в Three.js имеется возможность управления обзором сцены с помощью специальных «контролов», добавление которых позволяет менять точку обзора камеры с помощью мышки, приближаться или удаляться от сцены.

Скачайте файл TrackballControls.js с сайта http://threejs.org[3] и закиньте в папку js. Естественно, для ее использования нужно на нашей главной странице index.html добавить ссылку на эту библиотеку:

```
<script type="text/javascript" src="js/TrackballControls.js">
</script>
```

Теперь, после создания камеры, можно добавить наш контрол:

```
var controls;
…
controls = new THREE.TrackballControls( camera, container );
```

Первый параметр – наша камера camera. Второй параметр container указывает на раздел страницы, в котором производится отрисовка (см. ниже).

После создания контрола можно задать, например, скорость вращения камеры при движении мыши:

```
controls.rotateSpeed = 2;
```

будет ли изменяться положение камеры (приближаться или удаляться от сцены) при кручении колесика мыши (по умолчанию false – меняется), и скорость такого изменения:

```
controls.noZoom = false;
controls.zoomSpeed = 1.2;
```

Также можно указать, будет ли камера после остановки мыши немного двигаться по инерции или же нет; по умолчанию false (движется):

```
controls.staticMoving = true;
```

[3] http://threejs.org/examples/js/controls/TrackballControls.js

и т.д. Теперь можно будет с помощью мышки рассматривать объекты сцены со всех сторон, приближаться и удаляться от них, что, безусловно, придаст нашему приложению большей привлекательности.

Для удобства добавление камеры и контрола выделено в отдельную процедуру AddCamera(), полный код которой можно посмотреть в приложении А.

Замечание. Также имеются и другие контролы. Например, FirstPersonControls позволяет как бы «видеть» сцену от первого лица, и передвигаться по ней с помощью стрелок (или кнопочек W, A, S, D) и мышки. Пример можно посмотреть на сайте http://threejs.org[4].

1.3. Система координат в WebGL

Для расположения фигур в пространстве используется Декартова система координат:

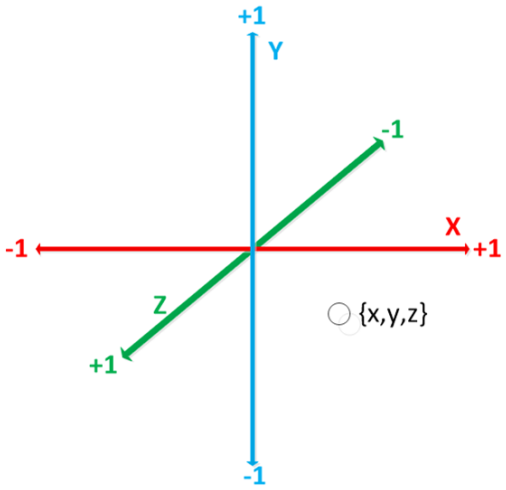

Рис.2. Декартова система координат в WebGL

Как обычно, для программ, работающих с 3D, наверх направлена ось Y. На рисунке также указаны правила изменения знаков. Например, при приближении к нам значение координаты Z увеличивается.

[4]URL: http://threejs.org/examples/webgl_geometry_minecraft.html

При добавлении объекта на сцену в Three.js можно указать его координаты:

```
object.position.x = -50;
object.position.y = 20;
object.position.z = 60;
```

или одной строкой:

```
object.position.set( -50, 20, 60 );
```

или с помощью класса трехмерных векторов Vector3:

```
object.position = new THREE.Vector3( -50, 20, 60 );
```

При этом в указанной точке обычно располагается геометрический центр тела. Если же координаты объекта не указаны, то они все равны нулю.

Можно задать коодинаты объекта, приравняв их к координатам другого:

```
object2.position = object1.position;
```

Для задания углов поворота тела используется свойство rotation. Углы указываются в радианах. Например, команда

```
object.rotation.y = Math.PI/2;
```

означает, что объект разворачивается на 90 градусов против часовой стрелки, если смотреть «сверху» – со стороны положительного направления оси OY, при этом осью вращения служит ось ординат OY. Синтаксис обращения к rotation такой же, как и у position.

1.4. Добавление света в Three.js

Освещение объектов придаст вашей сцене большей реалистичности. Для освещения можно использовать класс DirectionalLight, который представляет собой источник прямого направленного освещения. Этот источник можно сравнить с солнечными лучами. С его помощью создается поток параллельных лучей во всех направлениях.

Объявим глобальную переменную для источника света:

```
varlight;
```

Да будет свет:

```
light = new THREE.DirectionalLight( 0xffffff );
light.position.set( 0, 100, 100 );
scene.add( light );// добавление света на сцену
```

В качестве параметра при создании света можно указать цвет освещения (обычно белый). В приложении А добавление света оформлено в виде отдельной процедуры AddLight().

1.5. Добавление объекта визуализации

Для отображения сцены и ее объектов при помощи WebGL, в Three.js используется специальный класс WebGLRenderer. Объявим переменные для экземпляраэтого класса и для DOM-элемента, с которым этот класс будет работать:

```
varrenderer, container;
```

Вместе с классом создается холст canvas, который по умолчанию имеет ширину 300 пикселей и высоту 150 пикселей. Метод setSize позволяет изменить его размеры. Например, создадим объект-визуализатор, указав размеры холста «на все окно»:

```
renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth,window.innerHeight );
```

Также можно указать цвет фона (в разных версиях Three.js цвет фона по умолчанию менялся). Укажем белый фон:

```
renderer.setClearColor( 0xffffff );
```

Далее следует указать визуализатору renderer, где именно будет создаваться холст. Мы приготовили для этого раздел MyWebGLApp на главной странице:

```
container = document.getElementById('MyWebGLApp');
container.appendChild( renderer.domElement );
```

Теперь холст для рисования размещен в нашем разделе на главной странице.

Для придания моделям более реалистичного вида в 3D графике применяют **сглаживание**. В Three.js допускается использовать сглаживание при помощи параметра antialias. Тогда объявление можно изменить следующим образом:

```
renderer = newTHREE.WebGLRenderer( { antialias: true } );
```

Все примеры, демонстрируемые в данном пособии, сделаны при наличии этого параметра.

Также в Three.js имеется, например, класс CanvasRenderer, который обеспечивает визуализацию простой графики в браузерах без поддержки WebGL. Тогда можно составить более сложную конструкцию:

```
try
    {
renderer=newTHREE.WebGLRenderer( { antialias: true } );
}
catch(err)
    {
       alert('В вашем браузере отсутствует поддержка
                                                    WebGL!');
try
        {
            Renderer=new THREE.CanvasRenderer;
  }
catch(err)
         {
            alert('Пожалуйста, установите новый браузер
                              с поддержкой WebGL!');
}
   }
```

Конечно, при отображении в браузерах без поддержки WebGL пострадают качество и скорость отображения. И, повторюсь, изображаться корректно будут лишь объекты с простыми материалами.

1.6. Рендеринг и анимация

Рендеринг (отрисовка) производится с помощью созданного в предыдущем пункте объекта renderer. Для этого у него есть метод render, в параметрах которого указываются сцена и камера:

```
renderer.render(scene, camera);
```

Анимация есть, по сути, последовательный неоднократный рендеринг сцены (для отображения динамики). Такое последовательное отображение осуществляется специально для этого заточенной функцией requestAnimationFrame. Ее можно назвать аналогом другой Javascript-функции setInterval, но она уже оптимизирована, например, для отображения неподвижных объектов, и т.д.

Чтобы не усложнять новичку код новыми названиями, оставим стандартные для демонстрационных примеров Three.js функции – init(), animate() и render(). В первой происходит инициализация сцены, камеры, света, добавление всех объектов, которые мы хотим увидеть на сцене. Функция animate() будет использовать requestAnimationFrame и постоянно вызывать функцию render(), которая уже и будет отрисовывать все изменения:

```
function animate()
{
    requestAnimationFrame(animate);
    render();
}
```

Функция render() будет содержать все изменения на сцене. Например, задать движение объекта Cube можно такой функцей:

```
varCube;// переменная Cube должна быть глобальной

function render()
{
    Cube.position.x = Cube.position.x + 1;
    Cube.rotation.y = Cube.rotation.y + 0.01;
    controls.update();
    renderer.render(scene, camera);
}
```

Теперь объект Cube движется вправо и вращается вокруг своей оси, параллельной оси ординат OY.

Здесь мы также учли, что у нас добавлена возможность управления обзором мышкой при помощи controls. При движении мышки положение камеры меняется, и нужно каждый раз рисовать новую картину сцены. Метод update как раз служит для обновления картины при манипуляциях с мышкой.

1.7. Добавление простейших объектов

Перед добавлением объектов нужно создать сцену, добавить свет, рендерер, камеру, и т.д. Все это, как мы договаривались ранее, оформим в виде отдельной процедуры init() (приложение А). После этого можно добавлять объекты.

Добавление объектов производится следующим образом. Сначала объявляется вид *геометрии* объекта. В простейших случаях это может быть параллелепипед, сфера, цилиндр и т.д. Параметры геометрии включают обычно линейные размеры и количество сегментов, отвечающее за точность изображения.

Например, для прямоугольного параллелепипеда (кубоида) предусмотрен специальный класс BoxGeometry:

```
var geometry = new THREE.BoxGeometry[5]( 200, 300, 50);
```

Указываются ширина кубоида (вдоль оси X), высота (вдоль оси Y) и длина (вдоль оси Z).

Следующим шагом указывается *материал* будущего объекта (о материалах подробнее чуть позже):

```
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 });
```

Здесь мы указали зеленый цвет материала будущего объекта. И, наконец, создается сам объект с выбранными геометрией и материалом. Делается это через класс Mesh – *сетку* будущего объекта:

```
varCube = newTHREE.Mesh( geometry, material );
```

Осталось добавить объект на сцену. Можно указать позицию (свойство position) и поворот относительно осей координат – rotation:

```
Cube.position.z = -100;
```

[5] в старых версиях Three.js класс назывался CubeGeometry

```
Cube.rotation.z = Math.PI / 6;
scene.add( Cube );
```

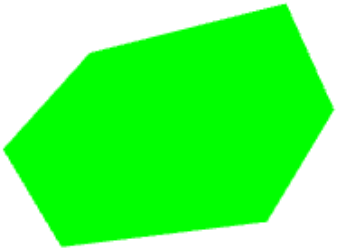

Рис.3. Кубоид

Если теперь добавить код из пункта 1.3, то наш куб будет двигаться вправо и вращаться вокруг оси. При этом переменная Cube должна быть глобальной (см. приложение А).

При нашем выборе материала не так заметна «трехмерность» объекта. Выберем другой вид материала:

```
varmaterial = newTHREE.MeshNormalMaterial();
```

Тогда за счет разноцветной окраски лучше видна трехмерность объекта (рис.):

Рис. 4. Кубоид

Благодаря controls наш кубоид можно рассматривать в браузере со всех сторон, приближаться к нему и удаляться от него. Достаточно лишь движения мышки и ее колеса (рис.):

Рис. 5.Тот же кубоид (с другой стороны)

Замечание. Для геометрии поверхности и материала необязательно объявлять отдельные переменные. Первые три строки создания куба эквивалентны, например, паре таких команд:

```
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 });
var Cube = new THREE.Mesh(
        new THREE.BoxGeometry( 200, 300, 50), material );
```

или даже одной:

```
var Cube = new THREE.Mesh(
        new THREE.BoxGeometry( 200, 300, 50),
        new THREE.MeshBasicMaterial( { color: 0x00ff00 } ) );
```

Как именно делать – дело вкуса и привычки.

1.7.1. Создание материала объекта

При создании материала будущей заготовки можно указать цвет или текстуру, прозрачность. Цвет имеет формат 0xHEX, где HEX – шестнадцатеричное обозначение сочетания красного, синего и зеленого цвета). Например, строка

```
color: 0xDC143C;
```

означает объявление малинового цвета. Прозрачность opacity меняется от 0 до 1 (0 – прозрачный, 1 – абсолютно непрозрачный) и применяется вместе со свойством transparent. Например:

```
var material = new THREE.MeshBasicMaterial( {
                color: 0x33CCFF, transparent: true, opacity: 0.6 } );
```

Материалы бывают разных видов. Например, это:

– MeshBasicMaterial – для закрашивания поверхностей фигур однородным цветом;

– MeshLambertMaterial – для градиентной заливки. Места, на которые падает свет, изображаются более светлыми;

– MeshNormalMaterial – позволяет подчеркнуть «трехмерность» объекта, раскрашивая его грани в разные цвета;

– MeshPhongMaterial – для блестящих поверхностей. Требователен к ресурсам;

– LineBasicMaterial – материал для рисования каркасов;

– LineDashedMaterial – материал для рисования пунктирных каркасов. Можно указать параметры dashSize – длину пунктира, и gapSize – длину разрыва (расстояние между пунктирами).

Некоторые материалы можно совмещать, то есть применять одновременно. Другие особенные виды материалов рассмотрим позже, по мере изложения материала.

Далее, параметр side регулирует видимость материала сторон двусторонних моделей:

–THREE.FrontSide – виден снаружи (по направлению нормалей) – по умолчанию;

–THREE.BackSide – виден изнутри;

–THREE.DoubleSide – виден с обеих сторон.

1.7.2. Добавление сферы

Создадим, например, шар синего цвета. Последовательно создаем геометрию, материал и «мэш»:

```
var geometry = new THREE.SphereGeometry(100, 50, 50);
var material = new THREE.MeshLambertMaterial(
                                        { color: 0x33CCFF } );
var Sphere1 = new THREE.Mesh( geometry, material );
```

Первый параметр SphereGeometry – радиус сферы; второй и третий параметры – количество сегментов по ширине и высоте сферы.

Чем их больше, тем точнее изображается сфера. В качестве материала мы выбрали градиентную заливку.

Указываем позицию:

```
Sphere1.position.x = 0;
Sphere1.position.y = 20;
```

Добавляем на сцену:

```
scene.add (Sphere1 );
```

Сфера готова (рис.):

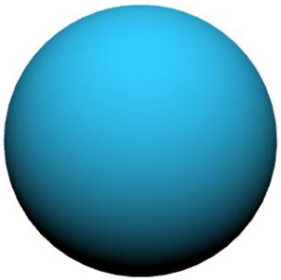

Рис. 6. Сфера (градиентная заливка)

Класс Mesh имеет метод scale, позволяющий растягивать геометрию вдоль указанной оси. Укажем

```
Sphere1.scale.x = 1.5;
```

и получим эллипсоид (рис.):

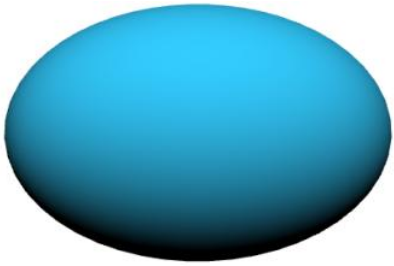

Рис. 7. Вытянутая сфера

Покажем, как количество сегментов влияет на изображение. Если в первом примере взять, например,

```
var geometry = new THREE.SphereGeometry( 100, 12, 8 );
```

(мы изменили количество сегментов на значения 12 и 8), то результат будет выглядеть совсем по-другому (рис.):

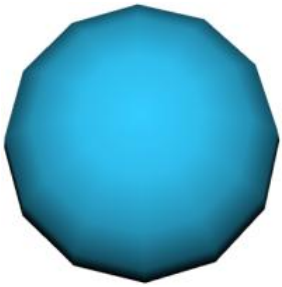

Рис. 8. Сфера (с небольшим количеством сегментов)

Видим, что чем больше сегментов, тем более правдоподобным будет изображение. Но и больше уйдет ресурсов компьютера на прорисовку такой сферы. На практике обычно выбирается разумный компромисс между быстродействием и реалистичностью.

1.7.3. Создание параллелепипеда

Пример создания прямоугольного параллелепипеда с помощью класса BoxGeometry мы рассматривали в начале главы. Более общее объявление класса имеет вид:

```
BoxGeometry( width, height, depth,
        widthSegments, heightSegments, depthSegments );
```

Кроме уже знакомых нам первых трех параметров, отвечающих за ширину, высоту и длину, вторая тройка обозначает количество сегментов, необходимых для детализации изображения соответствующих сторон кубоида. Эти параметры особенно актуальны при наличии текстуры. Если их не указывать, они равны единице.

Вернемся к прошлому примеру, когда мы создавали кубоид. При этом все его грани окрасились одинаковым цветом.

Что же делать, если нужно окрасить каждую грань в свой цвет, как на рис. 9? Нужно объявить массив из 6 материалов:

```
varmaterials = [
newTHREE.MeshBasicMaterial( { color: 0xff0000 }),
                    // правая сторона красная
newTHREE.MeshBasicMaterial( { color: 0x00ff00 }),
                    // левая сторона зеленая
newTHREE.MeshBasicMaterial( { color: 0x0000ff }),
                    //верх синий
new THREE.MeshBasicMaterial( { color: 0xff00ff }),
                    // низ пурпурный
new THREE.MeshBasicMaterial( { color: 0xffff00 }),
                    // лицевая сторона желтая
new THREE.MeshBasicMaterial( { color: 0x00ffff })
                    // задняя сторона цвета циан
];
```

Указываются соответственно правая, потом левая стороны, верх и низ, лицевая и задняя стороны. Для «объединения» массива материалов используется метод MeshFaceMaterial:

```
varmaterial = newTHREE.MeshFaceMaterial( materials );
```

Геометрию задаем аналогично:

```
vargeometry = newTHREE.BoxGeometry(100, 150, 200);
```

Теперь добавляем кубоид, указывая построенный материал:

```
Cube = new THREE.Mesh( geometry, material );
Cube.rotation.y = - Math.PI / 6;
scene.add( Cube );
```

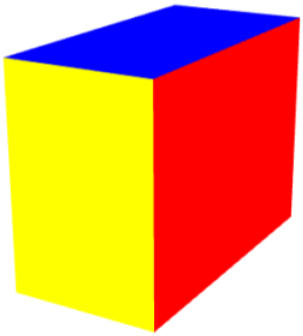

Рис. 9. Разноцветный параллелепипед

1.7.4. Создание пирамиды, призмы, цилиндра и конуса

Все эти поверхности создаются с помощью одной и той же команды CylinderGeometry. Создадим геометрию:

```
var radius_top = 0;
var radius_bottom = 128;
var heigth = 240;
var segments = 3;
var geometry = new THREE.CylinderGeometry(
                    radius_top, radius_bottom, heigth, segments );
```

Указываются радиусы верхнего и нижнего основания, высота фигуры, и количество сегментов, равное количеству сторон многоугольников на основаниях. Создадим материал

```
varmaterial= newTHREE.MeshNormalMaterial({color: 0xf2ddc6});
```

и треугольную пирамиду:

```
var piramida = new THREE.Mesh( geometry, material );
piramida.position.set(-20,0,100);
scene.add( piramida );
```

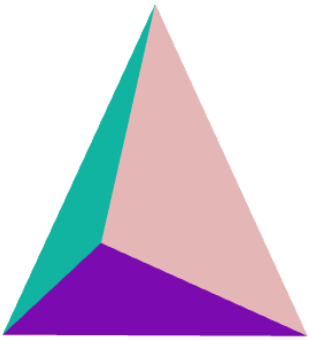

Рис. 10. Треугольная пирамида

Выбрав же верхний и нижний радиус одинаковыми, сразу получаем треугольную призму:

```
var radius_top = 128;
var radius_bottom = 128;
var heigth = 240;
var segments = 3;
var geometry = new THREE.CylinderGeometry(
                radius_top, radius_bottom, heigth, segments );
…//то же самое
```

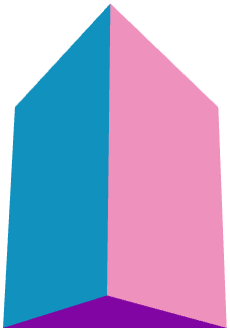

Рис. 11. Треугольная призма

Если теперь изменить количество сторон в основании, получим фигуру, похожую на цилиндр:

```
var radius_top = 128;
```

```
var radius_bottom = 128;
var heigth = 240;
var segments = 16;
var geometry =  …//то же самое
```

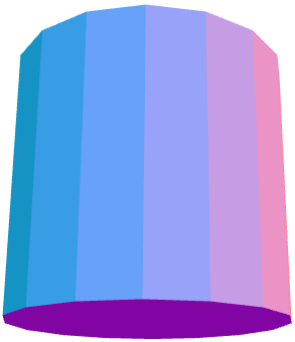

Рис. 12. Цилиндр

Понятно, что чем больше количество сегментов в основании, тем больше будет похожа фигура на цилиндр. И, наконец, уменьшив верхний радиус до нуля, получим конус:

```
var radius_top = 0;
var radius_bottom = 128;
var heigth = 240;
var segments = 16;
var geometry =  …//то же самое
```

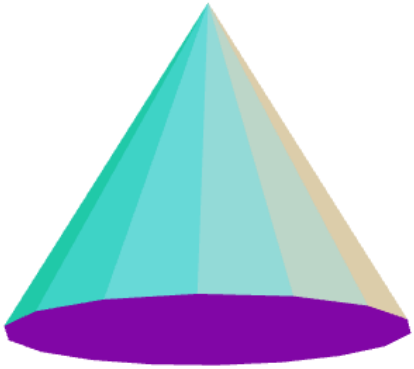

Рис. 13. Конус

Наконец, взяв верхний радиус ненулевым, получим усеченный конус:

```
var radius_top = 64;
var radius_bottom = 128;
var heigth = 240;
var segments = 16;
var geometry =  …
```

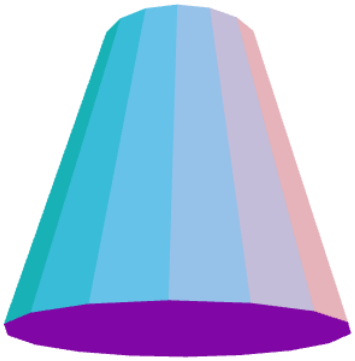

Рис. 14. Усеченный конус

1.8. Добавление текстур

1.8.1. Наложение текстуры на куб и на плоскость

Для наложения текстур в Three.js имеется специальный класс ImageUtils с методом loadTexture, главным параметром которого является адрес (путь) к изображению текстуры. На основе текстуры Texture создается материал с параметром map: Texture. Далее – все как обычно, создается фигура с заданной геометрией и нашим материалом.

Для того чтобы текстуры нормально отображались, необходимо, чтобы **обращение к веб-страницам происходило через сервер**. Для «домашнего» тестирования достаточно установить на компьютере Денвер. Запустим Денвер и скинем наш проект (папку webgl со всеми файлами) на виртуальный диск в папку home\localhost\www\. Теперь наша страница доступна в браузере по адресу http://localhost/webgl/.

Создадим в директории нашего проекта папку textures, закинем туда две заготовки для наших будущих текстур kote.jpg и checkerboard.jpg:

Рис. 15. Текстура kote.jpg

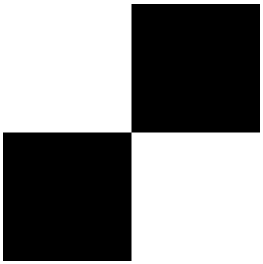

Рис. 16. Текстура checkerboard.jpg

Создадим, например, куб с текстурой kote.jpg:

```
var geometry = new THREE.BoxGeometry( 200, 200, 200);
var Texture =
      new THREE.ImageUtils.loadTexture(  'textures/kote.jpg' );
var material = new THREE.MeshBasicMaterial( { map: Texture
} );

Cube = new THREE.Mesh( geometry, material );
Cube.rotation.y = Math.PI / 4;
scene.add( Cube );
```

Результат (рис. 17):

Рис. 17. Куб с текстурой

Если нужно наложить на разные стороны разные текстуры, то нужно создать массив материалов, аналогично тому, как это мы делали в пункте 0, когда раскрашивали параллелепипед разными цветами.

Подготовим и закинем в папку textures картинки с цифрами 1, 2, 3, 4, 5, 6 с соответствующими названиями (1.png, 2.png, и т.д.). Объявим массив материалов и заполним его в цикле:

```
var materials = [];
for (i=1; i<=6; i++)
  {
    var Texture =  new THREE.ImageUtils.loadTexture(
                                    'textures/' + String(i) + '.png' );

    var Material = new THREE.MeshBasicMaterial(
                                  { map: Texture, color: 0x00dcff} );
    materials.push( Material );
  }
```

Далее поступаем аналогично: собираем материал, объявляем геометрию и создаем куб:

```
var material = new THREE.MeshFaceMaterial( materials ) ;
var geometry = new THREE.BoxGeometry( 100, 100, 100);

var Cube = new THREE.Mesh( geometry, material );
Cube.rotation.y = Math.PI / 6;
scene.add( Cube );
```

Рис. 18. Куб с разными текстурами

Однородную текстуру можно «размножать» по поверхности. При этом используемые изображения должны иметь размеры, равные степеням двойки. Например, 16px, 32px, 64px, 128px и т.д. Высота и ширина изображения не обязательно должны быть равными. Главное, чтобы их значения были равны степени двойки.

Создадим на основе текстуры рис. 16 шахматную доску. Объявим:

```
varTexture = newTHREE.ImageUtils.loadTexture(
'textures/checkerboard.jpg' );
```

Для размножения текстуры предусмотрено три режима, которые задаются числовыми константами. Это:

```
THREE.RepeatWrapping = 1000;
THREE.ClampToEdgeWrapping = 1001;
THREE.MirroredRepeatWrapping = 1002;
```

В первом случае, который нам сейчас и понадобится, текстура повторяется обычным образом. Во втором случае текстура прижимается к левому нижнему углу, а в третьем – множится с зеркальным отображением. Итак, укажем:

```
Texture.wrapS = THREE.RepeatWrapping;
Texture.wrapT = THREE.RepeatWrapping;
```

Чтобы получилась шахматная доска, наш рисунок нужно повторить четыре раза по горизонтали и четыре раза по вертикали:

```
Texture.repeat.set( 4, 4 );
```

Можно указать также смещение текстуры по горизонтали и вертикали. В скобках указываются не пиксели, а доли исходного изображения, например:

```
Texture.offset.set( 0.5, 0 );
```

Здесь рисунок сдвигается вправо на величину, равную половине его ширины. Осталось создать материал на основе заданной текстуры и геометрию нашей будущей доски, которую мы построим как кусок плоскости с помощью PlaneGeometry (подробнее в п. 3.3.2):

```
var Material = new THREE.MeshBasicMaterial(
{ map: Texture, side: THREE.DoubleSide } );
var Geometry = new THREE.PlaneGeometry(300, 300, 1, 1);
```

Создаем собственно шахматную доску с указанными геометрией и материалом, ее позицию и разворачиваем:

```
var checkerboard = new THREE.Mesh(Geometry, Material);
checkerboard.position.y = - 1;
checkerboard.rotation.x =  Math.PI / 2;
scene.add(checkerboard);
```

Результат на следующем рисунке:

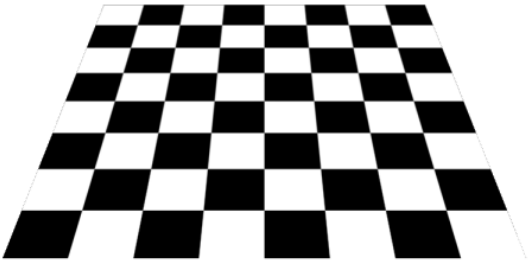

Рис. 19. Шахматная доска

1.8.2. Создание модели обращения Земли вокруг Солнца

В этой модели мы наложим текстуру на сферу. Подготовим бесплатную текстуру Земли небольшого разрешения earth.jpg в папке textures. Объявим глобальные переменные для объекта Земли, ее угла и радиуса орбиты:

```
var Earth, phi, R_orbit = 400;
```

Движение по окружности легко представит себе человек, хорошо знающий математику. Достаточно вспомнить уравнение окружности радиуса R в полярных координатах:

$$\begin{aligned} x &= R\cos\varphi, \\ y &= R\sin\varphi, \end{aligned} \quad 0 \le \varphi \le 2\pi.$$

У нас R это R_orbit, φ = phi. Создаем текстуру и материал на ее основе, объявляем сферическую геометрию:

```
var Texture = THREE.ImageUtils.loadTexture( "textures/earth.jpg" );
var Material = new THREE.MeshLambertMaterial( { map: Texture } );
var Geometry = new THREE.SphereGeometry( 50, 64, 32);
```

Создаем макет планеты Земля, указываем ее начальное положение, наклон оси (примерно 23.5 градусов):

```
Earth = new THREE.Mesh(Geometry, Material);
phi = - Math.PI/3;
        Earth.position.x = R_orbit * Math.cos( phi );
        Earth.position.z = R_orbit * Math.sin( phi );
        Earth.position.y = 0;
Earth.rotation.z = -23.5 * Math.PI/180;
scene.add( Earth );
```

Теперь для движения Земли по окружности (вокруг Солнца) достаточно внутри функции render() задать увеличение угла phi:

```
Earth.position.x = R_orbit * Math.cos( phi );
Earth.position.z = R_orbit * Math.sin( phi );
phi = phi + 0.002;
```

Также наша Земля крутится вокруг своей оси:

```
Earth.rotation.y = Earth.rotation.y + 0.02;
```

Конечно, истинные пропорции для размеров, расстояния, и времени не соблюдены. Теперь создадим Солнце:

```
var Sun = new THREE.Mesh(
        new THREE.SphereGeometry( 24, 32, 16 ),
        new THREE.MeshBasicMaterial( { color: 0xffaa00 } )
);
scene.add( Sun );
Sun.position = light.position;
```

В этом случае наше Солнце будет выглядеть как обычный желтый шарик. Можно добавить эффект сияния[6]. Подготовим текстуру в формате png (файл glow.png) вида

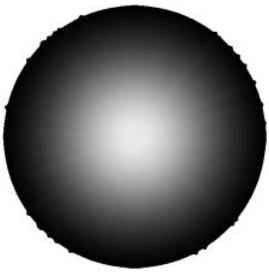

Рис. 20. Текстура для «сияния»

На его создадим спрайт, который разместим там же, где и наше Солнце. Для создания спрайта используется класс Sprite с материалом SpriteMaterial:

```
var spriteMaterial = new THREE.SpriteMaterial(
    {
      map: new THREE.ImageUtils.loadTexture( 'textures/glow.png' ),
           useScreenCoordinates: false,
             alignment: THREE.SpriteAlignment.center,
               color: 0xffaa00, transparent: false,
                 blending: THREE.AdditiveBlending
      });
      var sprite = new THREE.Sprite( spriteMaterial );
      sprite.scale.set(100, 100, 1.0);
      Sun.add(sprite);
```

[6]http://stemkoski.github.io/Three.js/Simple-Glow.html

Осталось добавить «небо» (точнее космос) и можно визуализировать орбиту. Для неба обычно создается большой куб синего цвета, который должен быть виден изнутри (side: THREE.BackSide):

```
var skyBoxGeometry =
                new THREE.BoxGeometry( 3000, 3000, 3000 );
var skyBoxMaterial = new THREE.MeshBasicMaterial(
                  { color: 0x203668, side: THREE.BackSide } );
var skyBox =
        new THREE.Mesh( skyBoxGeometry, skyBoxMaterial );
scene.add(skyBox);
```

Для создания траектории используем orbit класс CircleGeometry (подробнее о классе в пункте 3.3.5):

```
orbit_geometry = new THREE.CircleGeometry(R_orbit, 64);
orbit_geometry.vertices[0]['x'] = R_orbit;
orbit_material = new THREE.LineBasicMaterial(
                                  { color: 0xffcc00, linewidth: 1 } );
orbit = new THREE.Line( orbit_geometry, orbit_material );
orbit.rotation.x = Math.PI/2;
scene.add( orbit );
```

Здесь нам пришлось спрятать вертексную линию (во второй строке кода). И, наконец, результат (рис. 21):

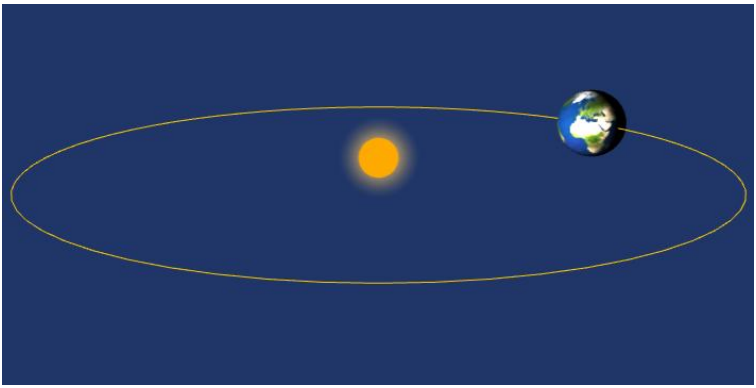

Рис. 21. Модели Земли и Солнца

1.9. Создание структурных объектов

Иногда приходится создавать сложную фигуру из нескольких простых объектов. Тогда, по идее, чтобы они двигались как единое целое, нужно каждому объекту задавать одно и тоже, общее для всех, правило движения. Конечно, это было бы довольно утомительно. Возникает вопрос, можно ли сгруппировать несколько объектов в одну группу? Чтобы они, например, двигались как единое целое.

Для решения этой проблемы можно использовать класс трехмерных объектов Object3D. При создании сложной фигуры все ее «запчасти» добавляются не на сцену, а «внутрь» объекта, и лишь потом объект выводится на сцену.

Создадим, к примеру, модель снеговика из нескольких простых фигур (рис. 22). Для этого сначала объявим глобальный объект:

```
varSnowman = newTHREE.Object3D();
```

Как видно из рис. 22, основу снеговика составляют пять сфер. Начнем с нижней сферы-основания:

```
varmaterial =
        newTHREE.MeshBasicMaterial( {color: 0x33CCFF} );
var geometry = new THREE.SphereGeometry( 60, 36, 36 );
var sphere1 = new THREE.Mesh( geometry, material);
sphere1.position.set( 0, 60, 0 );
```

Рис. 22. Снеговик

Теперь добавляем эту сферу, но не на сцену, а в наш объект Object3D:

```
Snowman.add( sphere1 );
```

Аналогично добавим остальные сферы

```
var material =  new THREE.MeshBasicMaterial({color: 0x00edff});
var geometry = new THREE.SphereGeometry( 44, 36, 361 );
var sphere2 = new THREE.Mesh( geometry, material);
sphere2.position.set( 0, 140, 0 );
Snowman.add( sphere2 );

var material = new THREE.MeshBasicMaterial({color: 0xafeeee});
var geometry = new THREE.SphereGeometry( 32, 36, 36 );
var sphere3 = new THREE.Mesh( geometry, material);
sphere3.position.set( 0, 206, 0 );
Snowman.add( sphere3 );

varmaterial=newTHREE.MeshBasicMaterial({color:0x1560bd});
var geometry = new THREE.SphereGeometry( 16, 16, 16 );
var sphere4 = new THREE.Mesh( geometry, material);
sphere4.position.set( -50, 156, 0 );
Snowman.add( sphere4 );

sphere5 = sphere4.clone();
sphere5.position.set( 50, 156, 0 );
Snowman.add( sphere5 );
```

Поскольку «руки» снеговика одинаковые, левую руку мы создали «клонированием» правой. Далее изобразим нос в виде конуса:

```
varmaterial =
newTHREE.MeshLambertMaterial({color: 0xf36223});
var geometry = new THREE.CylinderGeometry( 1, 7, 40, 8 );
var nose = new THREE.Mesh( geometry, material);
nose.position.set(0, 202, 45); nose.rotation.x = Math.PI/2;
Snowman.add( nose );
```

Добавляем ведро:

```
var material =
        new THREE.MeshLambertMaterial({color: 0x6600ff});
var geometry = new THREE.CylinderGeometry( 24, 34, 60, 18 );
```

```
var bucket = new THREE.Mesh( geometry, material);
bucket.position.set(0, 249, -12); bucket.rotation.x = -Math.PI/11;
Snowman.add( bucket );
```

Два глаза:

```
var material =
        new THREE.MeshLambertMaterial({color: 0x000000});
var geometry = new THREE.SphereGeometry( 5, 50, 11 );
var eye1 = new THREE.Mesh( geometry, material);
eye1.position.set( -15, 213, 27 );
Snowman.add( eye1 );

var eye2 = eye1.clone();
eye2.position.set( 15, 213, 27 );
Snowman.add( eye2 );
```

И, наконец, рот:

```
var material = new THREE.MeshBasicMaterial({color: 0x560319});
var geometry = new THREE.CircleGeometry( 10, 10, 0, Math.PI );
var mouth = new THREE.Mesh( geometry, material);
mouth.rotation.z = Math.PI;
mouth.position.set( 0, 194, 27 );
Snowman.add( mouth );
```

Укажем позицию снеговика и добавим его на сцену. Объект Object3D строится от нулевой координаты, т.е. при указании координат позиции объекта в указанном месте будет находиться не центр объекта, (как, например, у куба), а его самая нижняя часть:

```
Snowman.position.set( 0, -100, 0 );
scene.add( Snowman );
```

Снеговик готов (рис. 22). Если теперь в рендере указать

```
Snowman.position.z+= 0.5;
```

то наш снеговик будет двигаться «на нас» как единое целое.

Замечание. Последнюю строчку кода движения можно изменить на

```
Snowman.translateZ( 0.5 );
```

Метод translateZ(dist), очевидно, увеличивает третью координату на число dist (может быть и меньше нуля). Аналогичный смысл имеют методы translateY и translateZ.

Класс Mesh создан на основе класса Object3D, поэтому также имеет эти методы.

1.10. Добавление 3D текста

Для создания трехмерного текста предназначен класс TextGeometry (наследник класса ExtrudeGeometry). Конструктор:

```
TextGeometry( text, parameters );
```

Здесь text – содержимое текста, параметры parameters включают:

– size: размер текста;

– height: толщина текста;

– curveSegments: количество точек (сегментов) кривой при рисовании буквы;

– font: название шрифта;

– weight: тип шрифта ("normal", "bold");

– style: стиль шрифта ("normal", "italics");

Далее, можно включить фаску (т.е. добавить «скосы»):

– bevelEnabled: включение фаски (при true);

– bevelThickness: глубина фаски;

– bevelSize: ширина фаски.

Трехмерный текст получается из плоского «экструзией» («выдавливанием») на величину height. Можно использовать разные материалы для передней (задней) части текста и полученной экструзией боковой части. Нужно создать массив материалов и указать:

– material: индекс материала для переднего и заднего планов;

– extrudeMaterial: индекс материала для плана экструзии и плана фаски.

При этом обязательно нужно скачать и подключить скрипт, генерирующий шрифты. На сайте threejs.org это, например, файл helvetiker_regular.typeface.js[7]. Правда, в нем отсутствуют русские символы. Чтобы получить скрипт с нужным шрифтом, можно найти соответствующий бесплатный TrueType шрифт с расширением «.ttf», содержащий все необходимые символы, и сгенерировать скрипт на

[7]URL: http://threejs.org/examples/fonts/helvetiker_regular.typeface.js

сайте http://typeface.neocracy.org/fonts.html. Автор подготовил файл arial_regular.typeface.js в папке fonts. Подключим его на главной странице:

```
<scriptsrc="../fonts/arial_regular.typeface.js"></script>
```

Теперь можно работать с текстом со шрифтом arial, содержащим и русские буквы. Зададим текст и его геометрию:

```
vartext = "СНЕГ";
vartext_geometry = newTHREE.TextGeometry( text,
    {
      size: 24,
      height: 5,
      curveSegments: 4,
      font: "arial",
      style: "normal",
      bevelEnabled: true,
      bevelThickness: 2,
      bevelSize: 1,
    });
```

Подготовим материал лилового цвета:

```
var text_Material =
        new THREE.MeshPhongMaterial({ color: 0x62254a });
```

Теперь создадим, собственно, трехмерный объект – «мэш» с построенными геометрией и материалом:

```
var text3D = new THREE.Mesh( text_geometry, text_Material );
```

При рендеринге текст выстраивается слева направо от указанной позиции. Если хочется поместить текст по центру экрана, нужно найти его середину. В этом нам поможет метод computeBoundingBox(), который способен вычислить границы геометрии. Итак,

```
text_geometry.computeBoundingBox();
var text_Width = text_geometry.boundingBox.max.x -
                 text_geometry.boundingBox.min.x;

text3D.position.set( -0.5 * text_Width, 0, 0 );
scene.add( text3D );
```

Результат готов:

Рис. 23. 3D текст

Сделаем этот же пример, но уже с наложением текстуры. Загрузим произвольную «снежную» бесплатную картинку (файл sneg.jpg), и размножим ее:

```
var Texture = THREE.ImageUtils.loadTexture( 'textures/sneg.jpg' );
Texture.wrapS = Texture.wrapT = THREE.RepeatWrapping;
Texture.repeat.set( 0.05, 0.05 );
var text_Material =
        new THREE.MeshBasicMaterial( { map: Texture } );
```

С этим материалом текст будет выглядеть как:

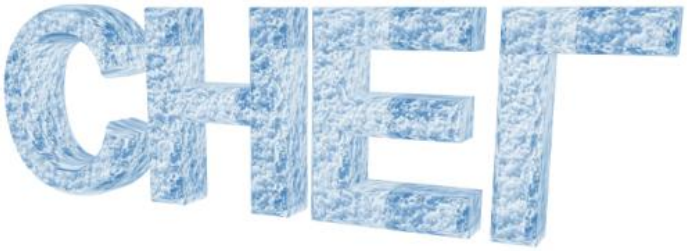

Рис. 24. 3D текст с текстурой

И, наконец, рассмотрим еще один пример, уже с двумя материалами. Первый материал будет предназначен для переднего и заднего планов, а другой для фаски – «боковой» части букв. Подготовим картинки – файлы mtr.jpg и extrmtr.jpg (при этом не забываем о том, что **размеры сторон размножаемых текстур должны быть равны степени двойки**) (рис. 25):

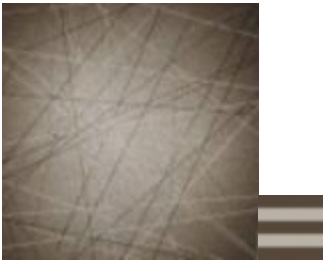

Рис. 25. Текстуры 3D текста

На их основе подготовим две текстуры Texture и ExtrTexture, потом, соответственно, два материала, Material и extrMaterial, и закинем их в массив materials:

```
var Texture = THREE.ImageUtils.loadTexture( 'textures/mtr.jpg' );
Texture.wrapS = Texture.wrapT = THREE.RepeatWrapping;
Texture.repeat.set( 0.05, 0.05 );
var Material = new THREE.MeshBasicMaterial( { map: Texture } );

var ExtrTexture =
        THREE.ImageUtils.loadTexture( 'textures/extrmtr.jpg' );
ExtrTexture.wrapS = ExtrTexture.wrapT
                                                =
THREE.RepeatWrapping;
ExtrTexture.offset.set( 0, 0.7 );
ExtrTexture.repeat.set( 0, 0.13 );
var extrMaterial =
        new THREE.MeshBasicMaterial( { map: ExtrTexture } );

var materials = [ Material, extrMaterial ];
```

Теперь объявляем материал текста:

```
var textMaterial = new THREE.MeshFaceMaterial( materials );
```

и геометрию (с содержанием будущего текста):

```
vartext = "ШОКОЛАД";
var text_geometry = new THREE.TextGeometry( text,
    {
      size: 24,
      height: 5,
```

```
    curveSegments: 4,
    font: "arial",
    style: "normal",
    bevelEnabled: true,
    bevelThickness: 2,
    bevelSize: 1,
    material: 0,
    extrudeMaterial: 1
});
```

Здесь добавились две новые строки. Указываем, что материал Material передних и задних частей располагается в массиве materials первым, а материал боковой части и фаски extrMaterial – вторым (нумерация, естественно, с нуля). Итог:

Рис. 26. 3D текст с двумя текстурами

1.11. Добавление теней

Для наложения теней на сцене нужно предварительно «включить» их в рендере:

```
renderer.shadowMapEnabled = true;
```

Далее нужно включить образование теней у источника света:

```
light.castShadow = true;
```

Это же свойство castShadow нужно включить у тех тел, от которых требуется отбрасывание теней. Рассмотрим пример. Пусть дана сфера:

```
varmaterial =
    newTHREE.MeshLambertMaterial( { color: 0x33CCFF } );
```

```
var geometry = new THREE.SphereGeometry(20, 50, 50);
var sphere1 = new THREE.Mesh( geometry, material );

sphere1.position.set( 0, 60, 0 );
sphere1.castShadow = true;
scene.add ( sphere1 );
```

Расположим под нею две плоскости:

```
var planeMaterial1 = new THREE.MeshLambertMaterial(
                    { color: 0x9999ff, side: THREE.DoubleSide });
var planGeo1 = new THREE.PlaneGeometry( 200, 200, 1, 1);

var plane1 = new THREE.Mesh(planGeo1,  planeMaterial1);
plane1.rotation.x = Math.PI/2;
scene.add( plane1 );

var planeMaterial2 = new THREE.MeshLambertMaterial(
                      { color: 0xff00ff, side: THREE.DoubleSide });
var planGeo2 = new THREE.PlaneGeometry( 300, 300, 1, 1);
var plane2 = new THREE.Mesh(planGeo2,  planeMaterial2);

plane2.position.set(0,-120,0);
plane2.rotation.x = Math.PI/2;
scene.add( plane2 );
```

Но как видим на рисунке, теней пока нет:

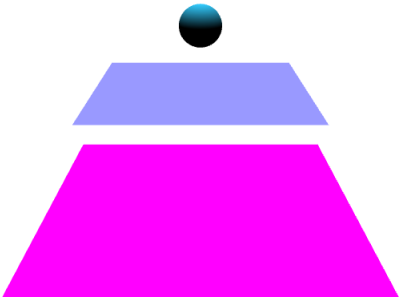

Рис. 27.Тени

Чтобы плоскости воспринимали тени, нужно включить у них свойство receiveShadow:

```
plane1.receiveShadow = true;
plane2.receiveShadow = true;
```

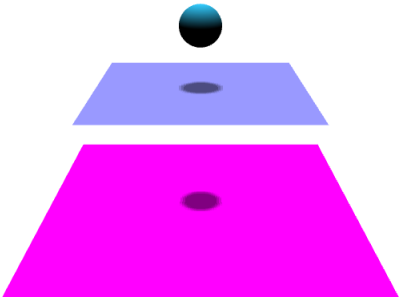

Рис. 28. Тени

Как видим, результат есть, но не совсем удовлетворительный. Решение простое: включить свойство castShadow у первой плоскости:

```
plane1.castShadow = true;
```

Результат:

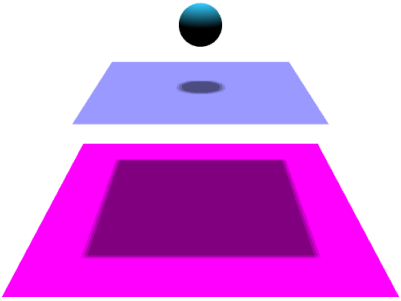

Рис. 29. Тени

2. ДОБАВЛЕНИЕ ИНТЕРАКТИВНОСТИ

2.1. Управление клавиатурой

2.1.1. Простой пример

Управление клавиатурой достигается несложно. В Интернете можно найти довольно много разных примеров. Здесь мы воспользуемся решением Артура Шрайбера[8] (Arthur Schreiber). Открываем файл mainapp.js и создаем класс Key:

```
var Key =
{
   _pressed: {},

   A:      65,
   W:      87,
   D:      68,
   S:      83,
   SPACE: 32,
   VK_LEFT:      37,
   VK_RIGHT:     39,
   VK_UP:        38,
   VK_SPACE:     32,
   VK_ENTER:     13,

isDown: function(keyCode) { return this._pressed[keyCode]; },
onKeydown: function(event) {
                        this._pressed[event.keyCode] = true; },
onKeyup: function(event) { delete this._pressed[event.keyCode]; }
};
```

Поскольку JavaScript работает с клавиатурой через коды клавиш, здесь указываются названия (для наглядности) клавиш и их коды. При необходимости можно добавить свои клавиши или удалить ненужные.

Теперь добавляем созданные обработчики событий на веб-страницу:

```
window.addEventListener( 'keyup',
            function(event) { Key.onKeyup(event); }, false);
```

[8]URL: http://nokarma.org/2011/02/27/javascript-game-development-keyboard-input/index.html

```
window.addEventListener('keydown',
            function(event) { Key.onKeydown(event); }, false);
```

Теперь для того, чтобы определить нужные действия при нажатии на клавишу, достаточно использовать конструкцию типа:

```
if (Key.isDown(Key.VK_LEFT)) // если нажата стрелка «влево»
                        { действия}
```

Подобную проверку нужно осуществлять внутри функции render(). Создадим пример кубика, который под управлением стрелок движется вправо и влево, а при нажатии на enter подскакивает.

Сначала внутри функции init() создаем куб обычным образом:

```
var geometry = new THREE.BoxGeometry( 50, 50, 50);
var material = new THREE.MeshNormalMaterial({color: 0x00ff00});
Cube = new THREE.Mesh( geometry, material );
scene.add( Cube );
```

Далее создадим отдельную функцию dynamo():

```
function dynamo()
{
    if (Key.isDown(Key.VK_LEFT))   // движение влево
        {  Cube.position.x -= 10; }

    if (Key.isDown(Key.VK_RIGHT))  // движение вправо
        { Cube.position.x += 10; }

    if (Key.isDown(Key.VK_ENTER))  // подскок
        {  Cube.position.y += 10; }

}
```

Эту функцию, как уже было сказано, мы будем вызывать внутри render():

```
functionrender()
{
    dynamo();  // управление с помощью клавиатуры
    controls.update(); // управление камерой с помощью мышки
    renderer.render(scene, camera);
}
```

Готово, теперь наш кубик управляется клавиатурой.

Замечание. При реализации событий одновременного нажатия нескольких клавиш автор столкнулся с неожиданной проблемой. На обычных (не игровых) клавиатурах не работают нажатые вместе «влево+вперед+пробел». Тогда как сочетание «вправо+вперед+пробел» прекрасно работает. Из-за этого автор сначала думал, что проблема в его коде.

Возможные пути решения проблемы:

– купить игровую клавиатуру, убедившись что у неё нет такой проблемы; не решает проблему при использовании вашего приложения другими юзерами;

– отключить Num Lock и использовать вместо стрелочек кнопки 8 – «вперед», 2 – «назад», 4 – «влево», 6 – «вправо»;

– не использовать стрелки, а «вешать» события, например, на кнопки W, A, S, D.

Другая проблема может заключаться в том, что события клавиатуры будут обрабатываться непрерывно. Легко представить себе ситуацию, например, стрельбу, когда необходимо ограничить частоту возможности произвести выстрел. Возможное решение этой проблемы мы рассмотрим в п. 3.4.2.

2.2. Обработка событий мышки

2.2.1. Обработка клика мышки по трехмерным объектам

Рассмотрение обработчика мыши я начал именно с этого пункта, поскольку он мне кажется наиболее содержательным. На сайте http://threejs.org имеется прекрасный пример с разноцветными кубами[9] (рис. 30).

При нажатии на куб он случайным образом меняет свой цвет, и в месте клика наносится круглая черная точка (шарик) (рис. 30).

Как это работает? Идея несложная. Нужно направить в точке клика луч в глубь сцены и посмотреть, какие объекты сцены он пересекает. Если луч по пути пересек какой-то объект, значит, клик «попадает» по нему. При этом луч может пересечь по пути несколько объектов. Всех их мы будем запоминать в том порядке, в котором они нам встретились. Сначала первый встреченный объект, потом второй, и т.д.

[9]URL: http://threejs.org/examples/#canvas_interactive_cubes

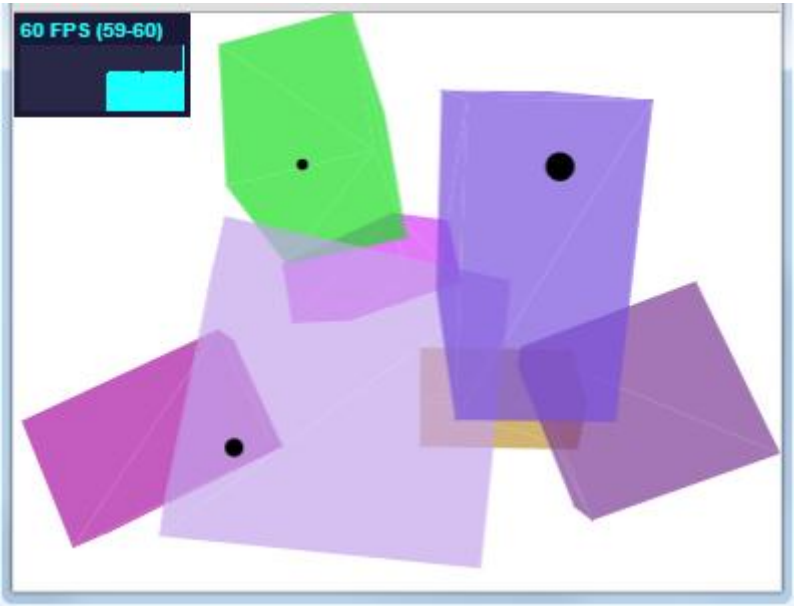

Рис. 30. «Интерактивные» кубы

По идее, нужно перебрать абсолютно все объекты сцены на предмет проверки пересечения с этим лучом. Это может занять довольно длительное время. На самом деле обычно требуется, чтобы на клик реагировали далеко не все, а только определенные объекты сцены. Поэтому на практике те элементы (объекты), клики на которые нужно отлавливать, помещаются в специальный массив. Объявим, например, глобальный массив objects:

```
varobjects = [ ];
```

В нашем примере отлавливаются клики на кубоидах. Поэтому все они будут закидываться в этот массив. Как обычно, объявляем геометрию, материал и т.д.:

```
vargeometry = newTHREE.BoxGeometry( 100, 100, 100 );

for ( vari = 0; i< 10; i ++ )
{
    ... // создаем кубоиды
    var object = new THREE.Mesh( geometry,
```

```
        new THREE.MeshBasicMaterial(
            { color: Math.random() * 0xffffff, opacity: 0.5 } ) );
        ... // задаем их растяжения, повороты, расположение
        scene.add( object );// кубоид выводим на сцену
        objects.push( object );// кубоид запоминаем в массиве objects
    }
```

Теперь наши кубоиды сидят в массиве objects, и при щелчке мыши каждый кубоид будет проверяться, попал ли щелчок по нему или нет.

Как уже было сказано, для того, чтобы определить объекты, на которые пришелся щелчок мышки, нужно в точке клика направить луч от «нас» в «глубь» сцены. Где начало такого луча? Поразмыслив немного, приходим к выводу, что это должно быть положение камеры. Камера – это и есть «мы» (рис. 31):

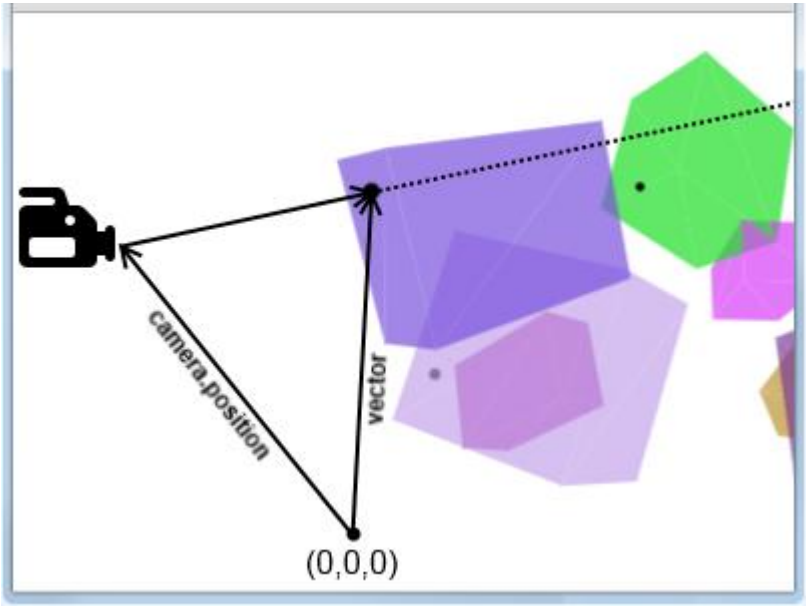

Рис. 31. «Интерактивные» кубы

Но, чтобы провести прямую, как мы знаем еще из школьной геометрии, нужно две точки. Вторая точка связана с координатами места клика мыши на экране, которые определяются как числа event.clientX, event.clientY. Эти координаты двумерные, а нам нужно

их перевести в трехмерные координаты сцены. Для осуществления такого перевода предусмотрен класс Projector() с методом unprojectVector. Итак, создаем:

```
projector = new THREE.Projector();
```

Поскольку мы не рассматривали всех тонкостей отображения сцены камерой, примем способ получения «мировых» координат мыши «как есть»:

```
var vector = new THREE.Vector3(
        ( event.clientX / window.innerWidth ) * 2 - 1,
        - ( event.clientY / window.innerHeight ) * 2 + 1,
        0.5 );
projector.unprojectVector( vector, camera );
```

Теперь, зная две точки, через которые он проходит, можно построить сам луч. Для построения луча используется класс Raycaster, для которого требуется указать начальную точку луча и его направление (направляющий вектор):

```
var raycaster = new THREE.Raycaster(
   camera.position, vector.sub( camera.position ).normalize() );
```

Направляющий вектор получается вычитанием координат мышки и камеры (a.sub(b) дает вектор a - b) с последующим приведением к «нормальному» виду (получение вектора единичной длины той же направленности, путем деления вектора на его длину).

Теперь нужно найти объекты, которые пересек наш луч. Класс Raycaster делает это за нас и запоминает найденные объекты в специальном массиве. Обратиться к этому массиву можно с помощью метода intersectObjects:

```
varintersects = raycaster.intersectObjects( objects );
```

Тогда условие, попал ли клик на какие-либо объекты, запишется в виде:

```
if ( intersects.length> 0 ){ … }
```

В массиве intersects теперь будет храниться следующая информация об объектах:

– object: объект, по которому пришелся клик;

– distance: расстояние до точки клика (по объекту!);
– point: трехмерные координаты точки клика;
– face: грань объекта, по которому произвели клик;
– faceIndex: номер этой грани.

Объекты находятся в массиве по порядку встречи с лучом (нумерация, естественно, с нуля). Поэтому объект, на который ткнули мышкой, будет первым. Тогда, например, чтобы определить координаты точки, на которую попал клик, можно как:

```
intersects[0].point;
```

В нашем примере куб красится в новый произвольный цвет:

```
if ( intersects.length> 0 ) // если массив не пуст
{
intersects[0].object.material.color.setHex( Math.random()*0xffffff );
    … // другие действия
}
```

Если вдруг нужно «поразить» все кубоиды, встретившиеся на пути луча, то можно использовать конструкцию:

```
for ( var i in intersects )
{
intersects[ i ].object.material.color.setHex( Math.random()*0xffffff );
    … // другие действия
}
```

Замечание. При изменении размера окна браузера камера сдвинется, и наш raycaster будет «промахиваться». Для решения этой проблемы можно использовать метод onWindowResize, обновляющий камеру после события изменения размера окна браузера. Создается функция обновления камеры:

```
functiononWindowResize()
{
    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();
    renderer.setSize( window.innerWidth, window.innerHeight );
}
```

Теперь внутри функции init() добавим эту реакцию на событие изменения размера окна:

```
window.addEventListener( 'resize', onWindowResize, false );
```

Если вдруг и окно сцены не равно размеру всего окна браузера, то, во избежание трудоемких вычислений положения камеры, проще засунуть окно приложения в отдельный фрейм.

Здесь мы рассмотрели методологию создания обработчика клика мышки на объекте. В следующем пункте рассмотрим пример применения.

2.2.2. Создание виртуального музыкального инструмента

Рассмотрим пример создания виртуального «пианино», на котором отработаем сразу два события мышки: наведение и клик на клавишах инструмента.

Наш музыкальный инструмент будет иметь семь клавиш:

```
var key_count = 7;
```

Нам нужно будет отслеживать выделенный объект SELECTED и номер кликнутой клавиши index, чтобы проиграть соответствующий звук:

```
varindex, SELECTED = null;
varobjects = [ ];
```

Кроме того, мы снова объявили массив objects для объектов, клик на которые мы будем отслеживать. Естественно, это будут наши клавиши. Положение мышки на экране будет хранить вектор mouse:

```
var mouse = new THREE.Vector2();
```

Изменение координат мышки будет отслеживать функция:

```
function onDocumentMouseMove( event )
{
    mouse.x = ( event.clientX / window.innerWidth ) * 2 - 1;
    mouse.y = - ( event.clientY / window.innerHeight ) * 2 + 1;
}
```

Добавим наш музыкальный инструмент. Для клавиш выберем «блестящий» материал MeshPhongMaterial:

```
var material = new THREE.MeshPhongMaterial(
    { color: 0x6a2089, specular: 0x00b2fc, shininess: 50,
        shading: THREE.FlatShading, blending:
            THREE.NormalBlending, depthTest: true } );
```

Указываются основной цвет color, отраженный цвет specular, величина блеска shininess, метод затенения shading, равный THREE.FlatShading – «простое» постоянное затенение (по умолчанию THREE.SmoothShading – плавное затенение).

Аналогично создадим материал для текущей клавиши. Он будет чуть светлее:

```
var material = new THREE.MeshPhongMaterial(
    { color: 0x00d9ff, specular: 0x00b2fc, shininess: 50,
        shading: THREE.FlatShading, blending:
            THREE.NormalBlending, depthTest: true } );
```

Добавление клавиш и событий оформим в виде отдельной процедуры:

```
function Add_keys()
{
    projector = new THREE.Projector();
    var geometry = new THREE.BoxGeometry( 50, 100, 300 );

    for ( var i = 0; i < key_count; i ++ )
    {
        var object = new THREE.Mesh( geometry, material);
        object.position.set( i * 100 - 350, 0, 0);
        scene.add( object );
        objects.push( object );
    }
    document.addEventListener( 'mousemove',
                        onDocumentMouseMove, false );
    document.addEventListener( 'mousedown',
                        onDocumentMouseDown, false );
    window.addEventListener( 'resize', onWindowResize, false );
}
```

Естественно, функцию Add_keys() вызовем внутри функции init(). Хотим же мы получить следующий результат: при наведении на клавишу она окрашивается в более светлый цвет; при клике же клавиша опускается ниже и проигрывается звук (рис. 32):

Рис. 32. Музыкальный инструмент

Для получения звука напишем небольшую функцию DHTMLSound:

```
function DHTMLSound(url, id, loop)
{
    try
        {
        document.getElementById(id).innerHTML=
            ' <audio src=" ' + url +
            ' " autoplay="autoplay" ' + loop +
            '></audio>';
        }
    catch(exception) {}
}
```

Для ее работы необходимо добавить раздел sound на главную страницу:

```
<span id = "sound"></span>
```

Звук проигрывается простым созданием тега audio с указанным адресом (пути) файла url. В папку snd закинем 7 произвольных звуков с названиями вида k.wav (k меняется от 0 до 6). Тогда для проигрывания звука (один раз) можно написать

```
DHTMLSound('snd/' + String(k) + '.wav', 'sound', '');
```

Для бесконечного проигрывания:

```
DHTMLSound('snd/' + String(k) + '.wav', 'sound', 'loop');
```

Вместе с добавлением клавиш внутри функции Add_keys мы вызвали функцию клика на клавиши onDocumentMouseDown:

```
function onDocumentMouseDown( event ) {

var vector = new THREE.Vector3(
        ( event.clientX / window.innerWidth ) * 2 - 1,
        - ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
projector.unprojectVector( vector, camera );

var raycaster = new THREE.Raycaster(
    camera.position, vector.sub( camera.position ).normalize() );

var intersects = raycaster.intersectObjects( objects );

if ( intersects.length > 0 )
  {
     index = objects.indexOf(intersects[ 0 ].object);
     objects[ index ].position.y = - 40;
     DHTMLSound('snd/' + String(index) + '.wav', 'sound', '');
  }
}
```

Принцип работы этой функции описан в предыдущем параграфе. Скажем лишь, что когда клик попадет по клавише, она опускается ниже на 40 пикселей, и проигрывается соответствующий звук.

Совершенно аналогично отслеживаем наведение мышки на клавишу:

```
function onKeyMouseOver() {

var vector = new THREE.Vector3( mouse.x, mouse.y, 0.5 );
projector.unprojectVector( vector, camera );

var raycaster = new THREE.Raycaster(
camera.position, vector.sub( camera.position ).normalize() );

var intersects = raycaster.intersectObjects( objects );

if ( intersects.length> 0 ) // попали кликом по клавише
  {
    controls.enabled = false; // отключаем перемещение камеры
```

```
    container.style.cursor = 'pointer';
    if ( SELECTED )// какая-то клавиша уже выделена
        {
            if ( SELECTED != intersects[ 0 ].object )
             {
                SELECTED.material = material;
                SELECTED = intersects[ 0 ].object;
                SELECTED.material = selected_material;
             }
        }
  else
        {
            SELECTED = intersects[ 0 ].object;
            SELECTED.material = selected_material;
        }

}

  else
    {
      controls.enabled = true;// включаем перемещение камеры
      container.style.cursor = 'auto';
      if ( SELECTED ) // какая-то клавиша выделена
      {
          SELECTED.material = material; SELECTED = null;
      }
    }

}
```

Осталось регулярно вызывать эту функцию внутри рендеринга:

```
function render()
{
      for ( var i = 0; i < key_count; i ++ )
        {
            if (objects[ i ].position.y < 0) {// клавиша опустилась
               objects[ i ].position.y = objects[ i ].position.y + 1;}
        }

      onKeyMouseOver();
      controls.update();
      renderer.render(scene, camera);
}
```

Здесь мы также поднимаем на место нажатые клавиши. Наш инструмент готов.

Замечание. Клик на клавише мы отлавливали на событии мышки mousedown, а нахождение мышки над клавишей мы отлавливали внутри рендеринга. Конечно, мы могли второе также отлавливать на событии перемещения мышки mousemove. Тогда бы проверка onKeyMouseOver осуществлялась бы реже, но терялась бы и точность из-за того, что наши клавиши движутся. Легко представить себе ситуацию, когда нажатая клавиша поднимается и получает фокус. Тогда, если мышка остается неподвижной, клавиша не поменяет своего цвета.

Если же объекты и сцена неподвижны, то функцию onKeyMouseOver вполне можно вызывать внутри mousemove. Тогда она будет иметь аргумент event, и координаты мышки будут определяться точно также, как и в onDocumentMouseDown.

2.2.3. Перемещение объектов на примере трехмерных шашек

Перемещение объектов мышкой также не вызывает принципиальных трудностей. При реализации бывает удобно не просто перемещать объект за курсором мыши, а перемещать «вдоль» какой-либо плоскости. Например, если это шашки, то их достаточно перемещать вдоль шахматной доски. Итак, создадим шахматную доску с шашками. Для их перемещения понадобится два «рейкастера». Алгоритм примерно следующий:

– первый рейкастер будет отслеживать клик на шашке; как только клик произведен, переменная SELECTED начинает ссылаться на объект шашки;

– отслеживаем перемещение мышки; если SELECTED не null, то «выстреливаем» второй рейкастер в доску и перемещаем шашку в место попадания;

– если мышку отпускаем (и SELECTED не null), шашка остается на новом месте, SELECTED обращаем в null.

При этом шашка не может лежать на доске где попало, поэтому в последнем пункте нужно добавить итоговое смещение к центру ближайшей черной клеточки. Если же положение шашки слишком неопределенно, лучше вернуть ее на место. Потому после клика на шашке будем запоминать ее предыдущее положение (x_previous, z_previous).

Итак, создадим шахматную доску аналогично пункту 1.5. Добавим шашки:

```
projector = new THREE.Projector();
objects.push( checkerboard );

var geometry = new THREE.CylinderGeometry( 22, 22, 16, 36 );
var material = new THREE.MeshPhongMaterial( { color: 0x9b2d30,
      specular: 0x00b2fc, emissive: 0x000000, shininess: 40,
          shading: THREE.FlatShading,
              blending: THREE.NormalBlending, depthTest: true } );

  var x0 = -175; z0 = 175;
  for ( var i = 0; i < key_count; i ++ )
      {
          var object = new THREE.Mesh( geometry, material );

          if (i==4) {x0 = -175 - 7 * 50; z0 = 175 - 50;}
          if (i==8) {x0 = -175 - 16 * 50; z0 = 175 - 2*50;}

          object.position.set( x0 + i * 2*50, 0, z0)
          scene.add( object );
          objects.push( object );
      }
```

Результат:

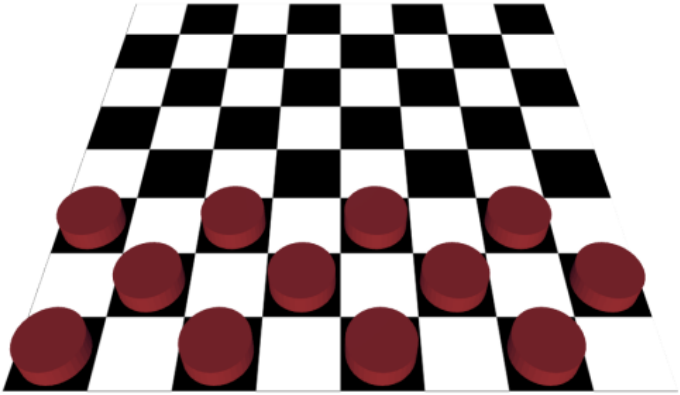

Рис. 33. Шашки

Саму доску мы тоже добавили в массив objects, чтобы, если наша мышка будет над доской, случайно не переместить камеру.

Возможность перемещения камеры останется при движении мышки в стороне от доски. Таким образом, пользователь может расположить доску наиболее удобным для себя образом.

Итак, событие щелчка мышки:

```
function onDocumentMouseDown( event ) {

var vector = new THREE.Vector3(
        ( event.clientX / window.innerWidth ) * 2 - 1,
        - ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
projector.unprojectVector( vector, camera );

var raycaster = new THREE.Raycaster(
    camera.position, vector.sub( camera.position ).normalize() );

var intersects = raycaster.intersectObjects( objects );
if ( intersects.length > 0 )
  {
        controls.enabled = false;
        var k = objects.indexOf(intersects[ 0 ].object);
            if (k==0) { return; } // мышка кликнула над доской
        SELECTED = intersects[ 0 ].object;
        x_previous = SELECTED.position.x;
        z_previous = SELECTED.position.z;
  }
}
```

Перемещаем мышку:

```
function onDocumentMouseMove( event )
{
var vector = new THREE.Vector3(
        ( event.clientX / window.innerWidth ) * 2 - 1,
        - ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
projector.unprojectVector( vector, camera );

var raycaster = new THREE.Raycaster(
    camera.position, vector.sub( camera.position ).normalize() );
if ( SELECTED )
    {
      var intersects = raycaster.intersectObject( checkerboard );
      SELECTED.position.x = intersects[ 0 ].point.sub( offset ).x;
      SELECTED.position.z = intersects[ 0 ].point.sub( offset ).z;
      SELECTED.position.y = 0;
```

```
        container.style.cursor = 'move';
    }
}
```

И, наконец, осталось расписать событие, когда отпускаем мышку. При этом найдем расстояние dr от шашки до центра ближайшего черного квадрата. Если оно покажется нам слишком велико (flag останется равным false), вернем шашку на место:

```
function onDocumentMouseUp( event )
{
  controls.enabled = true;
  flag = false;
  if ( SELECTED )
  {
  // находим центр ближайшей черной клеточки
    for (i=0; i<8; i++)
    {
    for (j=0; j<8; j++)
    {
         if ((i+j)%2==0)
        {
        var dx = Math.abs( SELECTED.position.x - ( -175 + 50 * i ) );
        var dz = Math.abs( SELECTED.position.z - ( 175 - 50 * j ) );
        var dr = Math.sqrt( dx*dx + dz*dz);
        if ( dr < 22 )
            {
                SELECTED.position.x =-175 + 50 * i;
                SELECTED.position.z =175 - 50 * j;
                flag = true; break;
            }
        }
    }
    }

  if (!flag) // черная клеточка слишком далека
  {
    SELECTED.position.x = x_previous;
    SELECTED.position.z = z_previous;
  }
  SELECTED = null;
  }
container.style.cursor = 'auto';
}
```

Теперь нашими шашками можно «ходить»:

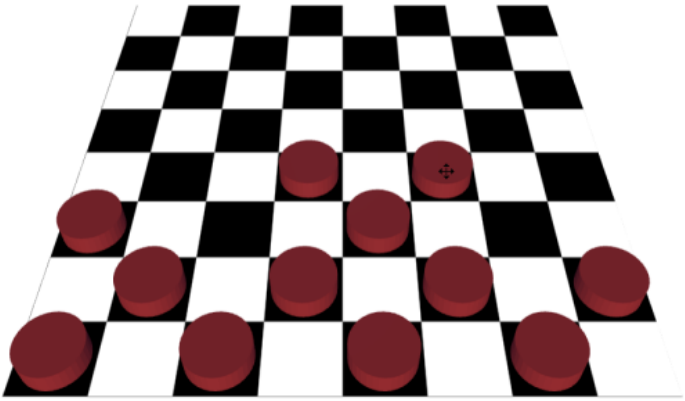

Рис. 34. Шашки

В нашем коде шашки могут перемещаться произвольно и непрерывно по всей доске, но встают всегда в центр черного квадрата. Можно сразу реализовать эффект, когда шашка движется «дискретно» только вдоль центров квадратов. Для этого достаточно перенести цикл нахождения центра ближайшего черного квадрата из функции onDocumentMouseUp в onDocumentMouseMove. Попробуйте и реализуйте наиболее удобный для вас вариант.

2.2.4. Создание куба с интерактивными гранями

При клике на объект можно не только определить его номер, но и номер грани (по которой кликнули), с помощью конструкции

```
intersects[0].faceIndex
```

Воспользуемся этим и создадим разноцветный кубик, при нажатии на грань которого выводится сообщение о соответствующем цвете. Аналогично объявим projector и глобальный массив:

```
var projector = new THREE.Projector();var objects = [ ];
```

куда потом поместим единственный объект – наш куб. Воспользуемся для куба тем же материалом, что и в пункте 1.4.3:

```
vargeometry = newTHREE.BoxGeometry( 200, 200, 200, 1, 1 );
Cube = new THREE.Mesh( geometry,
        new THREE.MeshFaceMaterial(materials) );
scene.add( Cube );
objects.push( Cube);
```

Получается вот такой кубик, который мы еще и будем вращать внутри рендера (рис. 35):

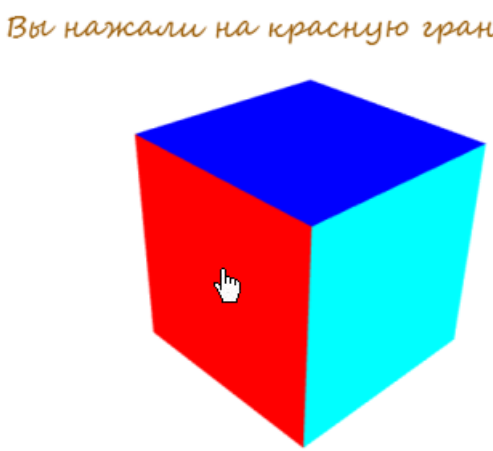

Рис. 35. Интерактивный кубик

При нашем способе объявления кубика на каждой его стороне находится две «полуграни» (рис. 36):

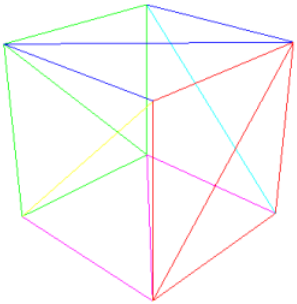

Рис. 36. Грани разноцветного кубика

Поэтому при клике на стороне кубика, на первую его сторону приходятся полуграни с номерами 0 и 1, на вторую – 2 и 3, и т.д. На главной странице создадим раздел для сообщения о цвете куба:

```
<div id = "info" style="position: absolute; ">
Кликните на сторону кубика
</div>
```

Учтем это и создадим функцию клика на кубе:

```
function onDocumentMouseDown( event ) {

var vector = new THREE.Vector3(
        ( event.clientX / window.innerWidth ) * 2 - 1,
        - ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
projector.unprojectVector( vector, camera );

var raycaster = new THREE.Raycaster(
camera.position, vector.sub( camera.position ).normalize() );

var intersects = raycaster.intersectObjects( objects );
if ( intersects.length > 0 )
  {
      var index = Math.floor( intersects[0].faceIndex / 2 );
      switch (index)
        {
        case 0:sssr = 'Вы нажали на красную грань!'; break;
        case 1:sssr = 'Вы нажали на зеленую грань!'; break;
        case 2:sssr = 'Вы нажали на синюю грань!'; break;
        case 3:sssr = 'Вы нажали на пурпурную грань!'; break;
        case 4:sssr = 'Вы нажали на желтую грань!'; break;
        case 5:sssr = 'Вы нажали на светло-голубую грань!';
        break;
        }
      info.innerHTML = sssr;
  }
}
```

Результат на рис. 35. Эффект изменения курсора мышки при наведении на кубик реализуется аналогично.

Замечание. Если увеличить количество сегментов на кубе и взять, например,

```
var geometry = new THREE.BoxGeometry( 200, 200, 200, 2, 2 );
```

то фокус, конечно, не сработает.

3. ИЗУЧАЕМ THREE.JS ДАЛЬШЕ

3.1. Класс Geometry

Класс Geometry является родительским классом для классов, представляющих различные «геометрии». Основными атрибутами этого класса являются наборы вершин и граней. Набор вершин содержится в свойстве-массиве vertices, грани фигуры содержатся в массиве faces. Наконец, правила освещения фигуры реализуются с помощью ее массива вершинных нормалей normals.

Для работы с гранями фигуры в Three.js предусмотрены специальные классы Face3 и Face4. В качестве обязательных аргументов нужно указать номера трех, или, соответственно, четырех вершин (из массива vertices).

Все рассмотренные ранее примеры геометрий (BoxGeometry, CylinderGeometry, SphereGeometry и т.д.) являются наследниками класса Geometry. Перечислим некоторые другие классы для построения фигур (некоторые из них мы рассмотрим подробнее позже):

– класс PlaneGeometry представляет плоский прямоугольник (кусок плоскости) (подробнее в п. 3.3.2);

– для построения кольца используется класс RingGeometry (п. 3.3.6);

– класс CircleGeometry позволяет строить плоский круг (п. 3.3.5);

– класс ExtrudeGeometry позволяет строить цилиндрические поверхности (с закругленными краями) (п. 3.3.8);

– класс LatheGeometry представляет тело вращения (подробнее в п. 3.3.9);

– класс ParametricGeometry представляет параметрическую поверхность (подробнее в п. 3.3.10);

– класс TorusGeometry позволяет строить тор («бублик»);

– класс TubeGeometry представляет «трубу», которая выдавливается вдоль профиля кривой (п. 3.3.3);

– класс ConvexGeometry позволяет генерировать геометрическую форму по данному списку вершин.

Наконец, особый класс BufferGeometry предназначен для работы со статическими объектами и позволяет снижать затраты памяти и процессорного времени, так как сохраняет данные геометрии в буфере.

3.2. Источники света

Один из классов освещения DirectionalLight нам уже знаком. Он представляет собой источник прямого направленного освещения. Наиболее общий вид конструктора класса имеет вид:

```
DirectionalLight( hex, intensity, distance );
```

Здесь hex – значение RGB-компонента цвета. Интенсивность света intensity по умолчанию обычно равна единице (у всех видов освещения). distance равно расстоянию от источника света, на котором интенсивность света станет равной нулю. Пример:

```
vardirectionalLight = newTHREE.DirectionalLight( 0xffffff, 0.5 );
```

Класс PointLight представляет точечный источник света. Его можно сравнить с лампочкой. Конструктор класса имеет аналогичный вид:

```
PointLight( hex, intensity, distance );
```

Пример:

```
var light = new THREE.PointLight( 0xff0000, 1, 100 );
light.position.set( 50, 50, 50 );
scene.add( light );
```

Класс AmbientLight представляет общее освещение, применяемое ко всем объектам сцены. Оно не имеет направления и затрагивает каждый объект сцены в равной степени, независимо от расположения объекта. Соответственно, у этого света нет позиции на оси координат. Конструктор класса:

```
AmbientLight( hex );
```

где hex – значение RGB-компонента цвета. Пример:

```
varlight = newTHREE.AmbientLight( 0x404040 );
scene.add( light );
```

Класс HemisphereLight представляет полусферическое освещение. Оно менее «жестко» в том смысле, что с ним больше переходных тонов и меньше чисто черных тонов. Конструктор класса:

```
HemisphereLight( skyColorHex, groundColorHex, intensity );
```

Класс SpotLight представляет прожектор. Конструктор класса:

```
SpotLight( hex, intensity, distance, angle, exponent );
```

Пример:

```
var spotLight = new THREE.SpotLight( 0xffffff );
spotLight.position.set( 100, 1000, 100 );
scene.add( spotLight ).
```

3.3. Другие фигуры

3.3.1. Рисование осей и координатной сетки

Three.js содержит готовые методы для добавления координатных осей и координатной сетки.

Для изображения осей служит метод AxisHelper. Его единственный аргумент – длина осей. Этот метод рисует сразу все три оси.

Для изображения сетки служит метод GridHelper. Он имеет два параметра – size (длина сетки), и step (шаг сетки). Центр сетки приходится по умолчанию на начало координат. Метод setColors позволяет установить цвет линий сетки. Первый аргумент метода отвечает за цвет центральных линий, второй аргумент – за цвет остальных линий сетки.

Тогда оси и три координатные сетки можно изобразить, например, таким кодом[10]:

```
var axes = new THREE.AxisHelper(300);
    axes.position.set( 0,0,0 );  scene.add(axes);

var gridXZ = new THREE.GridHelper(100, 20);
  gridXZ.setColors( new THREE.Color(0x006600),
new THREE.Color(0x006600) );
  gridXZ.position.set( 100,0,100 );
  scene.add(gridXZ);
```

[10]URL: http://stemkoski.github.io/Three.js/ Helpers.html

```
var gridXY = new THREE.GridHelper(100, 20);
  gridXY.position.set( 100,100,0 );
  gridXY.rotation.x = Math.PI/2;
  gridXY.setColors( new THREE.Color(0x000066),
new THREE.Color(0x000066) );
  scene.add(gridXY);

var gridYZ = new THREE.GridHelper(100, 20);
  gridYZ.position.set( 0,100,100 );
  gridYZ.rotation.z = Math.PI/2;
  gridYZ.setColors( new THREE.Color(0x660000),
new THREE.Color(0x660000) );
  scene.add(gridYZ);
```

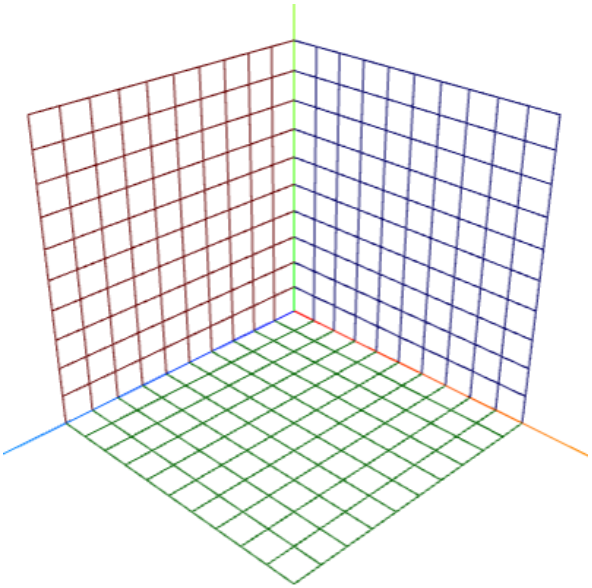

Рис. 37. Система координат

3.3.2. Рисование линий и плоскостей

Для рисования фигур, состоящих из линий в пространстве, используется метод Line(geometry, material). Первый аргумент является экземпляром класса Geometry и содержит набор вершин фигуры. Материал фигуры material выбирается либо

LineBasicMaterial – для сплошных линий, либо LineDashedMaterial – для пунктирных линий.

Изобразим, например, правильный шестиугольник в пространстве (рис. 38):

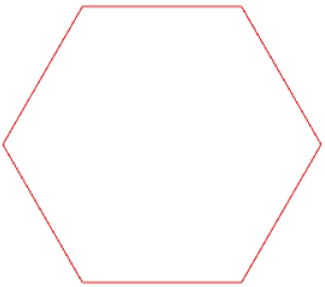

Рис. 38. Плоский шестиугольник

Каждая вершина такого шестиугольника лежит на стороне угла с величиной, кратной 60 градусам (в радианах Math.PI/3). Объявим новую геометрию и материал:

```
vargeometry = newTHREE.Geometry;
var material = new THREE.LineBasicMaterial( { color: 0xcc0000 } );
```

и в цикле добавим все вершины:

```
for (var i=0; i<=6; i++)
{
    var a = new THREE.Vector3(
      50*Math.cos( Math.PI/3*i ), 50*Math.sin( Math.PI/3*i ), 0 );
    geometry.vertices.push( a );
}
```

Для замыкания кривой мы добавили семь вершин, где седьмая совпадает с первой. Теперь создаем фигуру line и добавляем на сцену:

```
varline = newTHREE.Line( geometry, material );
scene.add(line);
```

Результат вы уже видели (рис. 38).

Для изображения прямоугольной части плоскости используется класс PlaneGeometry. Указываем ширину, высоту и количество сегментов:

```
PlaneGeometry( width, height, widthSegments, heightSegments );
```

Допустим часто встречающуюся ситуацию с пересечением прямой и плоскости. Изобразим плоскость:

```
var planeMaterial = new THREE.MeshBasicMaterial({
  wireframe: true, color: 0x9999ff });
var planGeo = new THREE.PlaneGeometry( 100, 100, 1, 1);

var plane = new THREE.Mesh(planGeo,  planeMaterial);

plane.position.set(50,0,50);
plane.rotation.x = Math.PI/2;

scene.add(plane);
```

Здесь мы, для того чтобы оставить только границы плоскости, использовали параметр wireframe, который взяли равным true.

Теперь изобразим прямую из двух кусков, нижнюю часть которой нарисуем пунктиром. При проведении прямой нужно сначала указать два вектора, концы которых лежат на данной прямой, и добавить их в объект класса THREE.Geometry:

```
  var lineGeometry = new THREE.Geometry();
      var a = new THREE.Vector3(50, 60, 50);
     var b = new THREE.Vector3(50,  0 , 50);
  lineGeometry.vertices.push( a, b );
```

Теперь укажем материал прямой и создадим, собственно, прямую:

```
var Material = new THREE.LineBasicMaterial(
{ color: 0xcc0000 } );

var line = new THREE.Line( lineGeometry, Material );
scene.add(line);
```

Аналогично построим вторую прямую.

```
var lineGeometry = new THREE.Geometry();
    var a = new THREE.Vector3(50, 0, 50);
    var b = new THREE.Vector3(50, -40, 50);
lineGeometry.vertices.push( a, b );
```

Поскольку мы ее будем делать пунктирной, обязательно нужно вызвать соответствующий метод:

```
lineGeometry.computeLineDistances();
```

В материале прямой укажем длину пунктира и расстояние между пунктирами:

```
var Material = new THREE.LineDashedMaterial(
{ color: 0xcc0000, dashSize: 4, gapSize: 2 } );
var line = new THREE.Line( lineGeometry, Material );
scene.add(line);
```

Результат на рисунке 39.

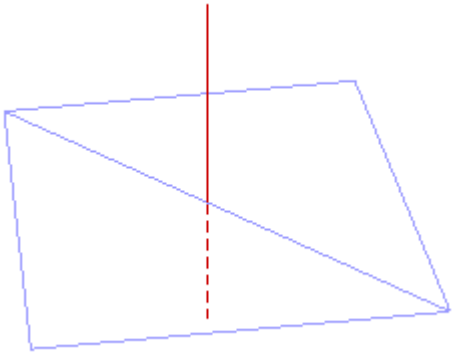

Рис. 39. Прямая и плоскость

Указываются радиус, количество сегментов, начальный угол и величина угла. Отчет идет против часовой стрелки.

3.3.3. Построение параметрических кривых

Самый простой способ построения кривой – с помощью класса Line, практически так же, как мы строили шестиугольник в

предыдущем параграфе. Изобразим, например, цепную линию (рис. 40):

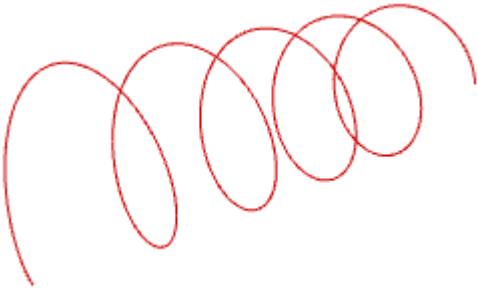

Рис. 40. Цепная линия

Параметрически цепную линию можно задать, например, так:

$$x = a\cos t, \qquad y = a\sin t, \qquad z = bt.$$

Объявим параметры кривой, геометрию и материал:

```
var a = 20, b = 4;
var geometry = new THREE.Geometry;
var material = new THREE.LineBasicMaterial( { color: 0xcc0000 } );
```

Осталось «забить» геометрию вершинами:

```
for (vart = 0; t<= 30; t+= 0.1)
    {
        var vec =
        new THREE.Vector3( a*Math.cos( t ), a*Math.sin( t ), b*t
        );
        geometry.vertices.push( vec );
    }
```

Наконец, создаем линию и добавляем на сцену:

```
var line = new THREE.Line( geometry, material );
scene.add( line );
```

Линия готова (рис. 40). Правда, толщина линии всегда равна единице. Можно воспользоваться классом TubeGeometry. Он представляет собой трубу, которая «выдавливается» вдоль кривой. Для использования этого класса наша кривая должна быть наследником класса Curve:

```
chain = new THREE.Curve.create( function(){},
    function(t)
    {
        t = 12 * Math.PI * t;
        var a = 10, b = 2;
        var x = a*Math.cos( t );
        var y = a*Math.sin( t );
        var z = b*t;
        return new THREE.Vector3(x, y, z).multiplyScalar(2);
    }
);
```

Объявляем экземпляр класса и геометрию:

```
mychain = new chain;
var tubegeo = new THREE.TubeGeometry(
                        mychain, 128, 2, 12, closed = false );
```

Первый аргумент класса TubeGeometry ссылается на нашу кривую, второй отвечает за количество «трубок». Третий аргумент – радиус трубы, четвертый – количество сегментов окружности трубы. Пятый аргумент позволяет замкнуть концы кривой (при closed = true). Осталось подготовить материал:

```
var material = new THREE.MeshPhongMaterial( { color: 0x9b2d30,
    specular: 0xd53e07, emissive: 0x000000, shininess: 40,
    shading: THREE.FlatShading, blending: THREE.NormalBlending,
    depthTest: true } );
```

Наконец, создаем мэш и добавляем его на сцену:

```
vartube = new THREE.Mesh( tubegeo, material );
scene.add( tube );
```

Результат на рисунке (рис. 41):

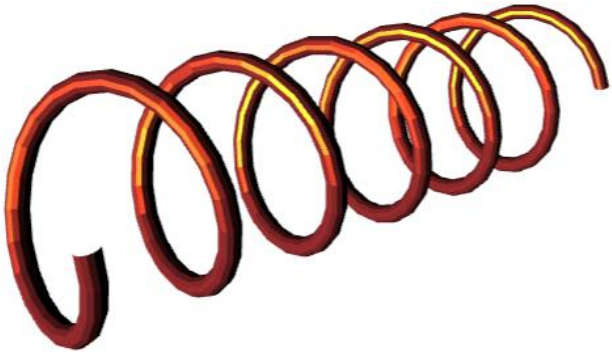

Рис. 41. Цепная линия

3.3.4. Интерполирование и кривые Безье

Интерполирование – это построение непрерывной функции, проходящей через заданные точки. Допустим, задано семейство точек (таблица):

Таблица 1. Значения функции в узлах

x_i	-2	-1	1	4	7	10	13	16
y_i	-1	-2	2	3	-1	2	-4	-2

Для построения функции, проходящей через данные точки, служит функция SplineCurve, аргументы которой – множество координат наших узлов:

```
spline = new THREE.SplineCurve([
                    new THREE.Vector2(-2, -1),
                    new THREE.Vector2(-1, -2),
                    new THREE.Vector2(1, 2),
                    new THREE.Vector2(4, 3),
                    new THREE.Vector2(7, -1),
                    new THREE.Vector2(10, 2),
                    new THREE.Vector2(13, -4),
                    newTHREE.Vector2(16, -2),
                                ]);
```

Теперь для построения кривой можно воспользоваться ее методом getPoint(t). Этот метод возвращает вектор для точки t

кривой, где t находится между 0 и 1. Итак, объявим геометрию и материал:

```
vargeometry = newTHREE.Geometry;
var material = new THREE.LineBasicMaterial( { color: 0xcc0000 } );
```

Закидываем вершины в geometry:

```
for (var i = 0; i <= 1; i+=0.01)
    {
        var x = spline.getPoint( i ).x;
        var y = spline.getPoint( i ).y;
        var vec = new THREE.Vector3( x, y, 0 );
        geometry.vertices.push( vec );

    }
```

Строим линию:

```
var line = new THREE.Line( geometry, material );
scene.add( line );
```

Результат на рисунке (рис. 42):

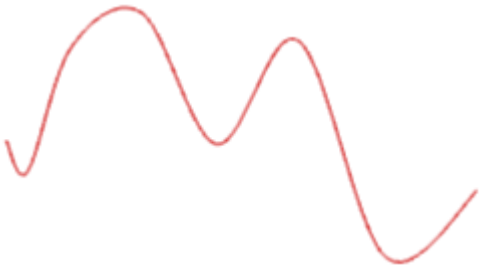

Рис. 42. Интерполяция сплайнами

Если использовать трехмерную интерполяцию с помощью SplineCurve3, то для отображения результата можно снова использовать класс TubeGeometry, как в предыдущем параграфе. Создадим аналогичный материал. Кривую строим на трехмерных векторах:

```
spline = newTHREE.SplineCurve3([
                    newTHREE.Vector3(-2, -1, 0),
                    new THREE.Vector3(-1, -2, 0),
                    new THREE.Vector3(1, 2, 0),
                    new THREE.Vector3(4, 3, 0),
                    new THREE.Vector3(7, -1, 0),
                    new THREE.Vector3(10, 2, 0),
                    new THREE.Vector3(13, -4, 0),
                    new THREE.Vector3(16, -2, 0),
                                    ]);

var tubegeo = new THREE.TubeGeometry( spline, 128, 1, 12 );
var tube = new THREE.Mesh( tubegeo, material );
scene.add( tube );
```

Результат на рисунке (рис. 43):

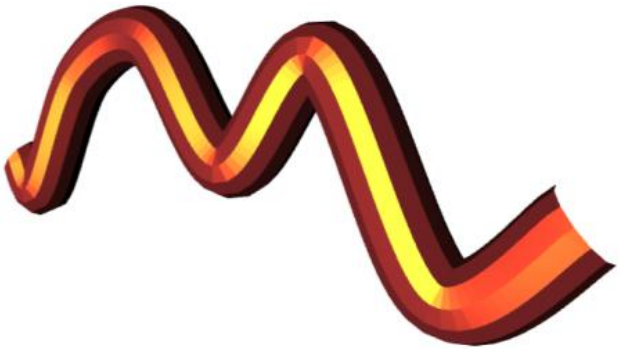

Рис. 43. Интерполяция сплайнами

Замечание. Вместо метода getPoint(t) можно было воспользоваться методом getPoints(N), который возвращает готовый массив из N точек кривой. Тогда следует написать:

```
var points = spline.getPoints( 100 );

for (var i = 0; i < points.length; i++)
    {
        var x = points[i].x;
        var y = points[i].y;
        var vec = new THREE.Vector3( x, y, 0 );
        geometry.vertices.push( vec );
    }
```

Остальное аналогично.

Наконец, для построения квадратичной кривой Безье используется метод QuadraticBezierCurve, параметрами которого являются три двумерных вектора. Изобразим, например, кривую Безье по трем опорным точкам (0,44), (84,80) и (120,0) (рис. 44):

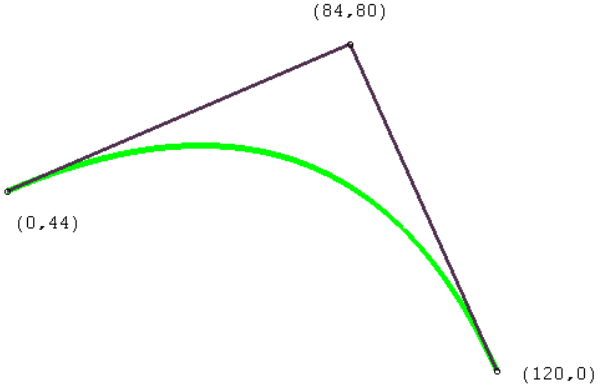

Рис. 44. Кривая Безье

Опишем три трехмерных вектора:

```
v0 = newTHREE.Vector2( 0, 44 );
v1 = new THREE.Vector2( 84, 80 );
v2 = new THREE.Vector2( 120, 0 );
```

Вычисляем кривую Безье:

```
var curve = new THREE.QuadraticBezierCurve( v0, v1, v2 );
```

Осталось описать геометрию и материал и заполнить геометрию точками кривой Безье:

```
var geometry = new THREE.Geometry;
var material = new THREE.LineBasicMaterial( { color:
                                        0xcc0000 } );

for (var i = 0; i <= 1; i+=0.01)
{
        var x = curve.getPoint( i ).x;
        var y = curve.getPoint( i ).y;
        var vec = new THREE.Vector3( x, y, 0 );
```

```
        geometry.vertices.push( vec );

    }

    var line = new THREE.Line( geometry, material );
    scene.add( line );
```

Результат аналогичен рис. 44. Для построения кубической кривой Безье по четырем заданным двумерным векторам используется функция CubicBezierCurve(v0, v1, v2, v3). Кроме того, существуют трехмерные аналоги этих кривых QuadraticBezierCurve3(v0, v1, v2) и CubicBezierCurve3(v0, v1, v2, v3). Их аргументы являются трехмерными векторами.

3.3.5. Построение плоского круга

Класс CircleGeometry позволяет строить плоский круг в пространстве (или его часть). Указываются радиус, количество сегментов, начальный угол и величина угла. Отчет идет против часовой стрелки.

Например, нарисуем два сектора разного цвета:

```
var material = new THREE.MeshBasicMaterial(
                                        {color: 0xe100ff});
var geometry =
     newTHREE.CircleGeometry( 100, 16, 0, Math.PI );
var figure = new THREE.Mesh(geometry, material);
scene.add( figure );//пурпурный сектор

var material = new THREE.MeshBasicMaterial(
                                        { color: 0x177245 });
var geometry =
     newTHREE.CircleGeometry( 100, 16, Math.PI, 2*Math.PI/3);
var figure = new THREE.Mesh(geometry, material);
scene.add( figure );//зеленый сектор
```

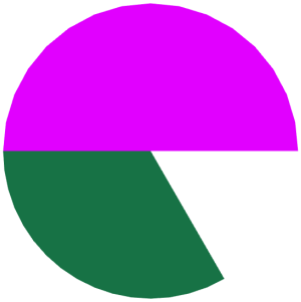

Рис. 45. Секторы

При этом фигура будет видна только с одной стороны. Изменить это можно с помощью параметра материала side:

```
var material = new THREE.MeshBasicMaterial(
                {color: 0xe100ff, side:THREE.DoubleSide}});
```

Теперь наша фигура видна с обеих сторон.

3.3.6. Построение плоского кольца

Для построения кольца используется класс RingGeometry. Синтаксис вызова следующий:

```
RingGeometry( innerRadius, outerRadius, thetaSegments,
phiSegments, thetaStart, thetaLength );
```

Указываются размеры внутреннего и внешнего радиусов, количество угловых и радиальных сегментов, начальный угол и величина угла.

Итак, создадим кольцо (рис. 46), видимое с обеих сторон:

```
var material = new THREE.MeshLambertMaterial(
    {color: 0x082567, side: THREE.DoubleSide});

var geometry = new
            THREE.RingGeometry(20, 100, 20, 10, 0, 2*Math.PI);
ring = new THREE.Mesh( geometry, material );
scene.add( ring );
```

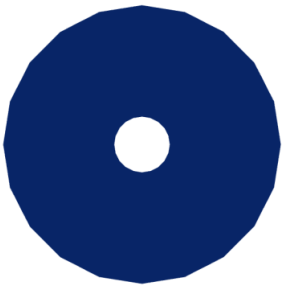

Рис. 46. Кольцо

3.3.7. Криволинейные цилиндры с ExtrudeGeometry

Класс ExtrudeGeometry позволяет рисовать поверхности, полученные «выдавливанием» какой-либо плоской фигуры вдоль прямой. С точки зрения математики, такая фигура называется криволинейным цилиндром. Правда, этот класс также позволяет добавлять фаску. Синтаксис класса:

```
ExtrudeGeometry( shapes, options );
```

Фигура shapes будет образовывать основания полученного цилиндра. Опции класса options во многом аналогичны опциям рассмотренного ранее класса TextGeometry. Это и неудивительно, поскольку последний, как уже было отмечено выше, является его наследником.

Рассмотрим пример рисования трехмерной стрелки. Сначала подготовим чертеж (рис. 47):

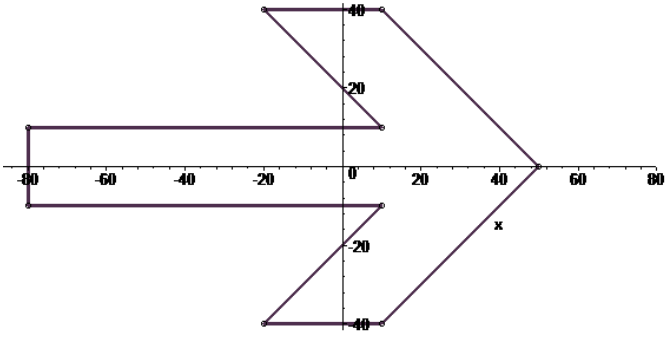

Рис. 47. Схема стрелки

Согласно чертежу построим фигуру arrowShape:

```
var arrowShape = new THREE.Shape();

( function roundedRect( ctx ){

    ctx.moveTo( -80, 10 );
    ctx.lineTo( 10, 10 );
    ctx.lineTo( -20, 40 );
    ctx.lineTo( 10, 40 );
    ctx.lineTo( 50, 0 );
    ctx.lineTo( 10, -40 );
    ctx.lineTo( -20, -40 );
    ctx.lineTo( 10, -10 );
    ctx.lineTo( -80, -10 );
    ctx.lineTo( -80, 10 );

} )( arrowShape);
```

Укажем параметры геометрии:

```
var size = 8;
var extrudeSettings =
    { amount: size, bevelSegments: 8, curveSegments: 32 };
```

Параметр amount отвечает за высоту экструзии. Объявим геометрию:

```
var geometry = new
    THREE.ExtrudeGeometry( arrowShape, extrudeSettings );
```

и материал:

```
var material = new THREE.MeshPhongMaterial({ color:
0x708090,
      specular: 0x708090, ambient: 0x777777, shininess: 50  });
```

Наконец, создаем нашу трехмерную стрелку:

```
var arrow_right = new THREE.Mesh( geometry, material );
arrow_right.position.set( 100, 0,  - size/2 );
scene.add( arrow_right );
```

Сразу создадим вторую стрелку, с той же геометрией, но направленной влево:

```
var arrow_left = new THREE.Mesh( geometry, material );
arrow_left.rotation.y = Math.PI;
arrow_left.position.set( -100, 0,  size/2 );
scene.add( arrow_left );
```

Результат:

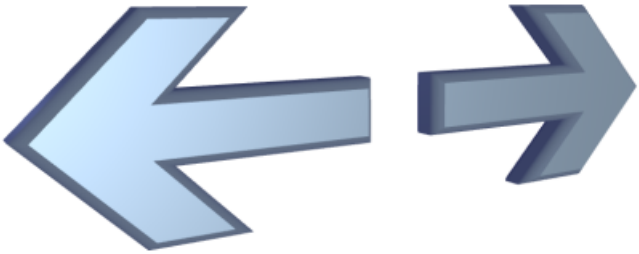

Рис. 48. Трехмерные стрелки

В этом примере мы использовали один материал на всю стрелку. В следующем пункте мы рассмотрим пример с разными материалами для оснований и для фаски.

3.3.8. Рисование параллелепипеда с закругленными краями

Раздел в:

```
var roundedRectShape = new THREE.Shape();
var width = 200, height = 100;

( function roundedRect( ctx, x, y, w, h, R ){

    ctx.moveTo( x, y + R );
    ctx.lineTo( x, y + h - R );
    ctx.quadraticCurveTo( x, y + h, x + R, y + h );
    ctx.lineTo( x + w - R, y + h) ;
    ctx.quadraticCurveTo( x + w, y + h, x + w, y + h - R );
    ctx.lineTo( x + w, y + R );
    ctx.quadraticCurveTo( x + w, y, x + w - R, y );
    ctx.lineTo( x + R, y );
    ctx.quadraticCurveTo( x, y, x, y + R );

} )( roundedRectShape, -width/2, -height/2, width, height, 20 );
```

Настройки геометрии:

```
var extrudeSettings =
{
    amount: 16,
    bevelSegments: 8,
    curveSegments: 32,
    material: 0,
    extrudeMaterial: 1
};
var geometry = new THREE.ExtrudeGeometry(
                            roundedRectShape, extrudeSettings );
```

Опишем материал сторон и экструзии. Возьмем те же текстуры, которые мы брали при создании трехмерного текста (рис. 25):

```
var Texture = THREE.ImageUtils.loadTexture( 'textures/mtr.jpg' );
Texture.wrapS = Texture.wrapT = THREE.RepeatWrapping;
Texture.repeat.set( 0.05, 0.05 );
var Material = new THREE.MeshBasicMaterial( { map: Texture } );

var ExtrTexture = THREE.ImageUtils.loadTexture(
                                    'textures/extrmtr.jpg' );
ExtrTexture.wrapS = ExtrTexture.wrapT = THREE.RepeatWrapping;
ExtrTexture.offset.set( 0, 0.7 );
ExtrTexture.repeat.set( 0, 0.13 );
var extrMaterial = new THREE.MeshBasicMaterial(
                                    { map: ExtrTexture } );

var materials = [ Material, extrMaterial ];
var material = new THREE.MeshFaceMaterial( materials );
```

Наконец, создаем мэш и добавляем его на сцену:

```
var mesh = new THREE.Mesh( geometry, material );
mesh.rotation.x = Math.PI/2;
scene.add( mesh );
```

Рис. 49. Кубоид с закругленными краями

Если в качестве фигуры взять эллипс и нарисовать соответствующую фигуру[11]:

```
var EllipseShape = new THREE.Shape();
var width = 200, height = 100;

( function Ellipse( ctx, x, y, w, h ){
    var kappa = 0.5522848;
    ox = (w / 2) * kappa,
    oy = (h / 2) * kappa,
    xe = x + w,
    ye = y + h,
    xm = x + w / 2,
    ym = y + h / 2;

    ctx.moveTo(x, ym);
    ctx.bezierCurveTo(x, ym - oy, xm - ox, y, xm, y);
    ctx.bezierCurveTo(xm + ox, y, xe, ym - oy, xe, ym);
    ctx.bezierCurveTo(xe, ym + oy, xm + ox, ye, xm, ye);
    ctx.bezierCurveTo(xm - ox, ye, x, ym + oy, x, ym);

} )( EllipseShape, -width/2, -height/2, width, height );
```

то получим такую поверхность (рис. 50):

[11] URL: http://jsbin.com/ovuret/2/edit

Рис. 50. Фигура, полученная вытягиванием эллипса

Конечно, важно, чтобы фигура в основании была замкнутой.

3.3.9. Поверхности вращения. Рисуем пешку и бокал

Тело вращения получается вращением кривой вокруг оси Y. Для создания поверхности вращения используется класс LatheGeometry. Конструктор класса:

```
LatheGeometry( points, segments, phiStart, phiLength );
```

Указывается множество точек points для построения вращаемой кривой, которые просто соединяются точками; количество сегментов segments радиальной окружности (по умолчанию равно 12). Также можно указать начальный угол phiStart и величину угла вращения phiLength.

Нарисуем, например, шахматную пешку. Для построения изгиба за основу возьмем функцию $10 + 20 * e^{-x^2}$. Ее график представлен на рис. 51.

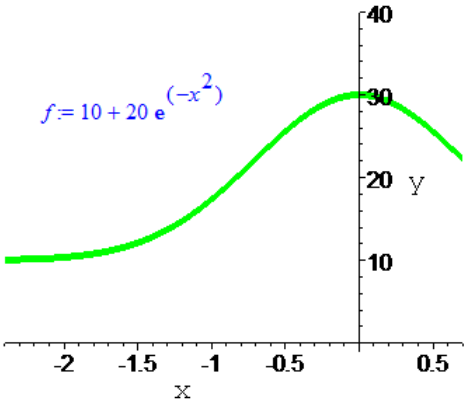

Рис. 51. График функции – «основы» пешки

Детали пешки будем собирать в трехмерном объекте pawn. Опишем также материал объекта:

```
pawn = newTHREE.Object3D;
    var material = new THREE.MeshPhongMaterial(
            { color: 0xdaa520, specular: 0x00b2fc, shininess: 50,
                    blending: THREE.NormalBlending, depthTest: true } );
```

Теперь заполняем массив вершин:

```
varpoints = [];
for ( var i = - 2.4; i < 0.7; i = i + 0.1 )
{
points.push( new THREE.Vector3( 10 + 20*Math.exp( -i*i ), 0, 24*i ) );
}
```

Добавляем в объект и поворачиваем (рис. 52):

```
var geometry = new THREE.LatheGeometry( points, 32 );
    object = new THREE.Mesh( geometry, material );
    object.position.set( 0, 26, 0 );
    object.rotation.x = Math.PI/2;
pawn.add( object );
```

Рис. 52. Основа пешки

Нарисуем голову пешки в виде сферы с тем же материалом:

```
var geometry = new THREE.SphereGeometry(16, 50, 50);
var Sphere1 = new THREE.Mesh( geometry, material );
Sphere1.position.set( 0, 102, 0 );
pawn.add( Sphere1 );
```

Добавим два диска (цилиндра) для придания большего сходства с пешкой:

```
// "подголовник" пешки
var geometry = new THREE.CylinderGeometry( 22, 22, 6, 32 );
var disc1 = new THREE.Mesh( geometry, material );
disc1.position.set( 0, 86, 0 );
pawn.add( disc1 );

// основание пешки
var geometry = new THREE.CylinderGeometry( 32, 32, 8, 32 );
var disc2 = new THREE.Mesh( geometry, material );
disc2.position.set( 0, 10, 0 );
pawn.add( disc2 );
```

И, наконец, ткань на основании пешки (опять-таки в виде диска), с темным материалом:

```
var material = new THREE.MeshPhongMaterial( { color:
0x4c3c18,
        blending: THREE.NormalBlending, depthTest: true } );

var geometry = new THREE.CylinderGeometry( 32, 32, 6, 32 );
var disc3 = new THREE.Mesh( geometry, material );
disc3.position.set( 0, 3, 0 );
pawn.add( disc3 );

scene.add( pawn );
```

Результат (рис. 53):

Рис. 53. Шахматная пешка

Теперь нарисуем бокал. Конечно, можно подобрать точки массива наугад. Но гораздо быстрее это произойдет, подобрав графики функций. Для нашего бокала можно взять за основу объединение двух функций – кривую Безье из рис. 44 и *кинк* $30 * \mathbf{arctg}(e^{3x})$ (рис. 54):

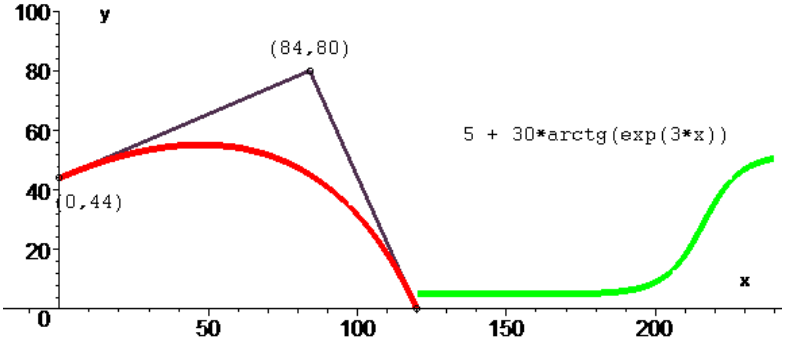

Рис. 54. Графики функций для бокала

Итак, мы должны создать массив points и забить его точками двух кривых:

```
var points = [];

v0 = new THREE.Vector2( 0, 44 );
v1 = new THREE.Vector2( 84, 80 );
v2 = new THREE.Vector2( 120, 0 );

var BezierCurve = new THREE.QuadraticBezierCurve( v0, v1, v2 );

for ( var i = 0; i < 1; i = i + 0.05 ) {points.push( new THREE.Vector3
                    ( BezierCurve.getPoint( i ).y,  0, 236 - 120*i ) ); }

for ( var i = - 4; i < 1.2; i = i + 0.1 ) {points.push( new THREE.Vector3
                    ( 5 + 30*Math.atan(Math.exp(3*i)), 0, 24 - 24*i ) );}
```

Описываем геометрию и материал, создаем бокал:

```
    var geometry = new THREE.LatheGeometry( points, 32 );

var material = new THREE.MeshPhongMaterial(
    { color: 0xd53044, specular: 0x00b2fc, shininess: 50,
        side: THREE.DoubleSide, blending: THREE.NormalBlending,
                depthTest: true } );
var glass = new THREE.Mesh( geometry, material );
```

```
    glass.position.set( 0, -100, 0 );
    glass.rotation.x = -Math.PI/2;

scene.add( glass );
```

Бокал готов (рис. 55):

Рис. 55. Бокал

3.3.10. Параметрические поверхности

Чтобы изобразить поверхности, заданные своими параметрическими уравнениями, используется класс ParametricGeometry. Синтаксис вызова:

ParametricGeometry(func, slices, stacks, useTris), где

func – функция, возвращающая координаты точки поверхности для заданных параметров u и v;
slices – уровень детализации (первой) u-координат;
stacks – уровень детализации (второй) v-координат.
Изобразим, к примеру, эллиптический параболоид

$$z = x^2 + y^2.$$

В параметрическом виде эту поверхность можно записать как

$$x = \rho \cos\theta, \qquad y = \rho \sin\theta, \qquad z = \rho^2.$$

При этом параметры функции можно обозначать как угодно (а не обязательно u и v). Кроме того, нужно учесть, что оба параметра в ParametricGeometry меняются **от нуля до единицы**.

Итак, опишем наш параболоид. Объявим глобальные переменные:

```
var segments = 32;
var graphGeometry;
var graphMesh;
```

и создадим функцию для ParametricGeometry:

```
ParamFunction = function(rho, teta)
  {
        rho = 100*rho; teta = 2*Math.PI*teta;
        x =  rho*Math.cos(teta);
        y =  rho*Math.sin(teta);
        varz = rho*rho/40;
        return new THREE.Vector3(y, z, x);
  };
```

При этом мы растянули область изменения переменной rho от нуля до 100, и teta от нуля до 2π, сжали функцию по оси Oz. Кроме того, учли расположение осей и вместо вектора (x, y, z) взяли (y, z, x), чтобы поверхность была направлена привычным нам образом.

Осталось создать геометрию поверхности, материал, фигуру («мэш»), и добавить на сцену:

```
graphGeometry = new THREE.ParametricGeometry(
 ParamFunction, segments, segments, false );

var material = new THREE.MeshNormalMaterial(
   {color: 0x33CCFF, side:THREE.DoubleSide });

graphMesh = new THREE.Mesh(
graphGeometry, material );
graphMesh.doubleSided = true;
// двусторонняя поверхность
graphMesh.position.set(0,0,0);
scene.add(graphMesh);
```

Добавим также оси координат, как было сделано выше. Результат на рисунке:

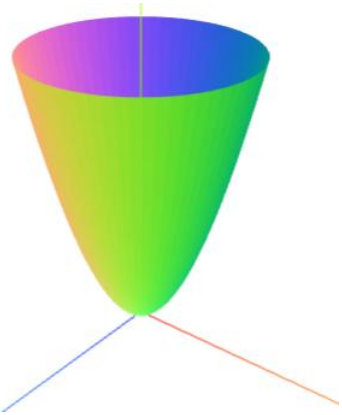

Рис. 56. Эллиптический параболоид

Для большей наглядности можно воспользоваться примером Lee Stemkoski[12]. Добавим небольшое изображение square.png в папку textures для будущей текстуры вида:

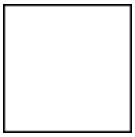

Рис. 57.Текстура square.png

Идея в том, чтобы «натянуть» эту текстуру на каждый сегмент изображения поверхности. Достаточно изменить материал следующим образом:

```
… // то же самое
graphGeometry = newTHREE.ParametricGeometry(
ParamFunction, segments, segments, false );

var Texture = new THREE.ImageUtils.loadTexture(
                                        'textures/square.png' );
Texture.wrapS = Texture.wrapT = THREE.RepeatWrapping;
```

[12]URL: http://stemkoski.github.io/Three.js/Graphulus-Surface.html

```
Texture.repeat.set( 40, 40 );

var material  = new THREE.MeshBasicMaterial( {
        map: Texture,
        vertexColors: THREE.VertexColors,
        side:THREE.DoubleSide } );
material.map.repeat.set( segments, segments );

graphMesh = new THREE.Mesh(
                              graphGeometry, material );
… // то же самое
```

Результат на следующем рисунке:

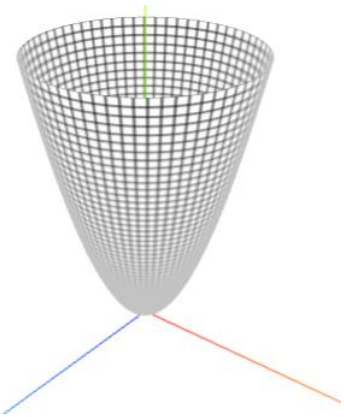

Рис. 58. Эллиптический параболоид

Также можно раскрасить каждый участок в свой цвет, при этом сделать участки одинаковой высоты одного цвета. Для этого добавим примерно такой код[13]:

```
graphGeometry.computeBoundingBox();
varyMin = graphGeometry.boundingBox.min.y;
var yMax = graphGeometry.boundingBox.max.y;
var yRange = yMax - yMin;
var color, point, face, numberOfSides, vertexIndex;

var faceIndices = [ 'a', 'b', 'c', 'd' ];
```

[13]URL: http://stemkoski.github.io/Three.js/Graphulus-Surface.html

```
for ( var i = 0; i < graphGeometry.vertices.length; i++ )
  {
    point = graphGeometry.vertices[ i ];
    color = new THREE.Color( 0x0000ff );
    color.setHSL( 0.7 * (yMax - point.y) / yRange, 1, 0.5 );
    graphGeometry.colors[i] = color;
  }

for ( var i = 0; i < graphGeometry.faces.length; i++ )
  {
    face = graphGeometry.faces[ i ];
    numberOfSides = ( face instanceof THREE.Face3 ) ? 3 : 4;

   for( var j = 0; j < numberOfSides; j++ )
            {
            vertexIndex = face[ faceIndices[ j ] ];
                face.vertexColors[ j ] =
                 graphGeometry.colors[ vertexIndex ];
            }
  }
```

Результат на следующем рисунке:

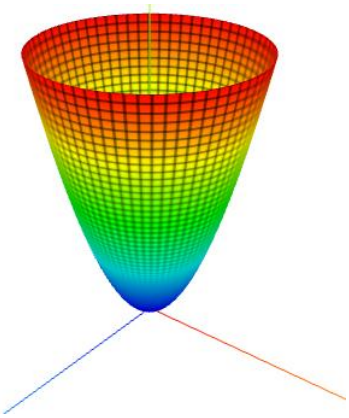

Рис. 59. Эллиптический параболоид

3.4. Таймер Clock

Таймер в Three.js позволяет отслеживать время, пройденное с момента того или иного события, и имеет разнообразное применение. Для добавления таймера напишем:

```
varclock = newTHREE.Clock( );
```

Класс имеет следующие методы:
– start() – запуск таймера;
– stop() – остановка таймера;
– getElapsedTime() – возвращает количество секунд elapsedTime (тип Float), прошедших с момента запуска таймера;
– getDelta() – возвращает количество миллисекунд (тип Float), прошедших с момента предыдущего обращения к таймеру.

Класс имеет следующие свойства:
– elapsedTime – количество секунд (тип Float), прошедших с момента запуска таймера. Вычисляется в момент обращения к таймеру;
– running – отслеживает, работает таймер или нет.

3.4.1. Периодическое изменение цвета материала

Рассмотрим пример с кубом, который меняет свой цвет случайным образом с заданным периодом. Объявим глобальные переменные:

```
varCube, material;
varclock = newTHREE.Clock( ); varperiod = 2;
```

Теперь перейдем к нашей функции инициализации init(). Добавим кубоид:

```
var geometry = new THREE.BoxGeometry( 200, 100, 150 );
material = new THREE.MeshPhongMaterial( { color: 0x00ff00 } );

Cube = new THREE.Mesh( geometry, material );
scene.add( Cube );
```

Запускаем таймер:

```
clock.start();
```

Теперь внутри render() добавим:

```
if ( clock.running )
    {
        var time = clock.getElapsedTime();
        if ( time > period )
            {
                material.color.setHex( Math.random() * 0xffffff );
                clock.elapsedTime = 0;// обнуляем счетчик таймера
            }
    }
```

Теперь цвет куба периодически меняется.

3.4.2. Ограничиваем частоту обработки событий клавиатуры (на некоторые клавиши)

Теперь вернемся к теме обработки событий клавиатуры. Модифицируем пример с движением куба из п. 2.1.1. В нем наш куб двигался влево-вправо при нажатии стрелок и поднимался при нажатии на клавишу ENTER. Ограничим возможность реагирования кубика, например, на ENTER так, что кубик реагирует на него не чаще одного раза в секунду. Создадим класс таймера и период ожидания:

```
varclock = newTHREE.Clock( );
varperiod = 1;
```

Модифицируем наш класс Key следующим образом:

```
varKey =
{
_pressed: {},

   VK_LEFT:      37,
   VK_RIGHT:     39,
   VK_ENTER:     13,

  isDown: function( keyCode )
    {
        var flag = this._pressed[keyCode];
        if ( keyCode == 13 ) {  delete this._pressed[ keyCode ]; }
        return flag;
    },
 onKeydown: function(event)
```

```
    {

    if ( event.keyCode == 13 )
        {
            var time = clock.getElapsedTime();
            if ( time >= period )
                {
                    this._pressed[event.keyCode] = true;
                    clock.elapsedTime = 0; //обнуляем счетчик
                                           таймера
                }
        }
    else
        {
            this._pressed[event.keyCode] = true;
        }

    },

  onKeyup: function(event)
    {
        delete this._pressed[event.keyCode];
    }
};
```

Теперь наш кубик будет реагировать на нажатие клавиши ENTER, только если прошло не менее секунды (у нас period = 1) с последнего реагирования. Это достигается проверкой условия if (time>= period). На остальные клавиши куб реагирует без изменений.

Естественно, нужно не забыть запустить таймер после создания сцены и объектов:

```
clock.start;  clock.elapsedTime = period;
```

Здесь мы также учли, что кубик сразу может реагировать на клавишу ENTER с момента загрузки страницы.

4. НЕМНОГО ПРАКТИКИ

4.1. Визуализация химических молекул

Представляет отдельный интерес задача изображения трехмерных моделей молекул. Молекулы имеют весьма сложный вид, подобрать координаты и нарисовать их довольно тяжело. К счастью, эта тема достаточно проработана. Имеется множество программ для изображения трехмерных формул. Например, программа Tinker хранит координаты молекул в файлах с расширением .xyz.

Файл формата .xyz содержит одну строку с целым числом, равным количеству атомов в молекулярной системе (N), и N строк, с разделёнными пробелами или табуляцией именами атомов и их координатами (в Ангстремах). Например, файл ethanol.xyz для координат атомов одной молекулы этанола выглядит следующим образом:

```
9  Ethanol
1  C  -0.231579  -0.350841  -0.037475  1  2  4  5  6
2  C   0.229441   0.373160   1.224850  1  1  3  7  8
3  O   0.868228  -0.551628   2.114423  6  2  9
4  H   0.619613  -0.833754  -0.565710  5  1
5  H  -0.709445   0.352087  -0.754607  5  1
6  H  -0.976393  -1.144198   0.191635  5  1
7  H  -0.628785   0.860022   1.736350  5  2
8  H   0.952253   1.174538   0.962081  5  2
9  H   0.204846  -1.119563   2.483509  21 3
```

Здесь также содержатся данные о связи атома с другими атомами (седьмая и дальнейшие цифры). Например, первый атом нужно соединить со 2, 4, 5 и 6 атомами. За что отвечает шестая цифра, автору неизвестно.

Введем глобальные переменные для молекулы, чтобы потом можно было заставить ее медленно вращаться:

```
var molecule, phi = 0;
```

Создадим молекулу как трехмерный объект, в который потом будем добавлять атомы (размерами пропорционально k):

```
molecule = new THREE.Object3D(); k = 0.4;
```

Теперь объявим массивы для атомов (обычные шарики), их материалов и геометрий. Длины последних двух массивов равны количеству разных атомов (для этанола – 3):

```
var atoms = [ ], materials = [ ], geometries = [ ];
```

Удобно массив atoms сделать ассоциативным. Содержать он будет номер атома, цвет атома и размер в ангстремах. При этом сами условно считаем, например, атом водорода H первым:

```
atoms[ 'H' ] = [ 0, 0x2a52be, 0.53 ];
atoms[ 'O' ] = [ 1, 0xff0000, 0.60 ];
atoms[ 'C' ] = [ 2, 0x00ff12, 0.91 ];
```

Объявляем циклом материалы и геометрии этих молекул:

```
for ( var Name in atoms )
  {
      var material = new THREE.MeshPhongMaterial(
      { color: atoms[Name][1], specular: 0x00b2fc, shininess: 50,
      blending: THREE.NormalBlending, depthTest: true } );
      materials.push( material );
      var geometry = new THREE.SphereGeometry(
                                        atoms[Name][2]*k, 64, 64);
      geometries.push( geometry );
  }
```

Информацию о координатах молекулы из файла ethanol.xyz будем хранить в массиве info следующим образом:

```
var info = [ ];
info.push( '  1  C   -0.231579   -0.350841  -0.037475   1   2   4   5   6 ' ) ;
info.push( '  2  C  0.229441  0.373160  1.224850    1   1   3   7    8 ' ) ;
info.push( '  3  O    0.868228 …
```

Конечно, работать напрямую с такими «бесструктурными» данными неудобно. Превратим это в двумерный массив данных arr с помощью регулярного выражения:

```
var arr = [ ];
for (var i=0; i< info.length; i++)
        { arr[i] = info[i].match(/\S+/g); }
```

Тогда arr[0] = [1, 'C', -0.231579, -0.350841, -0.037475, 1, 2, 4, 5, 6], и т.д. Теперь мы знаем, что, например, 3, 4 и 5 элементы такого массива содержат координаты молекулы. Теперь легко вывести все атомы:

```
for (var i=0; i< arr.length; i++)
{
        var Name = arr[i][1]; // номер элемента
        var Punct = new THREE.Mesh(
            geometries[ atoms[Name][0] ], materials[ atoms[Name][0] ] );
        Punct.position.set(arr[i][2], arr[i][3], arr[i][4]);
        molecule.add( Punct );
}
```

Осталось вывести связи. К сожалению, в Three.js нет возможности управлять толщиной обычных линий. Воспользуемся решением Lee Stemkoski[14] и возьмем его процедуру:

```
function cylinderMesh( pointX, pointY )
{
    var direction = new THREE.Vector3().subVectors( pointY, pointX );
    var arrow = new THREE.ArrowHelper(
                direction.clone().normalize(), pointX, direction.length() );
    var edgeGeometry =
      new THREE.CylinderGeometry( 0.1, 0.1, direction.length(), 36, 4 );
    var edgeMesh = new THREE.Mesh( edgeGeometry,
                  new THREE.MeshBasicMaterial( { color: 0x0000ff } ) );
    edgeMesh.position = new THREE.Vector3().addVectors( pointX,
                                        direction.multiplyScalar(0.5) );
    edgeMesh.setRotationFromEuler( arrow.rotation );
    returnedgeMesh;
}
```

Эта процедура позволяет соединить указанные точки pointX, pointY цилиндром. С ее помощью мы и будем изображать связи между атомами. При этом мы будем соединять половину расстояния между точками. Из школы мы знаем, что середина между точками (x_1, y_1, z_1) и (x_2, y_2, z_2) находится как полусумма соответствующих координат:

$$x = \frac{x_1 + x_2}{2}, \qquad y = \frac{y_1 + y_2}{2}, \qquad z = \frac{z_1 + z_2}{2}.$$

[14]URL: http://stemkoski.github.io/Three.js/LeapMotion.html

Пробежимся по массиву arr. Соединяем атом с номером в начале массива (arr[i][0] минус единица) с атомами, номера которых стоят на 6, 7 и т.д. местах:

```
for (i = 0; i < arr.length; i++)
  {
            var num = arr[i][0] - 1;   // нумератома
            var x1 =  parseFloat( arr[num][2] );
            var y1 =  parseFloat( arr[num][3] );
            var z1 =  parseFloat( arr[num][4] );

  for (j = 6; j < arr[i].length; j++)
            {

            var num = arr[i][j] - 1;   // нумератома

            var x2 = ( parseFloat( arr[num][2] ) + x1 ) / 2;
            var y2 = ( parseFloat( arr[num][3] ) + y1 ) / 2;
            var z2 = ( parseFloat( arr[num][4] ) + z1 ) / 2;

            var fingerLength = cylinderMesh(
                        new THREE.Vector3(x1, y1, z1),
                        new THREE.Vector3(x2, y2, z2) );
            fingerLength.material = materials[ atoms[ arr[i][1] ][0] ];
            molecule.add( fingerLength );

            }
  }
```

Наконец, добавляем молекулу на сцену:

```
scene.add( molecule );
```

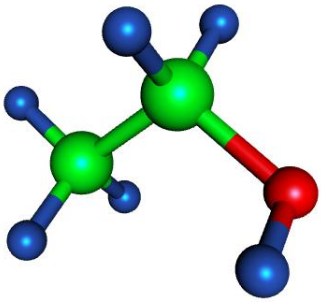

Рис. 60. Молекула этанола

Еще можно добавить внутри render() вращение молекулы:

```
function render()
{
    molecule.rotation.y = molecule.rotation.y + 0.007;
    light.position = camera.position;
    controls.update();
    renderer.render(scene, camera);
}
```

Здесь мы также добиваемся того, чтобы свет light всегда освещал молекулу с нашей стороны (со стороны камеры).

При разработке мы активно использовали массивы, чтобы легко можно было нарисовать другую молекулу. Например, взяв

```
atoms[ 'H' ] = [ 0, 0x2a52be, 0.53 ];
atoms[ 'O' ] = [ 1, 0xff0000, 0.60 ];
atoms[ 'C' ] = [ 2, 0x00ff12, 0.91 ];
atoms[ 'N' ] = [ 3, 0xff00ff, 0.92 ];
```

и (остальной код не трогаем)

```
var info = [ ];
info.push( '  1  C    -0.763510    0.487880    0.000000    3 2    3    5  ' ) ;
info.push( '  2  O     -1.378580   -0.561740    0.000000    7    1  ' ) ;
info.push( '  3  N      0.604080 …
```

сразу получаем изображение молекулы N-Метилформамида:

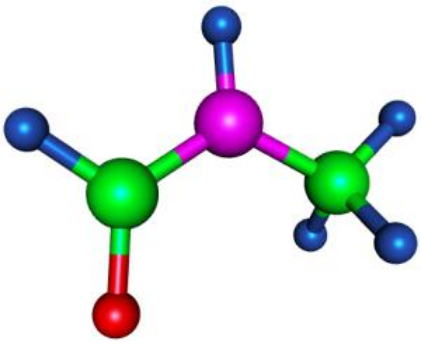

Рис. 61. Молекула N-Метилформамида

4.2. Рисование школьных примеров из стереометрии

Эта задача представляет отдельный интерес, поскольку здесь мы посмотрим пример создания текстовых спрайтов, которые всегда «смотрят» на наблюдателя. Здесь мы во многом будем опираться на пример Stemkoski[15].

Рассмотрим простенькую задачу из стереометрии: *Дано $ABCDA_1B_1C_1D_1$ – прямоугольный параллелепипед. BCC_1B_1 – квадрат. $AB = 12$, $BD_1 = 20$. Найти сторону BC.*

Изобразим параллелепипед в виде простых линий, соединяющих заданные точки. Конечно, пришлось сначала нарисовать основание на бумаге. Понадобятся два массива – для названий вершин и их координат:

```
var point_names =[ 'A', 'B', 'C', 'D', 'A1', 'B1', 'C1', 'D1' ];
varGeomArr = [ ]; k = 20;
```

Размещаем вершины (удобно это сделать с коэффициентами пропорциональности):

```
GeomArr.push( new THREE.Vector3( -7, -5, 5 ).multiplyScalar( k ));
GeomArr.push( new THREE.Vector3( 7, -5, 5).multiplyScalar( k ));
GeomArr.push( new THREE.Vector3( 7, -5, -5 ).multiplyScalar( k ));
GeomArr.push( new THREE.Vector3( -7, -5, -5 ).multiplyScalar( k ));
GeomArr.push( new THREE.Vector3( -7, 5, 5 ).multiplyScalar( k ));
GeomArr.push( new THREE.Vector3( 7,5, 5 ).multiplyScalar( k ));
GeomArr.push( new THREE.Vector3( 7, 5, -5 ).multiplyScalar( k ));
GeomArr.push( new THREE.Vector3( -7, 5, -5 ).multiplyScalar( k ));
```

С помощью метода multiplyScalar мы растягиваем вектора в k раз.

Для рисования спрайтов в Three.js предусмотрены специальные класс Sprite и материал SpriteMaterial. Удобно создать отдельную функцию createTextSprite для вывода текстовых спрайтов с заданными текстом, стилем шрифта и его цветом:

```
function createTextSprite(text, font, color)
{
  var canvas = document.createElement('canvas');
  var context = canvas.getContext('2d');
  context.font = font;
```

[15]URL: http://stemkoski.github.io/Three.js/Labeled-Geometry.html

```
    context.fillStyle   = color;
    context.lineWidth = 4;

    context.fillText( text, 60, 120);

    var texture = new THREE.Texture(canvas);
    texture.needsUpdate = true;

    var spriteMaterial = new THREE.SpriteMaterial({ map: texture });

    var sprite = new THREE.Sprite( spriteMaterial );
    sprite.scale.set( 40, 20, 1.0);
    return sprite;

  }
```

Теперь в цикле изобразим вершины (как маленькие шарики) и добавим их названия, которые будем располагать вблизи вершин:

```
var geometry = new THREE.SphereGeometry( 0.12*k, 24, 24);
var material = new THREE.MeshBasicMaterial( { color: 0x000000 } );

  for (var i=0; i<GeomArr.length; i++)
    {
      var Punct = new THREE.Mesh( geometry, material );
      Punct.position = GeomArr[i];
      scene.add( Punct );
      var TextSprite = createTextSprite(
    point_names[i], 'Bold 128px Arial', "#00F");
      var coor = new THREE.Vector3(
    GeomArr[i].x, GeomArr[i].y , GeomArr[i].z );
      coor.multiplyScalar( 1.1 );
      TextSprite.position = coor;
      scene.add( TextSprite );
    }
```

Далее найдем середину AB и там выставим длину AB – 12 (рис. 62):

```
  var x = ( GeomArr[0].x + GeomArr[1].x ) / 2;
  var y = ( GeomArr[0].y + GeomArr[1].y ) / 2;
  var z = ( GeomArr[0].z + GeomArr[1].z ) / 2;
```

```
var TextSprite = createTextSprite(
                        '12', 'Bold 128px Arial', '35B539');
TextSprite.position.set( 1.1*x, 1.1*y, 1.1*z );
scene.add( TextSprite );
```

Аналогично добавим размер диагонали BD_1 и знак вопроса возле стороны BC. Осталось соединить прямыми линиями нужные вершины. Создадим материал и геометрию линий:

```
var Material = new THREE.LineBasicMaterial({ color: 0x35B539 });
var lineGeometry = new THREE.Geometry();
```

Добавляем вершины в геометрию:

```
lineGeometry.vertices.push( GeomArr[0], GeomArr[1],
                    GeomArr[2], GeomArr[3], GeomArr[0] );
lineGeometry.vertices.push( GeomArr[4], GeomArr[5],
                    GeomArr[6], GeomArr[7], GeomArr[4] );
```

и соединяем линиями:

```
var line = new THREE.Line( lineGeometry, Material );
scene.add( line );
```

Пока мы соединили вершины верхнего и нижнего оснований. Осталось соединить их между собой. Соединим, например, вершины B и B_1. Их номера в массиве GeomArr первый и пятый:

```
var lineGeometry = new THREE.Geometry();
lineGeometry.vertices.push( GeomArr[1], GeomArr[5] );
var line = new THREE.Line( lineGeometry, Material );
scene.add(line);
```

Аналогично соединяем оставшиеся вершины и рисуем диагональ. Результат (рис. 62):

Дано $ABCDA_1B_1C_1D_1$ - прямоугольный параллелепипед. BCC_1B_1 - квадрат. $AB = 12$, $BD_1 = 20$. Найти сторону BC.
Ответ: $BC = 8\sqrt{2}$

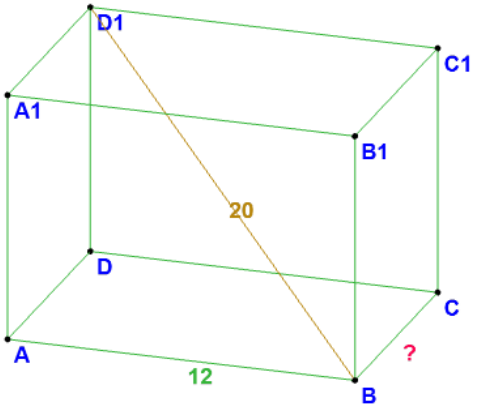

Рис. 62. Модель к школьной задаче

5. ИМПОРТ МОДЕЛЕЙ ИЗ ГРАФИЧЕСКИХ РЕДАКТОРОВ

5.1. Импорт модели из Blender

Для использования готовых трехмерных моделей Blender в ваших WebGL-приложениях нужно сначала экспортировать эту модель в формат JSON с помощью специального аддона. Скачайте его с сайта threejs.org и подключите к Blender следующим образом[16]. После загрузки аддона сначала выбираете подходящую версию для вашей версии Blender (например, папка 2.66). Копируете оттуда папку io_mesh_threejs в директорию, где установлен Blender, в папку addons (у меня это, например, C:\ProgramFiles\BlenderFoundation\Blender\2.69\scripts\addons).

Теперь нужно активировать плагин. Открываем настройки Blender: File – User Preferences…, выберем вкладку плагинов Addons, Import-Export, найдем three.js format и активируем плагин, установив напротив галочку.

Теперь можно в блендере экспортировать модель в формат JSON. В меню экспорта File – Export теперь доступна вкладка Three.js (.js). Установим настройки согласно рис. 63.

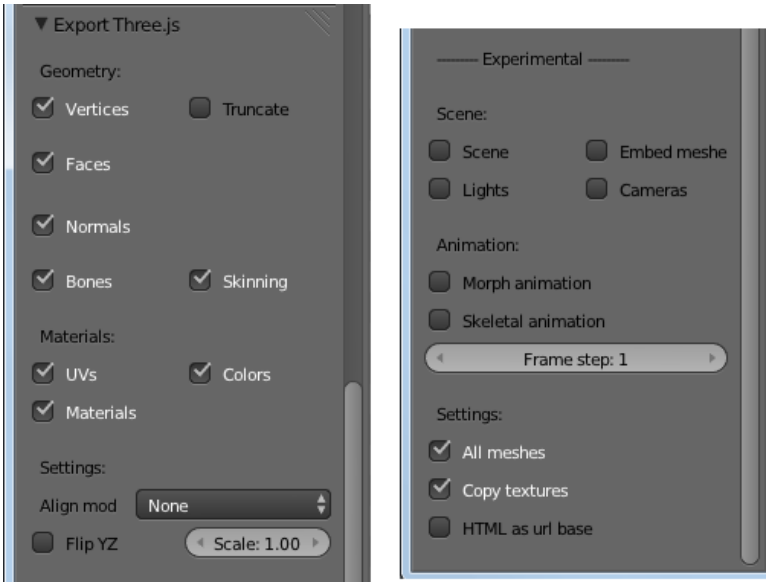

Рис. 63. Настройки экспорта в Blender

[16]URL: https://github.com/mrdoob/three.js/tree/master/utils/exporters/blender

Теперь можно экспортировать модель, нажав на кнопочку Export Three.js в правом верхнем углу. Полученный файл penguin.js (обычно сохраняется в той же папке, что и сама модель) будет иметь расширение js. Создадим в директории нашего проекта папку models и закинем ее туда.

Теперь загрузить нашу модель в браузер не составляет проблем. Воспользуемся как образцом примером Stemkoski[17] и создадим вспомогательную функцию

```
function addModelToScene( geometry, materials )
 {
    var material = new THREE.MeshFaceMaterial( materials );
    var model = new THREE.Mesh( geometry, material );
    model.scale.set( 3, 3, 3);
    scene.add( model );
 }
```

Теперь внутри init() загружаем модель с помощью всего лишь двух строчек кода:

```
var jsonLoader = new THREE.JSONLoader();
jsonLoader.load( "models/penguin.js", addModelToScene );
```

Готово! Теперь нашу модель можно посмотреть в браузере (рис. 64):

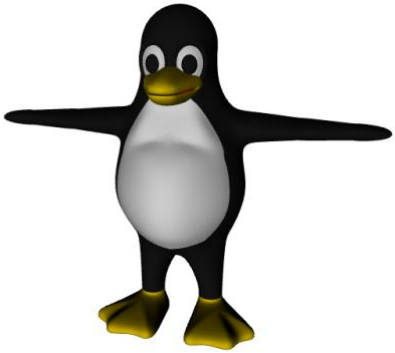

Рис. 64. Модель, экспортированная из Blender,в браузере

[17]URL: http://stemkoski.github.io/Three.js/Model.html

5.2. Импорт модели из 3D Max

Для использования модели, созданной в Autodesk 3ds Max, лучше сначала перевести ее в один из форматов .dae, .obj / .mtl, .vtk, .ply, и т.д. Затем эту модель можно загрузить на страницу с помощью подходящего лоадера Three.js.

Рассмотрим пример загрузки готовой модели с предварительным переводом в форматы .obj / .mtl. Файл формата .obj содержит данные о геометрии модели, файл .mtl – информацию о материале и текстурах. Запустите Autodesk 3ds Max и откройте готовую модель.

Для экспорта модели потребуется нажать на кнопку Главное меню → Export. Появляется меню сохранения файла, где выбираем из списка формат .OBJ. Выбираем название для модели (например, model.obj) и нажимаем кнопку Save. Откроется меню с настройками экспорта (рис. 65). В левой части раздела настроек Geometry нужно установить галочки на:

- Flip YZ-axis (Poser-like);
- Shapes/Lines;
- Hidden objects.

Далее, Faces установим на Triangles («треугольники») и поставим галочки на:

- Texture coordinates;
- Normals;
- Smoothing groups.

В правой части окна Material ставим галочку на:

- Export materials;
- Create mat-library.

Остальные галочки убираем. Далее открываем меню Map-Export … и убираем все галочки.

Нажимаем на экспорт (рис. 65). В соответствующей папке (у меня это Мои документы\3dsMax\export) появятся файлы model.obj и model.mtl. Создадим в директории с нашим проектом папку models и скопируем туда наши файлы.

Для возможности загрузки моделей нужно скачать с сайта threejs.org и подключить к проекту библиотеки MTLLoader.js[18] и OBJMTLLoader.js[19]:

```
<script src="js/loaders/MTLLoader.js"></script>
<script src="js/loaders/OBJMTLLoader.js"></script>
```

[18]URL: http://threejs.org/examples/js/loaders/MTLLoader.js
[19]URL: http://threejs.org/examples/js/loaders/OBJMTLLoader.js

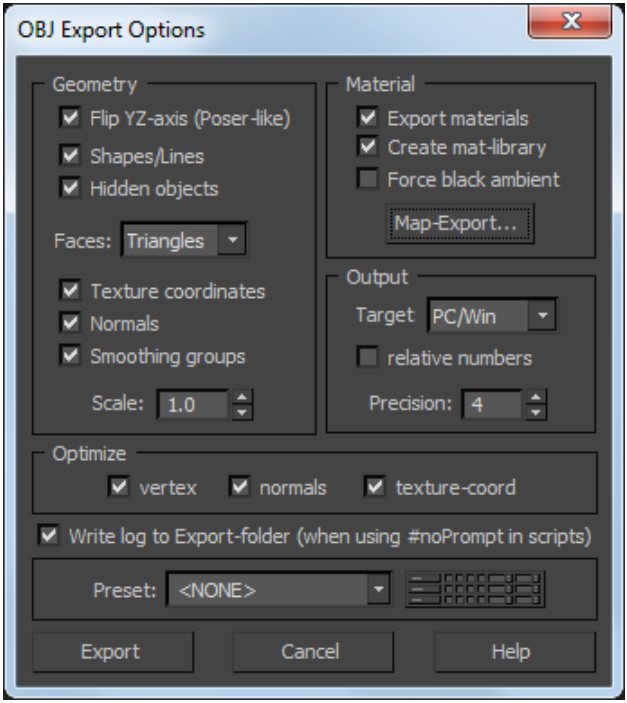

Рис. 65. Настройки экспорта модели в формат. obj / .mtl

Сам же код отображения модели будет очень простым:

```
var loader = new THREE.OBJMTLLoader();// лоадер

loader.load( 'models/model.obj', 'models/model.mtl',
      function ( GingerbreadMan )
      {
          GingerbreadMan.position.set( 0, 0, 0 );
          scene.add( GingerbreadMan );
      }
);
```

Теперь нашу модель можно посмотреть в браузере (рис. 66):

Рис. 66. Модель .obj / .mtl в браузере

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Three.js – JavaScript 3D library [Электронный ресурс] / Mr.doob. – Электрон. текстовые дан. – Режим доступа: http://threejs.org, свободный. – Загл. с экрана.

2. Stemkoski, Lee. Three.js – examples [Электронный ресурс] / Lee Stemkoski. – Электрон. текстовые дан. – Режим доступа: http://stemkoski.github.io/Three.js, свободный. – Загл. с экрана.

3. Фреймворки WebGL - Three.js [Электронный ресурс] / T.M. SoftStudio. – Электрон. текстовые дан. – Режим доступа: http://ru.tmsoftstudio.com/file/page/webgl-frameworks/three-api-ru/three.html, свободный. – Загл. с экрана.

4. Начало работы с WebGL (Windows) [Электронный ресурс] / Microsoft. – Электрон. текстовые дан. – Режим доступа: http://msdn.microsoft.com/ru-ru/Library/dn385807(v=vs.85).aspx, свободный. – Загл. с экрана.

5. Кантор, И. Центральный Javascript-ресурс. Учебник с примерами скриптов. Форум. Книги и многое другое [Электронный ресурс] / И. Кантор. – Электрон. текстовые дан. – Режим доступа: http://javascript.ru, свободный. – Загл. с экрана.

Листинг mainapp.js

```
// глобальные переменные
var container, camera, controls, scene, renderer, light;
varCube;

// начинаем рисовать после полной загрузки страницы
window.onload =
        function()
            {
                init();
                animate();
            }

function init()
{

    scene = new THREE.Scene();  //создаем сцену
    AddCamera( 0, 300, 500);  // добавляем камеру
    AddLight( 0, 0, 500 );  //устанавливаем белый свет

//создаем рендерер
    renderer = new THREE.WebGLRenderer( { antialias: true } );
    renderer.setClearColor( 0xffffff );
    renderer.setSize( window.innerWidth, window.innerHeight );
    container = document.getElementById('MyWebGLApp');
    container.appendChild( renderer.domElement );

//добавляем куб
    var geometry = new THREE.BoxGeometry( 200, 100, 150);
    var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
    Cube = new THREE.Mesh( geometry, material );
    Cube.position.z = -100;
    Cube.rotation.z = Math.PI / 6;
    scene.add( Cube );
}

function animate()
    {
        requestAnimationFrame(animate);
        render();
    }
```

```
function render()
{
    Cube.position.x = Cube.position.x + 1; //куб движется
    Cube.rotation.y = Cube.rotation.y + 0.01; //и вращается вокруг оси
    controls.update();
    renderer.render(scene, camera);
}


function AddCamera(X,Y,Z)
{

    camera = new THREE.PerspectiveCamera( 45, window.innerWidth /
    window.innerHeight, 1, 10000 );
    camera.position.set(X,Y,Z);

    controls = new THREE.TrackballControls( camera, container );

    controls.rotateSpeed = 2;
    controls.noZoom = false;
    controls.zoomSpeed = 1.2;
    controls.staticMoving = true;

}

function AddLight(X,Y,Z)
{
    light = new THREE.DirectionalLight( 0xffffff );
    light.position.set(X,Y,Z);
    scene.add( light );
}
```

Учебное издание

ВИЛЬДАНОВ Алмаз Нафкатович

3D-моделирование на WebGL с помощью библиотеки Three.js

Учебное пособие

Корректор О.В. Бутусова
Технический редактор З.Ф. Талипова

Лицензия на издательскую деятельность
ЛР № 021319 от 05.01.1999 г.

Подписано в печать 19.05.2014 г. Формат 60x84/16
Усл. печ. л. 7,5. Уч.-изд. л. 6,97. Тираж 100 экз.
Изд. № 127. Заказ 13.

Редакционно-издательский центр
Башкирского государственного университета
450076, РБ, г. Уфа, ул. Заки Валиди, 32.

Отпечатано в редакционно-издательском отделе
Нефтекамского филиала БашГУ.
452683, РБ, г. Нефтекамск, ул. Трактовая, 1.
Телефон (34783) 2-35-80.
Факс (34783) 2-35-80, 5-40-46.
E-mail: info@nfbgu.ru