Программирование графических приложений

Тема 18
Разработка игровой анимированой сцены 2

Цель изучения темы. Изучение методов создания анимированной 3D-сцены в WebGL на JavaScript с использованием библиотеки Three.js.

В качестве примера будет рассмотрено построение анимированной сцены в виде летящего над морем самолета с управлением и игровыми функциями.

На рис. 18.1 представлен начальный вариант сцены с упрощенным избражением объектов и простым управлением.

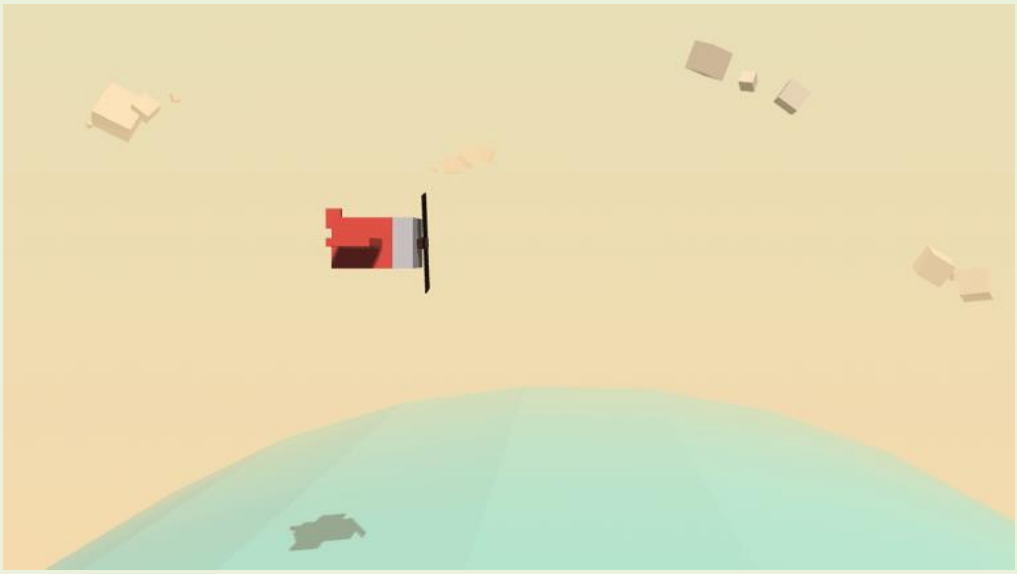

Рис. 18.1

18.1. Подготовка

HTML и CSS

Первое, что нужно сделать - добавить между открывающим и закрывающим тегом body блок, в котором будет выполняться рендеринг сцены:

```
<div id="world" class="world"></div>
```

Оформим блок так, чтобы он занимал все окно браузера. Фон страницы сделаем ввиде градиента, который заменит собой небо.

```
.world {
   position: absolute;
   width:100%;
   height:100%;
   overflow: hidden;
   background:linear-gradient(#e4e0ba, #f7d9aa);
}
```

Теперь необходимо подключить библиотеку Three.js к веб-странице. Для этого сразу перед закрывающим тегом body напишем скрипт:

```
<script src="js/three.js"></script>
```

Далее переходим к JavaScript.

Цветовая палитра

Для начала определимся с цветами в будущей сцене. Выбранная цветовая гамма представлена на рис. 18.2.

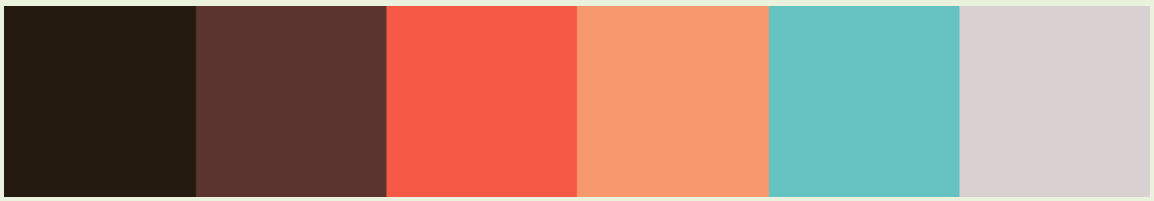

Рис. 18.2

Эти цвета будут постоянно использоваться в игре, поэтому вынесем HEX-значения этих цветов в объект *Colors*:

```
var Colors = {
    red: 0xf25346,
    white: 0xd8d0d1,
    brown: 0x59332e,
    pink: 0xF5986E,
    brownDark: 0x23190f,
    blue: 0x68c3c0,
};
```

Структура кода

Несмотря на то, что предстоит написать большое количество JavaScript-кода, его структура довольно проста. Все основные функции необходимо запустить при инициализации объекта *window* с помощью функции init():

```
window.addEventListener('load', init, false);

function init() {
    // настройка сцены, камеры и рендера
    createScene();

    // добавление освещения сцены
    createLights();

    // добавление объектов на сцену
    createPlane();
    createSea();
    createSky();

    // запускаем цикл для обновления объектов
    // на сцене и покадровый перерасчет сцены
    loop();
}
```

18.2. Формирование сцены

Используем библиотеку Three.js

Для создания нашей 3D игры на Three.js нам понадобятся:

1. Сцена - некоторая область, в которую мы поместим 3D-объекты для последующего рендера.

2. Камера: будем использовать камеру с перспективой.

3. Рендеринг, который будет отображать сцену с ипользованием WebGL.

4. Объекты для рендера: самолет, море и облака.

5. Источники света: будем использовать полушарный (градиентный) свет для атмосферы и направленный свет для теней.

Набор объектов сцены представлен на рис. 18.3.

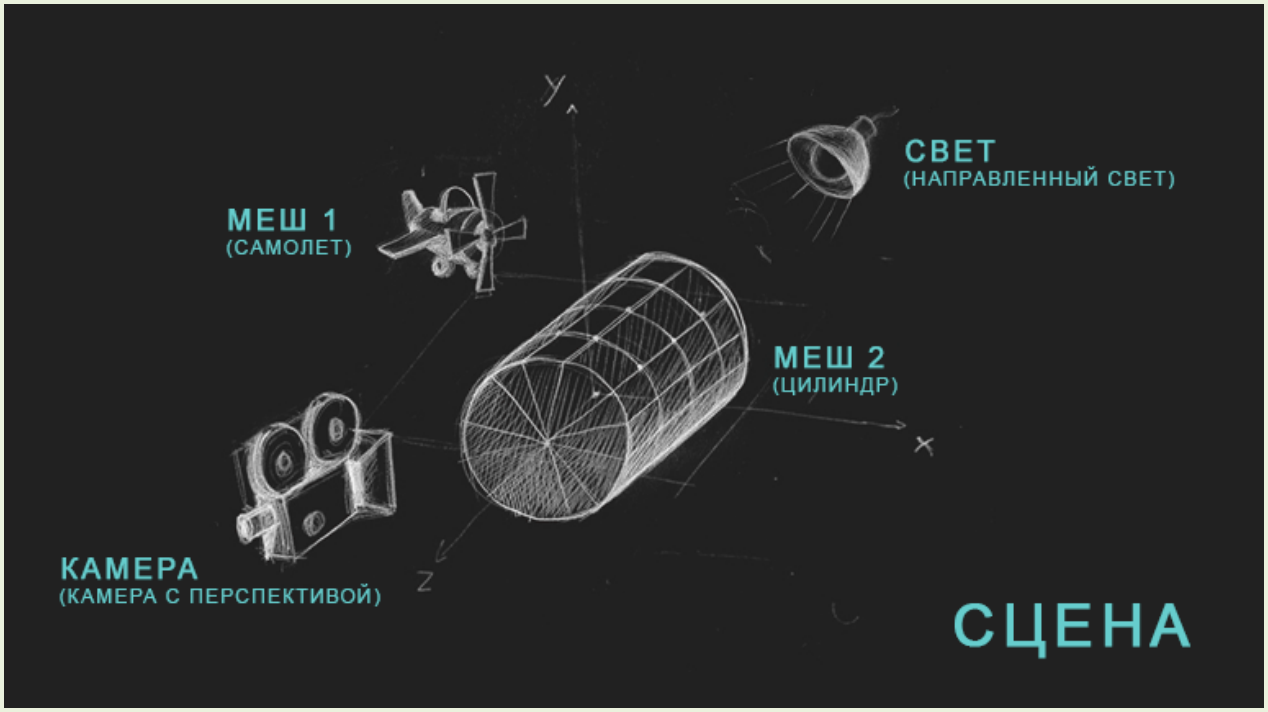

Рис. 18.3

Функция createScene()

Сцену, камеру и рендеринг запустим в функции createScene():

```
var scene, camera, fieldOfView, aspectRatio, nearPlane,
    farPlane, HEIGHT, WIDTH, renderer, container;

function createScene() {
// Получаем ширину и высоту сцены и используем
// их для установки размера и пропорций камеры,
// а также для размера финального рендера
HEIGHT = window.innerHeight;
WIDTH = window.innerWidth;

// Создание сцены
scene = new THREE.Scene();

// Добавим эффект тумана на сцену (его
// цвет возьмем из таблиц стилей, а не
// из объекта Colors
scene.fog = new THREE.Fog(0xf7d9aa, 100, 950);

// Создание камеры
```

```
aspectRatio = WIDTH / HEIGHT;
fieldOfView = 60;
nearPlane = 1;
farPlane = 10000;
camera = new THREE.PerspectiveCamera(
    fieldOfView,
    aspectRatio,
    nearPlane,
    farPlane
);

// Задаем позицию камеры в 3D пространстве
camera.position.x = 0;
camera.position.z = 200;
camera.position.y = 100;

// Создаем рендер
renderer = new THREE.WebGLRenderer({
// Разрешаем прозрачноть для сцены чтобы
// увидеть градиентное небо
alpha: true,

// Активируем сглаживание. Это может снизить
// производительность. К счастью, проект
// состоит из низкополигональных моделей
antialias: true
});

// Задаем размер рендера (в нашем случае
// он равен размеру сцены)
renderer.setSize(WIDTH, HEIGHT);

// Включаем рендер теней
renderer.shadowMap.enabled = true;

// Обратимся к DOM-элементу с идентификатором world,
// для рендера в него сцены.
container = document.getElementById('world');
container.appendChild(renderer.domElement);

// Отслеживаем изменения размера экрана.
// В случае изменения запускаем функцию
// handleWindowResize(), чтобы обновить камеру и рендер.
window.addEventListener('resize', handleWindowResize, false);
}
```

Так как размер окна браузера может меняться пользователем, необходимо обновлять сцену, рендер и камеру при изменении размера окна:

```
function handleWindowResize() {
  // обновим высоту, ширину камеры и рендера
  HEIGHT = window.innerHeight;
```

```
  WIDTH = window.innerWidth;
  renderer.setSize(WIDTH, HEIGHT);
  camera.aspect = WIDTH / HEIGHT;
  camera.updateProjectionMatrix();
}
```

Освещение сцены

Освещение является одним из самых важных аспектов формирования визуального восприятия сцены. Для начала добьемся того, чтобы объекты было видно на сцене.

```
var hemisphereLight, shadowLight;

function createLights() {
  // Полушарный свет - это градиентный свет
  // Первый параметр - цвет неба, второй - цвет земли,
  // а третий - интенсивность света
  hemisphereLight = new THREE.HemisphereLight(0xaaaaaa,0x000000, .9)

  // Направленный свет имеет определенное направление
  // Это значит, что продуцируемые лучи параллельны.
  shadowLight = new THREE.DirectionalLight(0xffffff, .9);

  // Устанавливаем направление света
  shadowLight.position.set(150, 350, 350);

  // Разрешаем отбрасывание теней
  shadowLight.castShadow = true;

  // Определяем видимую область теней
  shadowLight.shadow.camera.left = -400;
  shadowLight.shadow.camera.right = 400;
  shadowLight.shadow.camera.top = 400;
  shadowLight.shadow.camera.bottom = -400;
  shadowLight.shadow.camera.near = 1;
  shadowLight.shadow.camera.far = 1000;

  // Задаем разрешение теней.
  // Чем оно больше, тем ниже производительность.
  shadowLight.shadow.mapSize.width = 2048;
  shadowLight.shadow.mapSize.height = 2048;

  // Добавляем освещение на сцену
  scene.add(hemisphereLight);
  scene.add(shadowLight);
}
```

Как видим, для создания света требуется задать значительное число параметров. Можно поэкспериментировать с цветом освещения, интенсивностью, количеством источников света и, возможно, получится сделаете сцену еще лучше.

18.3. Создание объектов

Далее создадим объекты, используя примитивы, доступные в Three.js. Библиотека Three.js включает большое число готовых к использованию примитивов (куб, сфера, цилиндр и др.). Комбинируя эти примитивы, создадим объекты сцены.

Создание моря

Начнём с самого простого объекта сцены, которым является море. Для простоты представим море как обычный голубой цилиндр в нижней части экрана. Пока этого достаточно, а позже сделаем море более реалистичным.

```
// Определяем JS-объект моря
Sea = function(){

  // Создаем геометрию цилиндра со следующими параметрами:
  // верхний радиус, нижний радиус, высота, количество
  // сегментов по окружности и по вертикали
  var geom = new THREE.CylinderGeometry(600,600,800,40,10);

  // Поворачиваем объект по оси x
  geom.applyMatrix(new THREE.Matrix4().makeRotationX(-Math.PI/2));

  // создаем материал для объекта
  var mat = new THREE.MeshPhongMaterial({
    color:Colors.blue,
    transparent:true,
    opacity:.6,
    shading:THREE.FlatShading,
  });

  // Для создания объекта в Three.js нужно создать меш, который
  // представляет собой совокупность созданных ранее геометрии и материала
  this.mesh = new THREE.Mesh(geom, mat);

  // Разрешаем морю отбрасывать тени
  this.mesh.receiveShadow = true;
}

// Инициализируем море и добавляем его на сцену
var sea;

function createSea(){
  sea = new Sea();

  // Подвинем объект в нижнюю часть сцены
  sea.mesh.position.y = -600;

  // Добавляем финальный меш на сцену
  scene.add(sea.mesh);
}
```

То есть использован следующий порядок создания 3D-объект в Three.js:

1. Создаем геометрию.

2. Создаем материал.
3. Объединяем геометрию и материал в меш.
4. Добавляем меш на сцену.

Выполняя эти действия, мы можем создавать различные примитивы. Комбинируя их, получим более сложные формы.

Создание облаков

Смоделируем облака как совокупность кубов разного размера, случайным образом расположеных в одном объекте (рис.18.4).

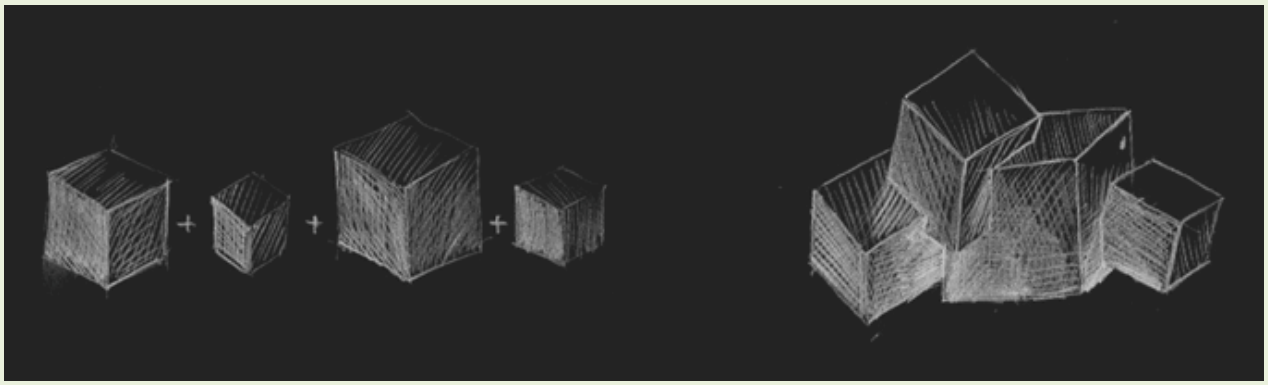

Рис. 18.4

```
Cloud = function(){
  // Создаем пустой контейнер, содержащий
  // составные части облаков (кубики)
  this.mesh = new THREE.Object3D();

  // Создаем геометрию кубика. Этот кубик мы
  // будем дублировать для создания облаков
  var geom = new THREE.BoxGeometry(20,20,20);

  // Создадим простой материал для нашего кубика
  var mat = new THREE.MeshPhongMaterial({
    color:Colors.white,
  });

  // Продублируем кубик произвольное число раз
  var nBlocs = 3+Math.floor(Math.random()*3);
  // Применим цикл для каждого куба и поместим их в наш меш
  for (var i=0; i<nBlocs; i++ ){
    // Создадим меш путем клонирования кубика циклом
    var m = new THREE.Mesh(geom, mat);

    // зададим случайную позицию и ротацию каждому кубику
    m.position.x = i*15;
    m.position.y = Math.random()*10;
    m.position.z = Math.random()*10;
    m.rotation.z = Math.random()*Math.PI*2;
    m.rotation.y = Math.random()*Math.PI*2;

    // Зададим произвольный размер кубикам
    var s = .1 + Math.random()*.9;
```

```
    m.scale.set(s,s,s);

    // Пусть кубики отбрасывают и преломляют тени
    m.castShadow = true;
    m.receiveShadow = true;

    // Поместим кубик в контейнер, созданный в начале этой функции
    this.mesh.add(m);
  }
}
```

Далее создадим небо из нескольких облаков, расположенных случайным образом на оси Z:

```
// Создадим объект неба
Sky = function(){
  // Создадим пустой контейнер
  this.mesh = new THREE.Object3D();

  // Количество облаков на небе будет равно 20
  this.nClouds = 20;

  // Чтобы распределить облака равномерно, рассчитаем
  // величину угла для каждого следующего облака
  var stepAngle = Math.PI*2 / this.nClouds;

  // Создадим облака для неба с помощью цикла
  for(var i=0; i<this.nClouds; i++){
    var c = new Cloud();

    // Зададим угол поворота и расстояние
    // между центром оси и облаком
    var a = stepAngle*i;
    var h = 750 + Math.random()*200;

    // Конвертируем полярные координаты (угол,
    // расстояние) в Декартовые координаты (x, y)
    c.mesh.position.y = Math.sin(a)*h;
    c.mesh.position.x = Math.cos(a)*h;

    // Поворачиваем облако относительно его позиции
    c.mesh.rotation.z = a + Math.PI/2;

    // Для большего реализма разместим облака
    // на случайной глубине на нашей сцене
    c.mesh.position.z = -400-Math.random()*400;

    // Установим случайный размер для каждого облака
    var s = 1+Math.random()*2;
    c.mesh.scale.set(s,s,s);

    // Добавим каждое облако в контейнер
```

```
    this.mesh.add(c.mesh);
  }
}

// Инициализируем небо и размещаем его внизу экрана
var sky;

function createSky(){
  sky = new Sky();
  sky.mesh.position.y = -600;
  scene.add(sky.mesh);
}
```

Создание самолета

Самолёт - более сложная модель, чем облака и море, но для его построения мы также будем использовать кубы, комбинируя их в нужном порядке. Модель самолёта, собранная из кубов, представлена на рис. 18.5.

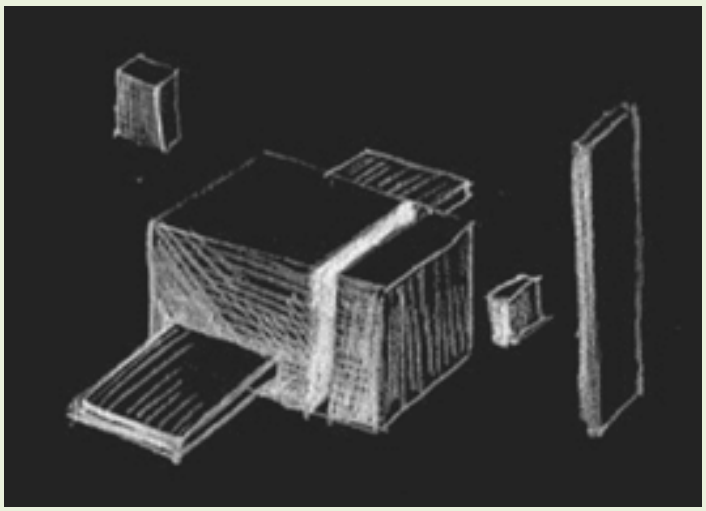

Рис. 18.5

```
var AirPlane = function() {

  this.mesh = new THREE.Object3D();

  // Создаем кабину
  var geomCockpit = new THREE.BoxGeometry(60,50,50,1,1,1);
  var matCockpit = new THREE.MeshPhongMaterial({color:Colors.red,
          shading:THREE.FlatShading});
  var cockpit = new THREE.Mesh(geomCockpit, matCockpit);
  cockpit.castShadow = true;
  cockpit.receiveShadow = true;
  this.mesh.add(cockpit);

  // Создаем двигатель
  var geomEngine = new THREE.BoxGeometry(20,50,50,1,1,1);
  var matEngine = new THREE.MeshPhongMaterial({color:Colors.white,
          shading:THREE.FlatShading});
  var engine = new THREE.Mesh(geomEngine, matEngine);
  engine.position.x = 40;
```

```
 engine.castShadow = true;
 engine.receiveShadow = true;
 this.mesh.add(engine);

 // Создаем хвост
 var geomTailPlane = new THREE.BoxGeometry(15,20,5,1,1,1);
 var matTailPlane = new THREE.MeshPhongMaterial({color:Colors.red,
        shading:THREE.FlatShading});
 var tailPlane = new THREE.Mesh(geomTailPlane, matTailPlane);
 tailPlane.position.set(-35,25,0);
 tailPlane.castShadow = true;
 tailPlane.receiveShadow = true;
 this.mesh.add(tailPlane);

 // Создаем крыло
 var geomSideWing = new THREE.BoxGeometry(40,8,150,1,1,1);
 var matSideWing = new THREE.MeshPhongMaterial({color:Colors.red,
        shading:THREE.FlatShading});
 var sideWing = new THREE.Mesh(geomSideWing, matSideWing);
 sideWing.castShadow = true;
 sideWing.receiveShadow = true;
 this.mesh.add(sideWing);

 // Создаем пропеллер
 var geomPropeller = new THREE.BoxGeometry(20,10,10,1,1,1);
 var matPropeller = new THREE.MeshPhongMaterial({color:Colors.brown,
        shading:THREE.FlatShading});
 this.propeller = new THREE.Mesh(geomPropeller, matPropeller);
 this.propeller.castShadow = true;
 this.propeller.receiveShadow = true;

 // Создаем лопасть
 var geomBlade = new THREE.BoxGeometry(1,100,20,1,1,1);
 var matBlade = new THREE.MeshPhongMaterial({color:Colors.brownDark,
        shading:THREE.FlatShading});

 var blade = new THREE.Mesh(geomBlade, matBlade);
 blade.position.set(8,0,0);
 blade.castShadow = true;
 blade.receiveShadow = true;
 this.propeller.add(blade);
 this.propeller.position.set(50,0,0);
 this.mesh.add(this.propeller);
};
```

Этот самолет выглядит несколько схематично, но далее его можно будет доработать, а пока что инициализируем его и добавим на сцену:

```
var airplane;

function createPlane(){
 airplane = new AirPlane();
```

```
  airplane.mesh.scale.set(.25,.25,.25);
  airplane.mesh.position.y = 100;
  scene.add(airplane.mesh);
}
```

Рендеринг сцены

Мы создали несколько объектов и добавили их на сцену. Но, если запустить игру, то мы ничего не увидим. Это потому, что мы не просчитали рендер сцены, поэтому добавим к коду строчку:

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

18.4. Управление сценой

Анимация объектов

Оживим игру, заставив перемещаться облака и море. Кроме того, сделаем пропеллер самолета вращающимся. Для этого нам понадобится бесконечный цикл:

```
function loop(){
  // Вращаем пропеллер, море и небо
  airplane.propeller.rotation.x += 0.3;
  sea.mesh.rotation.z += .005;
  sky.mesh.rotation.z += .01;

  // рендерим сцену
  renderer.render(scene, camera);

  // снова вызываем фунцию loop
  requestAnimationFrame(loop);
}
```

Мы перенесли вызов рендера в функцию loop(). Это сделано потому, что при изменении того или иного объекта нам необходимо пересчитать рендер сцены, чтобы увидеть эти изменения в объектах.

Добавляем реакцию на перемещение мыши

Сейчас самолет расположен в центре сцены. Мы заставим его следовать за курсором мыши. Для этого, когда страница загрузилась в браузере, необходимо повесить обработчик события на *document* и отслеживать перемещение курсора миши. Поэтому немного изменим функцию init():

```
function init(event){
  createScene();
  createLights();
  createPlane();
  createSea();
  createSky();

  //добавляем слушатель перемещения курсора мыши
  document.addEventListener('mousemove', handleMouseMove, false);

  loop();
}
```

Кроме того, нам нужно будет создать функцию, обрабатывающую передвижение курсора мыши:

```
var mousePos={x:0, y:0};

// Обрабатываем наше событие
function handleMouseMove(event) {
  // Конвертируем полученные значение положения мыши
  // в нормализолованное значение между -1 и 1;
  // вот формула для горизонтальной оси:
  var tx = -1 + (event.clientX / WIDTH)*2;

  // Для вертикальной оси необходимо инвертировать
  // формулу, так как в 2D ось Y идет в
  // противоположном направлении, в отличии от оси Y в 3D
  var ty = 1 - (event.clientY / HEIGHT)*2;
  mousePos = {x:tx, y:ty};
}
```

Имея нормализованное значение положения мыши по осям X и Y, мы можем перемещать самолет в соответствие с этими значениями. Для этого слегка модифицируем функцию loop() и добавим новую функцию для изменения позиции самолета:

```
function loop(){
  sea.mesh.rotation.z += .005;
  sky.mesh.rotation.z += .01;

  // Обновляем самолет в каждом кадре
  updatePlane();

  renderer.render(scene, camera);
  requestAnimationFrame(loop);
}

function updatePlane(){

  // Будем перемещать самолет в промежутке
  // от -100 до 100 по горизонтали и от 25 до 175
  // по вертикали, в зависимости от позиции курсора мыши,
  // у которого разброс значений между -1 и 1 в обоих
  // направлениях. Чтобы добиться этого, используем
  // функцию нормализации (normalize)

  var targetX = normalize(mousePos.x, -1, 1, -100, 100);
  var targetY = normalize(mousePos.y, -1, 1, 25, 175);

  // Обновляем позицию самолета
  airplane.mesh.position.y = targetY;
  airplane.mesh.position.x = targetX;
  airplane.propeller.rotation.x += 0.3;
}
```

```
function normalize(v,vmin,vmax,tmin, tmax){
  var nv = Math.max(Math.min(v,vmax), vmin);
  var dv = vmax-vmin;
  var pc = (nv-vmin)/dv;
  var dt = tmax-tmin;
  var tv = tmin + (pc*dt);
  return tv;
}
```

Мы сделали так, чтобы самолет реагировал на движения мыши.

Полный код можно посмотреть в файлах **ex18_01.html**, **ex18_01.js**.

Далее необходимо усовершенствовать 3D сцену, сделать чтобы самолет двигался более плавно, симитировать эффект волн на море.

18.5. Дорабатываем игру

Совершенствование сцены

Ранее мы создали очень простой самолетик, теперь модифицируем его и другие объекты, изменяя составляющие их примитивы.

На рис. 18.6 представлена усовершенствованная сцена с текстовой игровой информацией.

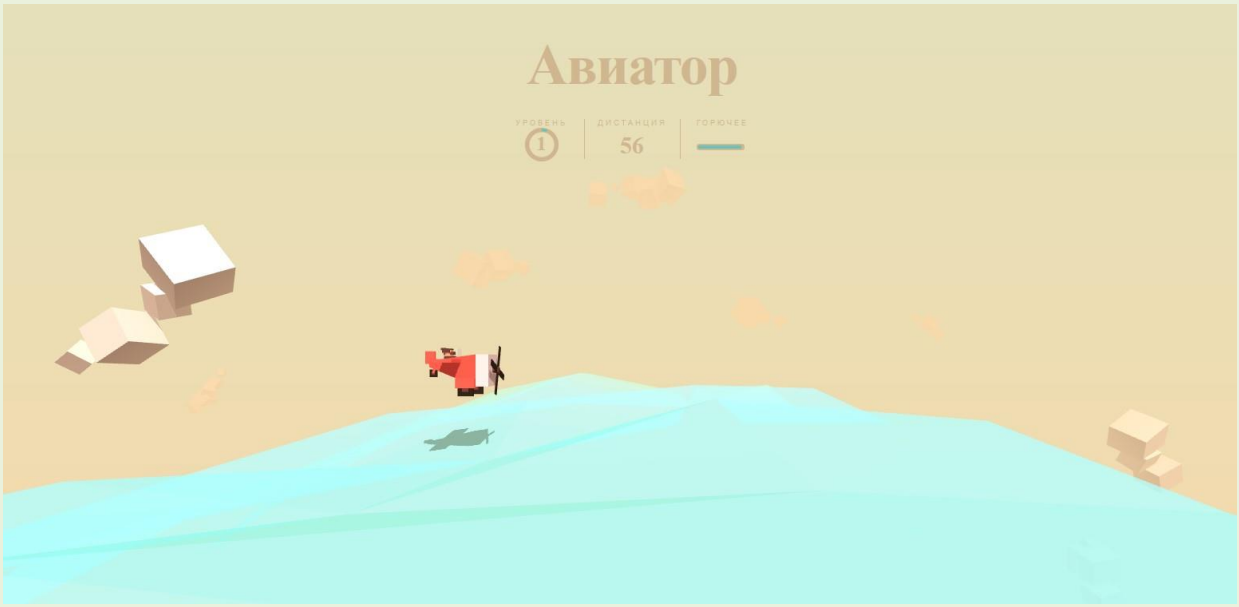

Рис. 18.6

Кубы, из которых состоит сложный объект, могут быть изменены путем перемещения их вершин. Модифицируем, например, кабину летчика, сделав её более узкой сзади (рис. 18.7).

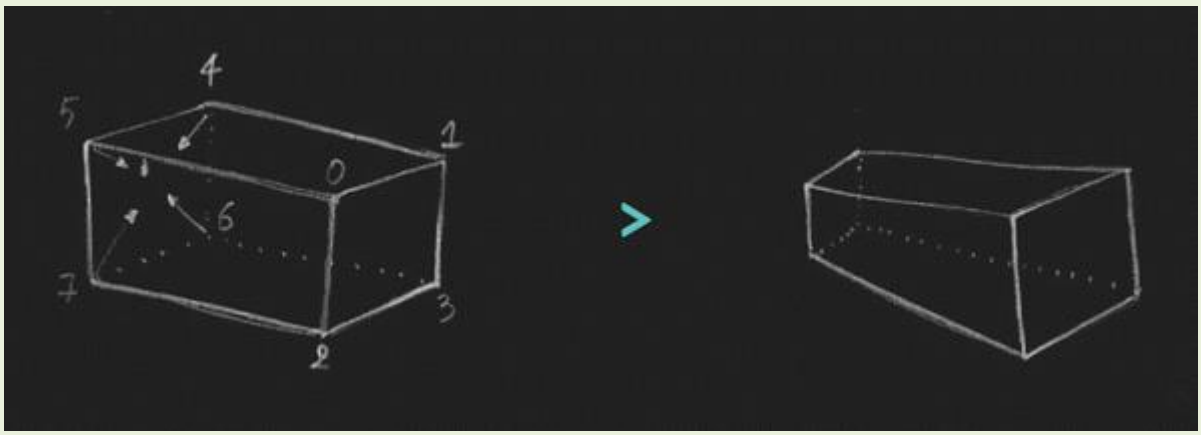

Рис. 18.7

```
// Кабина летчика
var geomCockpit = new THREE.BoxGeometry(80,50,50,1,1,1);
var matCockpit = new THREE.MeshPhongMaterial({color:Colors.red,
        shading:THREE.FlatShading});

// мы можем получить доступ к определенной вершине формы через
// массив vertices, а затем изменить её координты x, y, z:
geomCockpit.vertices[4].y-=10;
geomCockpit.vertices[4].z+=20;
geomCockpit.vertices[5].y-=10;
geomCockpit.vertices[5].z-=20;
geomCockpit.vertices[6].y+=30;
geomCockpit.vertices[6].z+=20;
geomCockpit.vertices[7].y+=30;
geomCockpit.vertices[7].z-=20;

var cockpit = new THREE.Mesh(geomCockpit, matCockpit);
cockpit.castShadow = true;
cockpit.receiveShadow = true;
this.mesh.add(cockpit);
```

Это пример того, как можно манипулировать формой, чтобы настроить её для целей моделирования объектов.

Если посмотреть на полный код самолета, мы увидим и другие объекты (например, окна и улучшенный пропеллер), которые можно модифицировать.

Модель пилота

Поскольку мы работаем на низкополигональной сцене, модель пилота также можно собрать из кубов.

Сделаем у пилота развивающиеся от ветра волосы. Для формирования такой модели также будем использовать кубы (рис. 18.8).

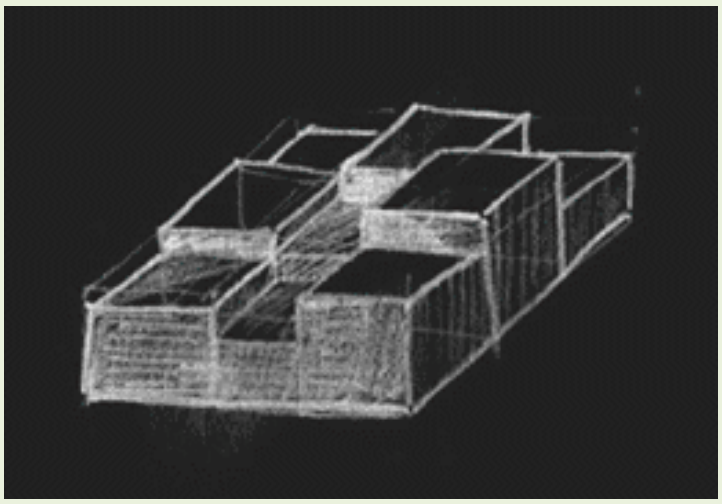

Рис. 18.8

Давайте рассмотрим код:

```
var Pilot = function(){
  this.mesh = new THREE.Object3D();
  this.mesh.name = "pilot";

  // angleHairs - свойство, используемое для анимации волос
  this.angleHairs=0;

  // Тело пилота
  var bodyGeom = new THREE.BoxGeometry(15,15,15);
  var bodyMat = new THREE.MeshPhongMaterial(
    {color:Colors.brown, shading:THREE.FlatShading}
  );
  var body = new THREE.Mesh(bodyGeom, bodyMat);
  body.position.set(2,-12,0);
  this.mesh.add(body);

  // Лицо пилота
  var faceGeom = new THREE.BoxGeometry(10,10,10);
  var faceMat = new THREE.MeshLambertMaterial({color:Colors.pink});
  var face = new THREE.Mesh(faceGeom, faceMat);
  this.mesh.add(face);

  // Элемент волос
  var hairGeom = new THREE.BoxGeometry(4,4,4);
  var hairMat = new THREE.MeshLambertMaterial({color:Colors.brown});
  var hair = new THREE.Mesh(hairGeom, hairMat);
  // Привести форму волос к нижней границе,
  // которая облегчит задачу с подбором размеров
  hair.geometry.applyMatrix(new THREE.Matrix4().makeTranslation(0,2,0));

  // создать контейнер для волос
  var hairs = new THREE.Object3D();

  // создать контейнер для волос в верхней части
  // головы (те из них, которые будут анимированы)
```

```
this.hairsTop = new THREE.Object3D();

// создать волосы в верхней части головы
// и расположить их на сетке 3 x 4
for (var i=0; i<12; i++){
    var h = hair.clone();
    var col = i%3;
    var row = Math.floor(i/3);
    var startPosZ = -4;
    var startPosX = -4;
    h.position.set(startPosX + row*4, 0, startPosZ + col*4);
    this.hairsTop.add(h);
}
hairs.add(this.hairsTop);

// создать волосы на лице (борода)
var hairSideGeom = new THREE.BoxGeometry(12,4,2);
hairSideGeom.applyMatrix(new THREE.Matrix4().makeTranslation(-6,0,0));
var hairSideR = new THREE.Mesh(hairSideGeom, hairMat);
var hairSideL = hairSideR.clone();
hairSideR.position.set(8,-2,6);
hairSideL.position.set(8,-2,-6);
hairs.add(hairSideR);
hairs.add(hairSideL);

// создать волосы на задней части головы
var hairBackGeom = new THREE.BoxGeometry(2,8,10);
var hairBack = new THREE.Mesh(hairBackGeom, hairMat);
hairBack.position.set(-1,-4,0)
hairs.add(hairBack);
hairs.position.set(-5,5,0);

this.mesh.add(hairs);

var glassGeom = new THREE.BoxGeometry(5,5,5);
var glassMat = new THREE.MeshLambertMaterial({color:Colors.brown});
var glassR = new THREE.Mesh(glassGeom,glassMat);
glassR.position.set(6,0,3);
var glassL = glassR.clone();
glassL.position.z = -glassR.position.z

var glassAGeom = new THREE.BoxGeometry(11,1,11);
var glassA = new THREE.Mesh(glassAGeom, glassMat);
this.mesh.add(glassR);
this.mesh.add(glassL);
this.mesh.add(glassA);

var earGeom = new THREE.BoxGeometry(2,3,2);
var earL = new THREE.Mesh(earGeom,faceMat);
earL.position.set(0,0,-6);
var earR = earL.clone();
earR.position.set(0,0,6);
```

```
    this.mesh.add(earL);
    this.mesh.add(earR);
}
// шевеление волос
Pilot.prototype.updateHairs = function(){

    // получаем волосы
    var hairs = this.hairsTop.children;

    // обновляем их в соответствии с углом angleHairs
    var l = hairs.length;
    for (var i=0; i<l; i++){
        var h = hairs[i];
        // каждый элемент волос будет масштабироваться
        // с помощью цикла от 75% до 100% от исходного размера
        h.scale.y = .75 + Math.cos(this.angleHairs+i/3)*.25;
    }
    // увеличение угла для следующего кадра
    this.angleHairs += 0.16;
}
```

На рис. 18.9 представлена модель самолёта с пилотом.

Рис. 18.9

Теперь, чтобы сделать движение волос, добавим эту строку в функцию loop:

```
airplane.pilot.updateHairs();
```

Создаем волны

Пока наше море представлено просто цилиндром с ребристой поверхностью. Создадим на нём эффект волн. Это может быть сделано путем объединения двух методов, использованных ранее:

1. Манипулирование геометрией вершины, как мы это делали с кабиной самолета.

2. Применение циклического движения к каждой вершине, как мы делали, чтобы переместить волосы пилота.

Чтобы создать волны, будем вращать каждую отдельную вершину цилиндра вокруг их начальных позиций со случайной скоростью и дистанцией (радиусом вращения), используя тригонометрические функции (рис. 18.10).

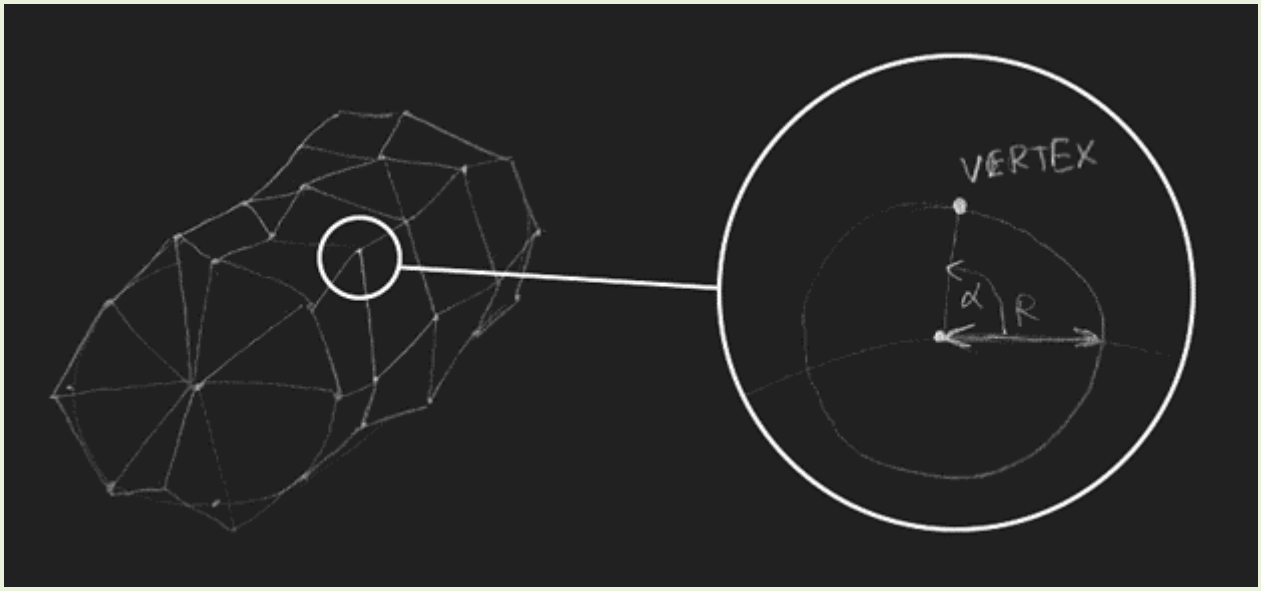

Рис. 18.10

Внесем некоторые изменения в модель моря:

```
Sea = function(){
  var geom = new THREE.CylinderGeometry(600,600,800,40,10);
  geom.applyMatrix(new THREE.Matrix4().makeRotationX(-Math.PI/2));

  // важно: путем слияния вершин мы обеспечиваем непрерывность волн
  geom.mergeVertices();

  // получаем вершины
  var l = geom.vertices.length;

  // создать массив для хранения новых данных, связанных с каждой вершиной
  this.waves = [];

  for (var i=0; i<l; i++){
    // получаем каждую вершину
    var v = geom.vertices[i];

    // сохраняем некоторые данные, связанные с нею
    this.waves.push({
      y:v.y,
      x:v.x,
      z:v.z,
      // случайный угол
      ang:Math.random()*Math.PI*2,
      // случайное расстояние
      amp:5 + Math.random()*15,
```

```
            // случайная скорость между 0,016 и 0,048 радиан/кадр
            speed:0.016 + Math.random()*0.032
        });
    };
    var mat = new THREE.MeshPhongMaterial({
        color:Colors.blue,
        transparent:true,
        opacity:.8,
        shading:THREE.FlatShading,
    });

    this.mesh = new THREE.Mesh(geom, mat);
    this.mesh.receiveShadow = true;
}
```

Теперь создадим функцию, которая будет вызываться в каждом кадре, чтобы обновить положение вершин для имитации волн.

```
Sea.prototype.moveWaves = function (){

    // получаем вершины
    var verts = this.mesh.geometry.vertices;
    var l = verts.length;

    for (var i=0; i<l; i++){
        var v = verts[i];

        // получить данные, связанные с ними
        var vprops = this.waves[i];

        // обновить положение вершины
        v.x = vprops.x + Math.cos(vprops.ang)*vprops.amp;
        v.y = vprops.y + Math.sin(vprops.ang)*vprops.amp;

        // увеличение угла для следующего кадра
        vprops.ang += vprops.speed;
    }

    // Сообщаем визуализатору, что геометрия моря изменилась.
    // На самом деле, для того, чтобы поддерживать оптимальный
    // уровень производительности, three.js кэширует геометрию и
    // игнорирует любые изменения, если мы не добавим эту строку
    this.mesh.geometry.verticesNeedUpdate=true;

    sea.mesh.rotation.z += .005;
}
```

На рис. 18.11 представлен результат – волны, сделанные на основе цилиндра.

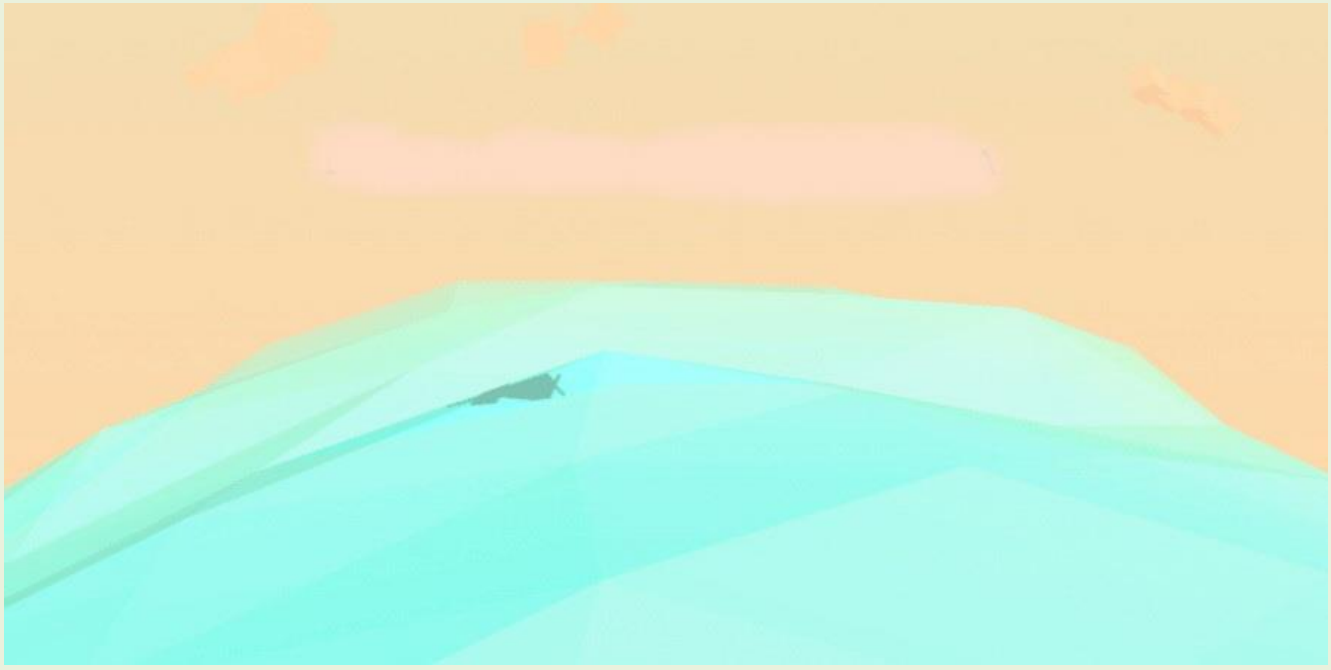

Рис. 18.11

Так же, как мы это делали для волос пилота, добавим строку в функцию loop:

```
sea.moveWaves();
```

Волны готовы.

Доработка освещения сцены

Освещение сцены уже существует. Можно теперь настроить яркость сцены и сделать более мягкие тени. Для этой цели будем использовать рассеянный свет. В функции createLights добавим следующие строки:

```
// AmbientLight изменяет глобальный цвет сцены
// и делает тени более мягкими
ambientLight = new THREE.AmbientLight(0xdc8874, .5);
scene.add(ambientLight);
```

Меняя цвета и интенсивность окружающего света, получим более насыщенную сцену.

Более плавный полет

Модель самолета уже следует за движениями мыши. Но полет при этом не выглядит реалистичным. Когда самолет меняет высоту, желательно, чтобы он изменял свое положение и ориентацию более плавно; реализуем это.

Самый простой способ сделать то, что мы задумали - заставить самолет двигаться к цели, добавив часть расстояния, которое отделяет его от этой цели в каждом кадре. Код будет выглядеть следующим образом:

```
currentPosition += (finalPosition - currentPosition)*fraction;
```

Для большей реалистичности, вращение самолета также можно изменять в зависимости от направления его движения. Если самолет взлетает слишком быстро, он должен быстро вращаться против часовой стрелки. Если он медленно опускается, то должен медленно вращаться по часовой. Для достижения точности мы можем просто присвоить

пропорциональное значение поворота к оставшемуся расстоянию между целью и положением самолета.

В коде функция updatePlane должна выглядеть следующим образом:

```
function updatePlane(){
   var targetY = normalize(mousePos.y,-.75,.75,25, 175);
   var targetX = normalize(mousePos.x,-.75,.75,-100, 100);

   // Перемещаем самолет в каждом кадре, добавив часть оставшегося расстояния
   airplane.mesh.position.y += (targetY-airplane.mesh.position.y)*0.1;

   // Вращаем самолет пропорционально оставшемуся расстоянию
   airplane.mesh.rotation.z = (targetY-airplane.mesh.position.y)*0.0128;
   airplane.mesh.rotation.x = (airplane.mesh.position.y-targetY)*0.0064;

   airplane.propeller.rotation.x += 0.3;
}
```

Теперь движение самолета выглядит более плавным и реалистичным. Изменяя значение fraction, можно сделать чтобы самолет быстрее или медленнее реагировал на движение мыши.

Полный код усовершенствованной сцены можно посмотреть в файлах **ex18_02.html**, **ex18_02.js**.

18.6. Добавляем игровой процесс

Далее из анимированной сцены с управлением объектом нам необходимо сделать игру. Сначала определим логику игры.

Целью игры буде пролететь как можно большее расстояние с начала полёта. С течением времени полёта у самолёта будет равномерно тратиться начальный запас горючего. В полёте нашему самолёту будут встречаться препятствия, от которых нужно будет уклоняться. Столкневение с каждым препятствием будет соответствующим образом менять траекторию движения самолёта и создавать эффект разрушения препятствия с помощью системы частиц («взрыв»); при этом будет убавляться запас горючего в самолёте на фиксированную величину. По пути будут встречаться и «полезные» объекты, увеличивающие запас горючего, если самолёт их «захватит».

«Нейтральные» объекты - облака – не меняют параметров полёта, но создают визуальные сложности при управлении самолётом.

Счётчик пройденного расстояния будет меняться в реальном времени.

При достижении заданного расстояния (каждой новой тысячи единиц) игра переходит на на следующий уровень, где можно увеличить скорость игры и число встречающихся объектов.

Счётчик запаса горючего также будет меняться в реальном времени и при достижении нулевой отметки самолёт будет падать в море – игра окончена.

Кнопка «Начать заново» сбрасывает все счётчики и начинает новую игру.

Полный код игры можно посмотреть в файлах **ex18_03.html**, **ex18_03.js**.

Контрольные вопросы

1. Структура программы.
2. Последовательность формирования 3D-сцены.
3. Добавление объектов к сцене.
4. Добавление управления сценой.