Программирование графических приложений

Тема 15
Система частиц

15.1. Работа с частицами в Canvas
15.2. Пример применения системы частиц
15.3. Поверхность из частиц

Контрольные вопросы

Цель изучения темы. Изучение методов применения системы частиц в WebGL.

15.1. Работа с частицами в Canvas

Система частиц покрывает огромный пласт визуальных эффектов. Дым, осадки, взрывы, искры, фейерверки - всё это и многое другое является частным случаем системы частиц, У каждой отдельной частицы может быть своя скорость, своё направление и даже внешний вид, но в то же время все они подчиняются определённому закону, общему для системы.

Каждую частицу в системе представляет отдельный объект. Это может быть полноценный 3D-объект, прямоугольник с заливкой или текстурой, или пиксельный объект (спрайт).

Рассмотрим последовательность создания ситемы частиц в WebGL без использования библиотек. Создадим новый элемент canvas размером 200x200:

```
<canvas id="example" width="200" height="200">Ваш браузер не поддерживает
        canvas</canvas>
```

Элемент canavas можно также добавить на страницу с помощью javascript

```
var canvas = document.createElement("canvas");
var context = canvas.getContext("2d");
canvas.width = 200;
canvas.height = 200;
document.body.appendChild(canvas);
```

Создаем context:

```
context.fillRect (0, 0, canvas.width, canvas.height);
```

Мы рисуем прямоугольник с координатами начала canvas'а и во всю ширину холста. Так как параметр fillStyle не указан, стиль заливки по умолчанию будет черным.

Добавление частицы

Добавим на холст частицу в виде белого квадрата с помощью метода fillRect:

```
context.fillStyle = "white";
context.fillRect(300, 200, 10, 10);
```

Можно задать частицу в форме круга или спрайта. Пример отрисовки круглой частицы:

```
context.beginPath();
context.fillStyle = "white";
context.arc(500, 200, 10, 0, Math.PI*2, true);
context.closePath();
context.fill();
```

Движение частицы

Эффект перемещения на элементе canvas создается с помощью покадровой перерисовки холста с предварительным очищением canvas перед рисованием нового кадра. Для этого чаще всего используется функция requestAnimationFrame() или функции setTimeout, setInterval.

Установим координаты частицы как переменные, чтобы в дальнейшем мы могли ее передвигать.

```
var posX = 20, posY = 100;
```

Добавим в код интервал перерисовки холста setInterval:

```
// Рисуем частицу на холсте с интервалом в 30ms
setInterval (function()
{
  // Очищаем canvas context.fillStyle = "black";
  context.fillRect (0,0,canvas.width, canvas.height);
  posX += 1;
  // На каждой итерации меняем позиции X и Y
  posY += 0.25;
  // Рисуем частицу
  context.beginPath();
  context.fillStyle = "white";
  context.arc(posX, posY, 10, 0, Math.PI*2, true);
  context.closePath();
  context.fill();
}, 30);
```

Скорость и гравитация

Введем переменные гравитации и векторной скорости:

```
var  vx = 10, vy = -10, gravity = 1;
```

Векторная скорость указывает на то, что каждую итерацию (30мс) частица будет перемещаться на 10 пикселей вправо и 10 пикселей вниз. Заменим часть кода в нашем примере и добавим движение с прибавлением векторной скорости и учётом гравитации в векторной скорости по вертикали:

```
posX + vx;
poxY + vY;
posY + gravity;
```

Отскок

Для того, чтобы частицы не проваливались вниз, а отскакивали от нижней поверхности, можно «переворачивать» векторную скорость по оси Y в нужный момент и домножать на произвольный коофицент отскока.

```
if (posY > canvas.height * 0.75) {
vy *= -0.6;
vx *= 0.75;
posY = canvas.height * 0.75;
}
```

Если положение частицы ниже, чем две трети (*0.75) холста, умножаем вектор скорости на коэффициент отскока, равный 60% (т.е. каждый следующий отскок будет ниже на 40%), и замедляем на 25% скорость по горизонтали при каждом отскоке.

Создание системы частиц

Перейдем к созданию нескольких частиц и добавим элемент случайности в их движение.

Нам необходимо расширить код для создания множества частиц. Для этого напишем функцию с именем Particle, которая будет создавать частицы и управлять ими. Реализуем также логику хаотичного перемещения каждой частицы в отдельности. Первым делом вынесем все настройки будущих частиц в один объект.

```
var particles = {}, particleIndex = 0, settings = { density: 20, particleSize: 10, startingX:
        canvas.width / 2, startingY: canvas.height / 4, gravity: 0.5, maxLife: 100 };
```

Мы определили пустой объект Particles, в котором будут храниться частицы, переменную для индекса и объект с настройками размера, стартовой позиции, гравитации и жизни частицы. Вместо того, чтобы хранить частицы в массиве, будем использовать объект, т.к. далее будет проще работать с ключами, а не с индексами. Функция, которая будет генерировать частицу:

```
function Particle() {
// Устанавливаем позицию согласно настройкам
this.x = settings.startingX;
this.y = settings.startingY;
// Случайные X и Y скорости
this.vx = Math.random() * 20 - 10;
this.vy = Math.random() * 20 - 5;
//Увеличиваем индекс и добавляем частицу с новым индексом.
particleIndex ++; particles[particleIndex] = this;
this.id = particleIndex; this.life = 0;
}
```

Рисование множества частиц

Для рисования мы должны расширить наш объект Particle и добавирь метод draw:

```
Particle.prototype.draw = function() {
this.x += this.vx;
this.y += this.vy;
// Добавляем гравитацию
this.vy += settings.gravity;
// Отсчитываем жизнь
this.life++;
// Если жизнь частицы больше максимальной - удаляем
if (this.life >= settings.maxLife) { delete particles[this.id]; }
// Рисуем частицу
context.clearRect(settings.leftWall, settings.groundLevel, canvas.width, canvas.height);
context.beginPath();
context.fillStyle="#ffffff";
context.arc(this.x, this.y, settings.particleSize, 0, Math.PI*2, true); context.closePath();
        context.fill();
}
```

Метод draw выполняет сразу несколько функций. Во-первых, он регулирует скорость частицы и перемещает её. Так как значения скорости по осям были случайны, частица будет двигаться хаотично. Также этот метод подсчитывает время жизни частицы и при необходимости удаляет устаревшие частицы. Кроме того направление по оси Y корректируется с учетом силы тяжести.

Полный код примера – в файлах **ex15_01.html**, **ex15_01.js**.
Результат:

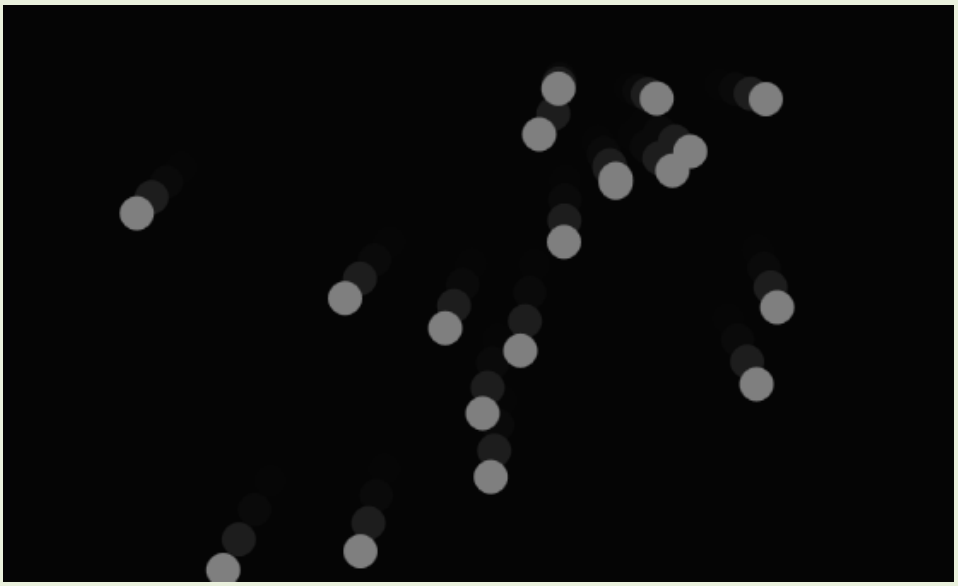

Рис. 15.1

15.2. Пример применения системы частиц

Для демонстрации работы системы частиц смоделируем бенгальский огонь, являющийся типичным примером этого класса объектов. У нас будет источник появления частиц (эмиттер), а каждая частица будет имеет свою скорость, направление и время жизни. Дополнительно каждая частица будет оставлять за собой след, как и все искры.

Для отображения искр будем использовать спрайт, так как для каждой частицы достаточно всего одной вершины. К тому же спрайт по определению всё время повёрнут в сторону наблюдателя, независимо от поворота сцены, и искра всегда остаётся искрой.

Одна частица

Рассмотрим простой пример - отображение одной частицы (искры бенгальского огня). Для написания кода понадобятся только базовые принципы WebGL и знание основ текстурирования. Для искры нужно изображение с прозрачным фоном, которое мы будем использовать в качестве текстуры (**spark.png**):

Рис. 15.2

Мы можем задать размер точки с помощью gl_PointSize в вершинном шейдере, и тогда точка превращается в квадрат указанного размера с центром в этой точке. Поведение такой фигуры будет отличаться от квадрата, заданного двумя треугольниками: квадрат из двух треугольников будет подвержен матричным преобразованиям, то есть он будет вращаться при повороте и уменьшаться при удалении от наблюдателя, в то время как заданный одной точкой квадрат будет всегда направлен к наблюдателю и иметь размер gl_PointSize.

Вершинный шейдер будет самым обычным, добавится лишь задание размера квадрата через gl_PointSize. Желательно, чтобы размер точки совпадал с размером изображения искры.

```
attribute vec3 a_position;
uniform mat4 u_mvMatrix;
uniform mat4 u_pMatrix;
void main() {
   gl_Position = u_pMatrix * u_mvMatrix * vec4(a_position, 1.0);
   // размер искры
   gl_PointSize = 32.0;
}
```

Фрагментный шейдер будет типовым для использования текстур:

```
precision mediump float;
uniform sampler2D u_texture;
void main() { gl_FragColor = texture2D(u_texture, gl_PointCoord);  }
```

Теперь рассмотрим код JavaScript. Сначала необходимо загрузить изображение и инициализировать текстуру искры, только после этого мы начинаем отрисовывать сцену.

```
var texture = gl.createTexture();
var image = new Image();
image.src = "spark.png";
image.addEventListener('load', function() {
   gl.bindTexture(gl.TEXTURE_2D, texture);
   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
       image);
   gl.generateMipmap(gl.TEXTURE_2D);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
   gl.bindTexture(gl.TEXTURE_2D, null);
   // отрисовку сцены начинаем только после загрузки изображения
   requestAnimationFrame(drawScene);
});
```

Для фильтрации текстуры используется значение gl.NEAREST вместо значения по умолчанию gl.LINEAR. Дело в том, что при использовании gl.LINEAR при определённых значениях размера canvas (например, 512x513) текстура незначительно меняется в размере и пиксель начинает формироваться из нескольких окружающих его пикселей, из-за чего искра получается немного размытой (правая часть изображения). При значении фильтра gl.NEAREST берётся один наиболее подходящий пиксель, и искра выглядит сравнительно более чёткой (левая часть изображения).

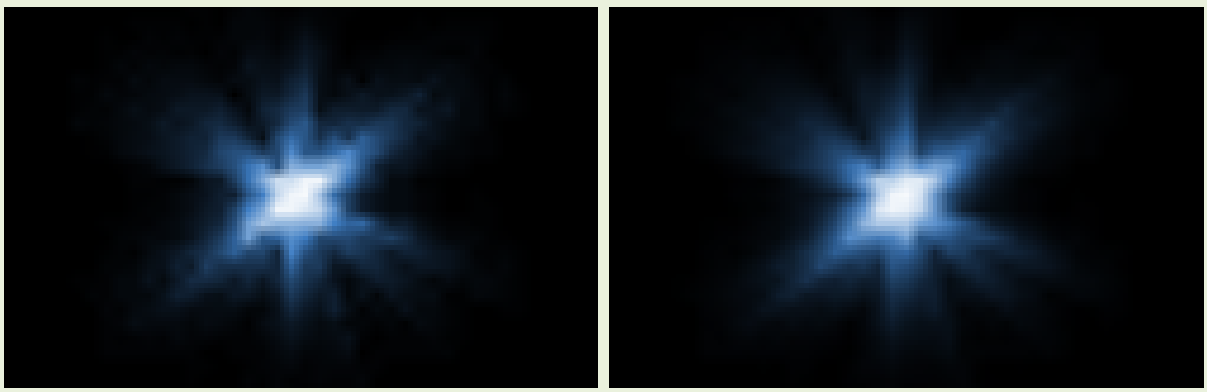

Рис. 15.3

Для того чтобы WebGL понимал прозрачность и избавлялся от фона искры, нам необходимо включить смешивание цветов.

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE);
```

Смешивание определяет как будут взаимодействовать между собой пиксели, которые уже были отрисованы, с пикселями, которые рисуются в данный момент. Когда система частиц заработает, искры будут многократно накладываться друг на друга и без правильного задания смешивания нельзя ожидать приемлемого результата. Вызов gl.enable(gl.BLEND) включает смешивание, а функция gl.blendFunc определяет взаимодействие рисуемых и уже отрисованных пикселей. Первый параметр функции - множитель для рисуемого пикселя. Значение gl.SRC_ALPHA означает, что каждый канал рисуемого пикселя нужно умножить на прозрачность рисуемого пикселя. Так мы уберём прозрачный фон изображения и получим только изображение искры. Второй параметр — множитель для уже отрисованного пикселя. Значение gl.ONE означает, что уже отрисованные пиксели никак не меняются (умножение на 1 не меняет значение).

Остальной код шаблонный - инициализация буферов, uniform-переменных, атрибутов. В результате получится код, который можно посмотреть в файле **ex15_02.html**.

Результат:

Рис. 15.4

Следы частиц

Искры бенгальского огня оставляют за собой след, идущий от источника появления частиц до текущего положения искры. Этот след можно представить отрезком прямой, ограниченной диумя точками: одна из них будет текущей координатой частицы, а вторая - координатой эмиттера (в нашем случае эмиттер располагается в точке 0,0,0). Причём след будет более яркий возле эмиттера, а по мере отдаления будет становиться менее ярким, приближаясь к оранжевому — так мы сымитируем остывание искры.

Но сейчас наши шейдеры не приспособлены для отображения линии: они отображают точку вместо линии и текстуру вместо цвета. Поэтому нам понадобится ещё один набор шейдеров и, соответственно, ещё одна программа.

Вершинный шейдер линии через varying-переменную будет передавать значение цвета во фрагментный шейдер, а также определять текущую координату частицы:

```
attribute vec3 a_position;
attribute vec3 a_color;
varying vec3 v_color;
uniform mat4 u_mvMatrix;
uniform mat4 u_pMatrix;
void main() {
   v_color = a_color;
```

```
    gl_Position = u_pMatrix * u_mvMatrix * vec4(a_position, 1.0);
}
```

Фрагментный шейдер будет просто устанавливать итоговый цвет пикселя из значения varying-переменной:

```
precision mediump float;
varying vec3 v_color;
void main() {      gl_FragColor = vec4(v_color, 1.0);   }
```

Нам понадобится две программы и два набора ссылок на атрибуты и uniform-переменные:

```
// инициализация программы следов искр
var programTrack = webglUtils.createProgramFromScripts(gl, ["vertex-shader-track",
        "fragment-shader-track"]);
var positionAttributeLocationTrack = gl.getAttribLocation(programTrack,
        "a_position");
var colorAttributeLocationTrack = gl.getAttribLocation(programTrack, "a_color");
var pMatrixUniformLocationTrack = gl.getUniformLocation(programTrack,
        "u_pMatrix");
var mvMatrixUniformLocationTrack = gl.getUniformLocation(programTrack,
        "u_mvMatrix");

// инициализация программы искр
var programSpark = webglUtils.createProgramFromScripts(gl, ["vertex-shader-spark",
        "fragment-shader-spark"]);
var positionAttributeLocationSpark = gl.getAttribLocation(programSpark,
        "a_position");
var textureLocationSpark = gl.getUniformLocation(programSpark, "u_texture");
var pMatrixUniformLocationSpark = gl.getUniformLocation(programSpark,
        "u_pMatrix");
var mvMatrixUniformLocationSpark = gl.getUniformLocation(programSpark,
        "u_mvMatrix");
```

В отрисовке сцены появится вызов двух функций - отрисовка следов и отрисовка искр. В каждую функцию будут передаваться координаты всех существующих на данный момент искр (сейчас у нас будет три искры вместо одной):

```
var positions = [
    1, 0,   0,
   -1, 0.5, 0,
   -0.5, -1, 0
];
drawTracks(positions);
drawSparks(positions);
```

В начале каждой функции отрисовки drawTracks и drawSparks будет вызываться gl.useProgram для установки текущей программы. Таким образом, при отрисовке искр функцией drawSparks будет активироваться программа programSpark, и шейдеры этой программы отобразят точку с текстурой. А при отрисовке следов искр функцией drawTracks будет активироваться программа programTrack, и соответствующие шейдеры отобразят линию с плавным переходом цвета от белого к оранжевому.

Для отрисовки следов искр нам необходимо дополнить массив координат нулевыми точками для координаты каждой искры (чтобы получить линию из двух координат), а также задать цвета - 3 канала для точки появления искры (1, 1, 1 - белый) и 3 канала для текущей позиции искры (0.47, 0.31, 0.24 - оранжевый). Шейдер сам сделает плавный переход цвета от белого к оранжевому на протяжении линии:

```
var colors = [];
var positionsFromCenter = [];
for (var i = 0; i < positions.length; i += 3) {
   // для каждой координаты добавляем точку начала координат, чтобы получить
      след искры
   positionsFromCenter.push(0, 0, 0);
   positionsFromCenter.push(positions[i], positions[i + 1], positions[i + 2]);
   // цвет в начале координат будет белый (горячий), а дальше будет приближаться
      к оранжевому
   colors.push(1, 1, 1, 0.47, 0.31, 0.24);
}
```

Полный код можно посмотреть в файле **ex15_03.html**.

В результате мы получим искры со следами:

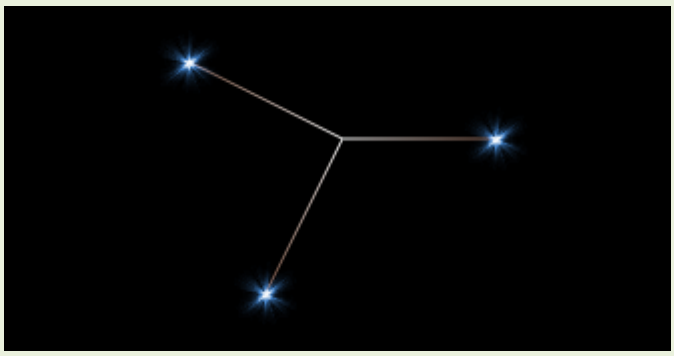

Рис. 15.5

Система частиц

Осталось значительно увеличить количество искр и заставить их двигаться. Алгоритм работы бенгальского огня будет следующий:

- создаём искру с произвольным направлением, скоростью и длиной пути;
- при каждой отрисовке увеличиваем положение искры на приращение;
- когда искра пройдёт весь отрезок пути, запускаем её заново из начала координат.

Создадим класс для искры с двумя функциями - функция init вызывается для создания и инициализации искры, функция move вызывается при каждом цикле отрисовки для приращения координат искры.

```
function Spark() {    this.init(); };

// количество искр
Spark.sparksCount = 200;

Spark.prototype.init = function() {
   // время создания искры
   this.timeFromCreation = performance.now();

   // задаём направление полёта искры в градусах, от 0 до 360
```

```
    var angle = Math.random() * 360;
    // радиус - это расстояние, которое пролетит искра
    var radius = Math.random();
    // отмеряем точки на окружности - максимальные координаты искры
    this.xMax = Math.cos(angle) * radius;
    this.yMax = Math.sin(angle) * radius;

    // dx и dy - приращение искры за вызов отрисовки, то есть её' скорость,
    // у каждой искры своя скорость. multiplier подобран экспериментально
    var multiplier = 125 + Math.random() * 125;
    this.dx = this.xMax / multiplier;
    this.dy = this.yMax / multiplier;

    // Для того, чтобы не все искры начинали движение из начала координат,
    // делаем каждой искре свой отступ, но не более максимальных значений.
    this.x = (this.dx * 1000) % this.xMax;
    this.y = (this.dy * 1000) % this.yMax;
};

Spark.prototype.move = function(time) {
    // находим разницу между вызовами отрисовки, чтобы анимация работала
    // одинаково на компьютерах разной мощности
    var timeShift = time - this.timeFromCreation;
    this.timeFromCreation = time;

    // приращение зависит от времени между отрисовками
    var speed = timeShift;
    this.x += this.dx * speed;
    this.y += this.dy * speed;

    // если искра достигла конечной точки, запускаем её заново из начала координат
    if (Math.abs(this.x) > Math.abs(this.xMax) || Math.abs(this.y) > Math.abs(this.yMax))
            {
        this.init();
        return;
    }
};
```

Вся основная логика уместилась в одном классе. Нам осталось лишь создать необходимое количество искр при инициализации программы

```
    var sparks = [];
    for (var i = 0; i < Spark.sparksCount; i++) {     sparks.push(new Spark());    }
```

Теперь надо вызвать смещение искр при каждой отрисовке и получить координаты искр для передачи в функции drawTracks и drawSparks

```
    for (var i = 0; i < sparks.length; i++) {      sparks[i].move(now);    }

    var positions = [];
    sparks.forEach(function(item, i, arr) {
        positions.push(item.x);
        positions.push(item.y);
```

```
    // искры двигаются только в одной плоскости xy
    positions.push(0);
  });

  drawTracks(positions);
  drawSparks(positions);
```

Полный код можно посмотреть в файле **ex15_04.html**.
В результате получится полноценная система частиц бенгальского огня:

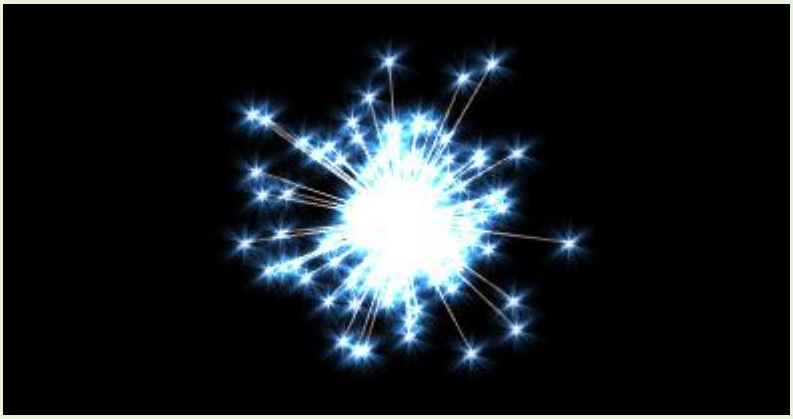

Рис. 15.6

15.3. Поверхность из частиц

Рассмотрим еще один способ применения системы частиц: построение поверхности из частиц в пространстве, воспроизводящей какой-то сложный рельеф. Это может быть, например, рельеф поверхности земли. Используем библиотеку Three.js.

Структура программы будет традиционной – файлы **ex15_05.html, ex15_05.js**. Скрипт **ex15_05.js** также имеет традиционный набор функций: init, animate, render, AddCamera, AddLight, отвечающих за инициализацию сцены, её отрисовку, создание и добавление на сцену объектов (частиц particles), освещения, камеры и фона. Функция init:

```
function init()
{
 scene = new THREE.Scene(); //создаем сцену
 AddCamera( 0, 200, 1700); //добавляем камеру
 AddLight( 0, 0, 500 ); //устанавливаем белый свет

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

 //создаём чёрный фон
 var floorTexture = new THREE.ImageUtils.loadTexture( 'black.jpg' );
 floorTexture.wrapS = floorTexture.wrapT = THREE.RepeatWrapping;
          floorTexture.repeat.set( 10, 10 );
```

```
var floorMaterial = new THREE.MeshBasicMaterial( { map: floorTexture, side:
        THREE.DoubleSide } );
var floorGeometry = new THREE.PlaneGeometry(10000, 10000, 10, 10);
var floor = new THREE.Mesh(floorGeometry, floorMaterial);
scene.add(floor);

//создаём частицы
var terrainSize = 200; //размер поверхности
var geometry = new THREE.Geometry();
var vertex;
for (var i = 0; i < terrainSize; i++) {
  for (var j = 0; j < terrainSize; j++) {
    vertex = new THREE.Vector3();
    vertex.x = (i - terrainSize / 2) * 8 + (Math.random() - 0.5) * 8;
    vertex.y = 125;
    vertex.z = (j - terrainSize / 2) * 8 + (Math.random() - 0.5) * 8;
    geometry.vertices.push(vertex);
  }
}

// Создание текстуры
var textureLoader = new THREE.TextureLoader();
var material = new THREE.PointsMaterial({
      color: 0xffdb8f,
      size: 10,
      map: textureLoader.load("particle.png"),
      blending: THREE.AdditiveBlending,
      transparent: true
 });
// добавляем систему частиц на сцену
var particles = new THREE.Points(geometry, material);
scene.add(particles);
}
```

Изображение, которое использовалось для создания текстуры (**particle.png**):

Рис. 15.7

В качестве фона использована чёрная текстура.
Полный код можно посмотреть в файлах **ex15_05.html**, **ex15_05.js**.
Результат:

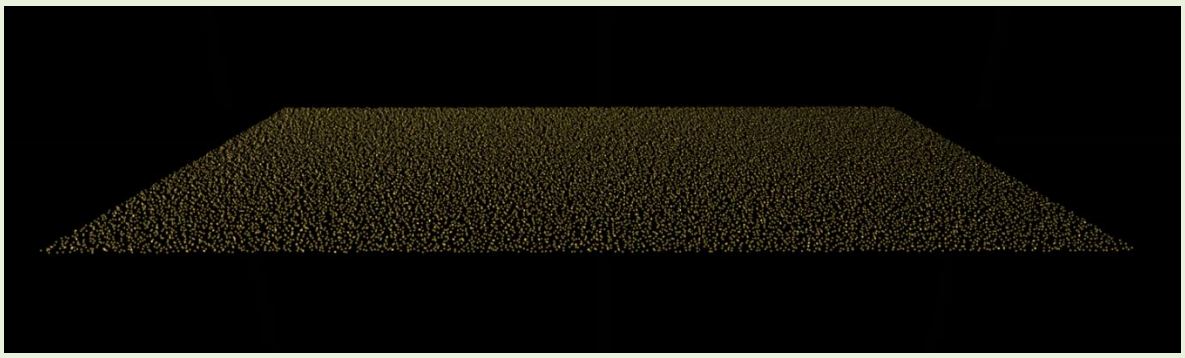

Рис. 15.8

Получилась плоскость из частиц. Теперь нужно изменить высоты частиц таким образом, чтобы добиться рельефа.

Создание рельефа

При добавлении частиц в функции init размер поверхности terrainSize фиксированный и равен 200x200 точек. Далее все точки сдвигаются на половину размера поверхности, чтобы она отображалась по центру сцены, координаты пропорционально увеличиваются и добавляется небольшой случайный сдвиг, чтобы точки не располагались ровными рядами.

Координата Y у всех точек одинакова, поэтому поверхность получилась плоской. Наша задача изменить координаты Y таким образом, чтобы получился рельеф. Для этого будем использовать карту высот — это черно-белое изображение, каждая точка которого обозначает высоту рельефа. Чем темнее точка на карте, тем выше точка на поверхности. Такое изображение можно нарисовать в любом графическом редакторе или использовать готовое (**terrain.png**):

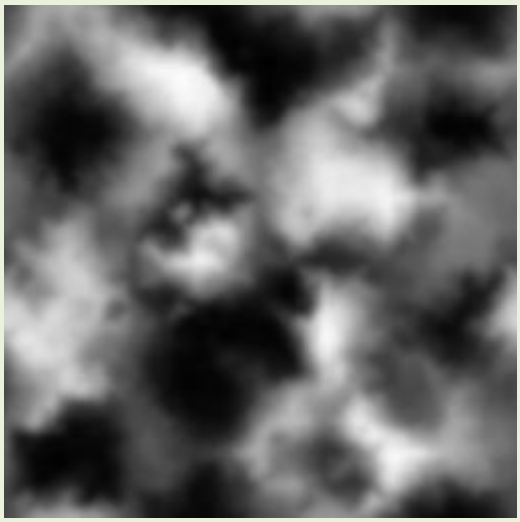

Рис. 15.9

Это изображение необходимо преобразовать в массив высот. Для этого можно составить программу на любом языке: она должна считать изображение и получить цвет в каждой точке. Получившийся результат сохраняем в файл **terrain-bitmap.js** и подключаем его к нашей программе. Содержимое этого файла будет примерно таким:

```
var terrain = [
   [74, 74, 75, 76, ...],
   [75, 76, 76, 77, ...],
   ...
];
```

Для простоты будем считать, что изображение квадратное. В метод init добавим еще один параметр массив высот; значение переменной terrainSize сделаем равным размеру массива:

```
var terrainSize = terrain.length;
```

Изменим способ генерации частиц:

```
var vertex, value;
for (var i = 0; i < terrainSize; i++) {
   for (var j = 0; j < terrainSize; j++) {
      value = terrain[i][j];
      vertex = new THREE.Vector3();
      vertex.x = (i - terrainSize / 2) * 8 + (Math.random() - 0.5) * 8;
      vertex.y = value / 3;
      vertex.z = (j - terrainSize / 2) * 8 + (Math.random() - 0.5) * 8;
      geometry.vertices.push(vertex);
   }
}
```

Координата Y теперь задаётся на основе значения высоты (все координаты Y делятся на константу, чтобы визуально уменьшить перепады высот).

Полный код можно посмотреть в файлах **ex15_06.html**, **ex15_06.js**.

Результат:

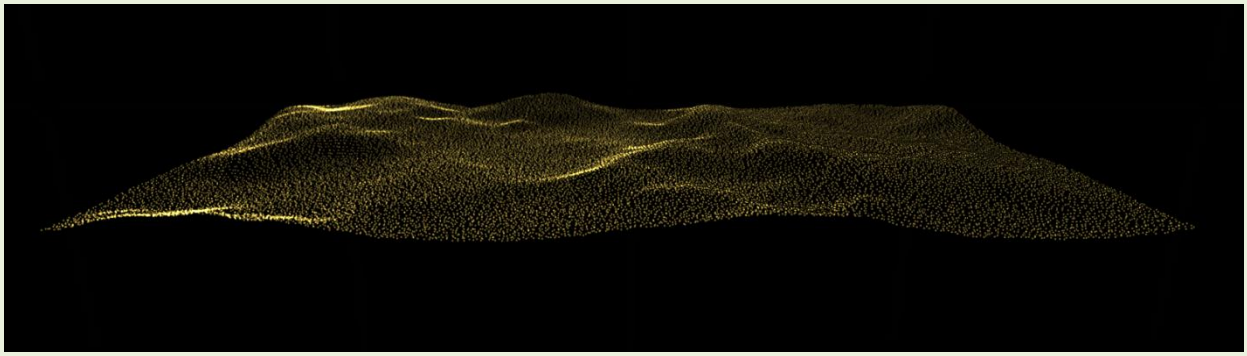

Рис. 15.10

Мы получили рельефную поверхность из точек. Чтобы сделать края этой поверхности менее резкими, можно добавить вероятность отображения каждой точки в зависимости от расстояния до центра поверхности (чем больше расстояние, тем меньше вероятность) и от значения высоты (чем больше высота, тем больше вероятность):

```
valueProb = value / terrainSize;
distanceProbX = Math.abs(i1 / (terrainSize / 2));
distanceProbY = Math.abs(j1 / (terrainSize / 2));

if (Math.random() < valueProb
   && Math.random() > distanceProbX
   && Math.random() > distanceProbY) {
   // добавляем точку
}
```

Полный код можно посмотреть в файлах **ex15_07.html**, **ex15_07.js**.

Результат:

Рис. 15.11

В качестве частиц вместо текстур можно использовать другие объекты – точки, сферы, кубы и т.д. При этом время формирования кадра может заметно увеличиться. Например, в нашем примере формирования рельефной поверхности текстурированные частицы можно заменить на сферы:

```
var terrainSize = terrain.length; //размер поверхности
// Создание частиц
var geometry = new THREE.SphereGeometry(5, 5, 5);
var material = new THREE.MeshLambertMaterial( { color: 0xffdb8f } );
for (var i = 0; i < terrainSize; i++) {
    for (var j = 0; j < terrainSize; j++) {
      var Psphere = new THREE.Mesh( geometry, material );
      Psphere.position.x = (i - terrainSize / 2) * 8 + (Math.random() - 0.5) * 8;
      Psphere.position.y = terrain[i][j] / 3;
      Psphere.position.z = (j - terrainSize / 2) * 8 + (Math.random() - 0.5) * 8;
      scene.add (Psphere);
}  }
```

Полный код можно посмотреть в файлах **ex15_08.html**, **ex15_08.js**.
Результат без использования чёрного фона:

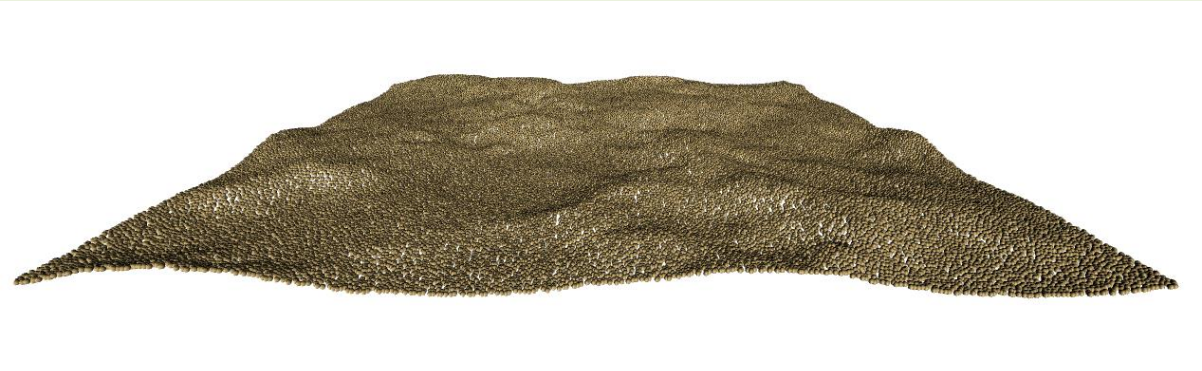

Рис. 15.12

Контрольные вопросы

1. Система частиц.
2. Поверхность из частиц.