Сентябрь 2005

Подарок от Redikod:
Учебное пособие по 3D- программированию для мобильных устройств, часть три: системы Частиц и непосредственный режим рендеринга

Мы теперь достигли третьей части серии учебных пособий по M3G (JSR 184), созданных Mikael Baros, старшим программистом из Redikod.

Рассмотрев некоторые основы в частях один и два, он теперь проведет Вас через несколько более продвинутых методик, программирования M3G.

Ниже Вы можете загрузить и zip файлы исходного кода и пакета приложения для части три, так же можете освежить вашу память двумя частями из учебного пособия.

Часть три: системы Частиц и непосредственный режим рендеринга

Часть три путеводителя от Mikael-a.

Введение

Добро пожаловать в третью часть обучающей программы по M3G! Сегодня я пройду, как получить полный контроль над процессом рендеринга(непосредственный режим рендеринга) и как создать очень хорошую систему частиц. Вот - также все ссылки из последней обучающей программы, на всякий случай если Вы их потеряли:

Прежде всего, и вероятно наиболее важно, является секция посвященная Mobile Java 3D на портале Мир Разработчиков фирмы Sony Ericsson: Sony Ericsson Developer World.
Во вторых, если у Вас когда-либо будут проблемы –посетите форум Sony Ericsson Mobile Java 3D forum. Для всего остального, используйте Web портал Мир Разработчиков Sony Ericsson, где Вы найдете ответы на ваши вопросы и более того.

Цель этой обучающей программы состоит в том, чтобы показать Вам, как рендерить тот же самый объект несколько раз, с различными трансформациями. Это называют непосредственным режимом. Вы будете учиться применять свехмощный этот способ - для очень многих вещей. Также, эта обучающая программа будет основой для дальнейших более продвинутых обучающих программ, так как с этого времени мы будем почти всегда исключительно использовать непосредственный режим. Так как код предназначен для образовательных целей, то он не оптимален, и не закрывает все ошибки, с которыми Вы могли бы столкнуться при 3D-программировании. Они - более продвинутые темы, к которым будут обращаться позже.
Что Вы должны знать
Прежде, чем Вы начнете читать это, Вы должны были прочитать первые две части обучающей программы, чтобы иметь несколько устойчивый охват основных функциональных возможностей M3G.

- Часть первая: Быстрый скачок в мир программирования Mobile Java 3D
- Часть два: Теория 3D освещения и ориентации

Сохраненный режима против Непосредственного режима рендеринга

Сохраненный режим - используется, когда Вы рендерите полный мир со всей информацией, которую мир содержит, включая камеры и источники освещения.
Это - симпатичный ограниченный режим, так как мы почти всегда хотим многократно чертить единичные модели(объекты), с различными трансформациями, не используя полный граф сцены.
Так, когда мы отдаем единственную группу, узел или подпетлю(submesh) в M3G,то это называется непосредственным режимом рендеринга. Методы непосредственного режима рендеринга:
render (Node node, Transform transform)
render (VertexBuffer vertices, IndexBuffer triangles, Appearance appearance, Transform transform)
render (VertexBuffer vertices, IndexBuffer triangles, Appearance appearance, Transform transform, int scope)

Как можете видеть, все три из методов требуют некоторые данные вершин: Узел(Node) и Буферы вершин/Индексов. Узлом может в основном быть любая часть графа сцены, даже мир рассматривают

как Узел. Обычно Вы передаете Мешь или Группу по первому методу рендеринга. Буфер вершин(VertexBuffer), о котором мы говорили во второй обучающей программе, является набором данных Меша, который описывает модель в 3D-пространстве.
Последние два метода также требуют передачи класса Внешность(Appearance), для понимания как отображать внешность Меша. В этой обучающей программе мы будем только использовать первый метод, чтобы показывать Вам, как работает непосредственный режим.

Другая вещь, которую все методы имеют в общем - то, что они все нуждаются в классе Трансформация(Transform), который описывает преобразование модели из локального в мировое пространство. Помните, мы говорили об этом в последней обучающей программе. Также, вещь, чтобы помнить - то, что большинство объектов, которые Вы хотите отдать является трансформируемыми (Transformables). Значит они имют их собственную внутреннюю матрицу трансформации. Однако, непосредственный режим рендеринга игнорирует всю такую локальную информацию о трансформации и использует только Матрицу Трансформации, поставляемую методу. Это фактически очень удобно, поскольку Вы, например, можете держать один Мешь космического корабля в памяти, но рендерить его много раз с различными Матрицами Трансформации, чтобы показать много различных космических кораблей. Вы будете видеть это в действии, поскольку мы начинаем проектировать движок частиц.

Это все?

"Проблема" рендеринга непосредственного режима состоит в том, что перед рендерингом Вы должны заботиться о большем количестве вещей, потому что Вы не имеете комфорта типа класса Мира, который хранит всю информацию о камере, фоне и освещении. Так, что Вы должны теперь вручную управлять очисткой объектом Фона(Background) буфера viewport, освещением вашей сцены и камерой.

Фон

Чтобы рендерить в непосредственном режиме, мы должны вручную очистить viewport, таким образом готовясь к следующему циклу черчения. Это может быть сделано или до или после цикла рендеринга, но это должно быть сделано после того, как Вы связали(bind) объект Graphics3D и прежде, чем Вы его освободили(released). M3G использует Фон(Background) класс, чтобы помочь Вам с этим. Класс Background содержит много изящной информации типа цвет очистки экрана и какое изображение выводить как фон. Это также очень полезно, поскольку Вы может использовать большое изображение для фона, но показывать часть его, по мере движения вокруг. Например, Вы могли бы иметь большой PNG вашего горизонта, и затем вслед за шагам игрока в игровом мире, Вы можете переместить область показа части вашего Фона, чтобы показать другие части горизонта. Остерегайтесь однако, использовать большой PNGs, он не только очень медленный, но также и очень неэффективно использует память. Самые важные методы класса Background - следующее:

setColor (int ARGB)
setCrop (int cropX, int cropY, int width, int height)
setImageMode (int modeX, int modeY)
setImage (Image2D image)

Давайте рассмотрим их по очереди. Первый метод самый легкий и наиболее используемый. Он устанавливает цвет фона (цвет очистки экрана) в формате 0xAARRGGBB. Например, ярко красный фон будет 0xFFFF0000. Большинство людей устанавливает фон черным, или в цвет неба. По умолчанию, однако, цвет - белый.

setCrop метод очень полезен, если Вы используете изображение для фона. С этим методом, Вы можете решить, что только часть полного изображения фона будет прорендерена. В некоторых специальных случаях, подобно тому, когда часть прямоугольника - вне границ изображения фона, используется третий метод. Он определяет то, что случается с пикселами, которые попадают "вне" исходного изображения. Есть два режима; REPEAT(ПОВТОРИТЬ) и BORDER(ГРАНИЦА). REPEAT означает, что картина повторяется самостоятельно неопределенно, подобно плиточной текстуре, в то время BORDER означает, что пикселы вне исходного изображения окрашиваются цветом фона, переданным в методе setColor. Одна изящная вещь в том, что Вы можете определить поведение режима изображения для x и осей Y отдельно. Это означает, что Вы могли иметь обертку фона изображения вокруг оси X (горизонтально) при статической оси Y (вертикально).

Последний метод, setImage определяет, какое изображение будет использоваться как фон. Вы можете поставлять этому методу пустой указатель для выключения рендеринга изображения фона и

иметь экран только заполненный цветом фона. Это - режим по умолчанию. Обратите внимание, хотя, что изображение обязано быть Image2D, а не стандартным Image из Java™ ME Platform. Image2D не трудно создать из Image, Вы можете использовать очень полезный класс Loader, с указанием прямо изображения в .PNG файле, или использовать конструктор Image2D:

```
Image2D (int format, java.lang.Object image)
```

Это - очень простой конструктор, который сначала берет формат изображения (который в 99 % случаев есть Image2D.RGBA или Image2D. RGB в зависимости от того, используете ли Вы прозрачность или нет) и затем непосредственно изображение. Вторым параметром должен быть ваш класс Image. Таким образом Вы конвертировали стандартный Image из Java™ ME Platform в Image2D:

```
Image img = Image.createImage("/myimage.png");
Image2D img2d = new Image2D(Image2D.RGBA, img);
```

Легко как пирог! Так что Image2D - не нечто запугивающее, а только обертка Image, которая используется M3G системой.

Теперь Вы могли бы задаваться вопросом, как Вы очистите фон, используя класс Background прежде, чем Вы начнете рендерить что - нибудь. Это очень легко и здесь приведен отрывок кода, который все показывает Вам:

```
// Фон
Background back = null;

// Инициализация фона
public void initBackground()
{
   back = new Background();
   back.setColor(0);

}

public void draw(Grahics g)
{
   // Здесь связывается Ваш Graphics3D объект
   //...

   // Это простая очистка экрана
   g3d.clear(back);
}
```

Смотрите, как легко? Это - первая вещь, которой Вы должны управлять при использовании непосредственного режима рендеринга, и теперь мы – на один шаг ближе к нашей цели: двигателю частиц.

Освещение

Другая вещь, которой Вы должны управлять вручную - освещение. Вы должны создать источники света и поместить их в 3D-пространстве с помощью матрицы преобразования. Это все сделается внутри Graphics3D класса следующими методами:

```
addLight ( Light  light, Transform  transform)
setLight (int index, Light  light, Transform  transform)
resetLights ()
```

Они довольно очевидны, но я пробегусь по ним быстро. Вы должны уже знать, как создать Свет в M3G, поскольку мы сделали это в прошлых двух обучающих программах. Первый метод просто добавляет свет к внутреннему массиву источников света, которые должны быть прорендерены. Вы добавляете свет, передавая актуальный класс Light и его Transform.
Матрица трансформации, определяет, где Свет будет прорендерен. addLight метод также возвращает индекс текущего света, который является необходимым, чтобы с помощью индекса в методе setLight позже изменить Свет. Требуется индекс света, чтобы изменить(Graphics3D) новыми Light Светом и Transform. Вызывая setLight с пустым указателем на Light, Вы фактически удалите тот свет из массива. Последний метод - просто метод чистки, который удаляет все источники света из Graphics3D-объекта.

Камера
Вы также должны создать вашу собственную камеру, вместо использования той, что получена из класса World, как мы делали прежде. В этой обучающей программе, мы только создадим Камеру, вызыая ее конструктор по умолчанию. В более поздних частях обучающей программы мы пройдем более продвинутые вещи, которые Вы можете сделать с Камерой, такие как изменение матрицы трансформации. Я не буду прямо сейчас говорить больше об этой теме, вместо этого я только покажу Вам отрывок кода, как это может быть сделано:

```
Camera cam = new Camera();
Graphics3D g3d = Graphics3D.getInstance();
g3d.setCamera(cam, getCameraTransform());
```

Установка камеры очень подобно установке света, так как Вы добавляете вашу Камеру, и матрицу трансформации, которая перемещает камеру в точку в 3D- пространстве. Снова! Будьте осторожны, с этого времени, после добавления, камеры этим режимом, внутренние трансформации узла класса Camera будет игнорироваться. Используются только трансформации из матрицы Трансформации, переданной в методе setCamera.

Подготовка почвы
Теперь Вы знаете о трех вещах, которые мы должны установить вручную, и мы будем готовы прорендерить, что-нибудь в непосредственном режиме. Прежде, чем я покажу Вам, давайте резюмируем необходимые шаги:
1. Мы должны добавить источники света к нашему объекту Graphics3D,обычно это делается, когда сцена инициализируется.
2. Мы должны добавить камеру к объекту Graphics3D. Вы можете сделать это один раз, или в каждом шаге цикла игры, в зависимости от обращения с матрицей Трансформации Камеры.
3. Мы должны очистить фон, так что мы можем рендерить на недавно окрашенный холст.
4. Мы только рендерим наши меши и освобождаем Graphics3D.

Давайте посмотрим как это выглядит в коде:

```
// Получаем контекст Graphics3D
 g3d = Graphics3D.getInstance();

 // Вначале свяжем графический объект. Мы используем предопределенные намеки рендеринга.
 g3d.bindTarget(g, true, RENDERING_HINTS);

 // Очистка фона
 g3d.clear(back);

 // Связываем камеру в фиксированную позицию в начало координат
 g3d.setCamera(cam, identity);

 // Рендерим  сам меш
 g3d.render(someMesh, someMeshTransform);
```

Это немного более сложно, чем рендеринг Мира, который был сделан одним вызовом метода (g3d.render (world);), но в непосредственном режиме Вы получаете намного больше контроля над процессом рендеринга. Теперь, давайте посмотрим, как мы можем использовать непосредственный режим, чтобы фактически сделать кое-что полезное! Система частиц!

Система Частиц (Particle Systems)
Система 3D- частиц обычно состоит из текстуры данных, представляющих частицу и ее физические качества (скорость, жизнь и положение) и системы, которая управляет излучением частиц. Это - очень простая модель, но Вы по желанию можете сделать систему частиц сложной. Так что, давайте сначала создадим наш класс **Particle(**Частица). Чтобы представлять Частицу в 3D- пространстве, мы будем вероятно нуждаться в ее положении в 3D- пространстве, состоящем из координат x, y и z. Мы также нуждаемся в ее скорости, так как мы хотим, чтобы Частица двигалась вокруг в 3D- мире. Мы можем также нуждаться в цвете частицы, чтобы мы могли делать разноцветные частицы. Наконец, мы будем также нуждаться в жизни частицы. Жизнь частицы - то, как долго она остается в 3D-

вселенной прежде, чем уничтожена, или возвращена к жизни в новом положении с новыми скоростями и цветами. Вот - класс Частицы, и он закроет наши основные потребности:

```
/ **
 * Держит всю информацию частицы.
 * Альфа частицы управляет непосредственно ее жизнью. Его альфа - всегда
 * жизнь * 255.
 */

public class Particle
{
   // Жизнь частицы. Изменяется от 1.0f до 0.0f
   private float life = 1.0f;

   // Деградация частицы.
   private float degradation = 0.1f;

   // Скорости частицы.
   private float[] vel = {0.0f, 0.0f, 0.0f};

   // Позиция частицы частицы.
   private float[] pos = {0.0f, 0.0f, 0.0f};

   // Цвет частицы. (RGB формат 0xRRGGBB)
   private int color = 0xffffff;

   /** Пустая инициализация */
   public Particle()
   {

   }

   /**
    * Инициализация частицы.
    * @param velocity Установка скорости
    * @param position Установка позиции
    * @param color Установка цвета (не alpha)
    */
   public Particle(float[] velocity, float[] position, int color)
   {
      setVel(velocity);
      setPos(position);
      this.setColor(color);
   }
 /**
   * @param life Установка жизни.
   */
   void setLife(float life) {
   this.life = life;
   }

 /**
   * @return Возврат жизни.
   */
   float getLife() {
   return life;
   }

 /**
   * @param vel Установка скорости
   */
   void setVel(float[] tvel) {
   System.arraycopy(tvel, 0, vel, 0, vel.length);
```

```
  }

 /**
  * @return Возврат скорости.
  */
  float[] getVel() {
  return vel;
  }
 /**
  * @param pos Установка позиции.
  */
  void setPos(float[] tpos) {
  System.arraycopy(tpos, 0, pos, 0, pos.length);
  }
 /**
  * @return Возврат позиции.
  */
  float[] getPos() {
  return pos;
  }
 /**
  * @param color Установка цвета.
  */
  void setColor(int color) {
  this.color = color;
  }
 /**
  * @return Возврат цвета.
  */
  int getColor() {
  return color;
  }
 /**
  * @param degradation Установка деградации.
  */
  public void setDegradation(float degradation) {
  this.degradation = degradation;
  }
 /**
  * @return Возврат.
  */
  public float getDegradation() {
  return degradation;
  }
  }
```

Так как мы хотим, чтобы наша система частицы была несколько продвинута, мы также сделаем постепенное исчезновение частицы по мере уменьшения их жизни. Это - также очень хороший эффект, и я объясню, как мы сделаем это в M3G позже. Пока, давайте посмотрим на следующую часть системы Частицы, ParticleEffect. Интерфейс ParticleEffect определяет общий интерфейс для всего ParticleEffects. Метод init вызывает Particle, когда она рождается, а метод update(модернизации) вызывают периодически каждый шаг цикла игры, чтобы модернизировать параметры частицы. Наконец метод рендеринга используется чтобы отрендерить Частицу и сделан после метода update.

```
import javax.microedition.m3g.Graphics3D;
/ **
  * Интерфейс, который определяет эффекты для показа,
  * которые производит двигатель частицы покажет.
  * ParticleEffect класс также содержит информацию об используемой bitmap
  * для показа частицы (если любой)
  */
  public interface ParticleEffect
  {
```

```
// Инициализация частицы
public void init(Particle p);

// Модернизация частицы
public void update(Particle p);

// Рендеринг частицы
public void render(Particle p, Graphics3D g3d);
}
```

Теперь, наконец мы нуждаемся в ParticleSystem, который фактически создает Частицы, испускает их и применяет ParticleEffect для них. Я оформил это классом:

```
import javax.microedition.m3g.Graphics3D;
/**
* Управление эмиссией частиц в Вашем 3D мире
*/
public class ParticleSystem
{
// Эффект
private ParticleEffect effect = null;

// Частицы
Particle[] parts = null;

/ **
* Создает систему частиц, которая испускает частицы согласно определенному эффекту.
* @param effect  эффект, который управляет поведением частиц
* @param numParticles число частиц, чтобы испустить
*/
public ParticleSystem(ParticleEffect effect, int numParticles)
{
// Копирование эффекта
setEffect(effect);

// Инициализация частиц
parts = new Particle[numParticles];
for(int i = 0; i < numParticles; i++)
{
parts[i] = new Particle();
effect.init(parts[i]);
}
}

/ ** метод, который делает все. Он вызывается периодически каждый шаг цикла игры */
public void emit(Graphics3D g3d)
{
for(int i = 0; i < parts.length; i++)
{
getEffect().update(parts[i]);
getEffect().render(parts[i], g3d);
}
}
/**
* @param effect установка эффекта.
*/
public void setEffect(ParticleEffect effect) {
this.effect = effect;
}
/**
* @return Возврат эффекта.
*/
public ParticleEffect getEffect() {
return effect;
```

```
  }
  }
```

Как Вы видите, это(класс ParticleSystem) - довольно простой класс, который только создает определенное число частиц, управляет ими через метод init класса ParticleEffect и затем продолжает управлять ими через метод update класса ParticleEffect, когда вызван его(класса ParticleSystem) собственный метод emit. Это - довольно простая, но мощная система частиц и она позволяет нам сделать кое-что подобно следующему:

```
// Мы создаем ParticleEffect класс, который мы написали
ParticleEffect pFx = createParticleEffect();

// Теперь мы создаем ParticleSystem с 20 частицами
ParticleSystem pSys = new ParticleSystem(pFx, 20);

// Где-нибудь в нашем шаге цикла игры...
pSys.emit (g3d);
```

Смотри насколько чисто и просто это? Те строки кода инициализируют и используют нашу новую Систему Частицы. Теперь, прежде, чем я покажу Вам фактический рендеринг и обновление системы частицы, давайте рассмотрим другую тему.

Создание Мешей(Петель) в Коде

Чтобы представлять частицу в 3D- пространстве, лучше всего использовать Мешь, который состоит из простого текстурированного Квадратаю. Квадрат, как Вы помните, - фактически два треугольника, соединенных так, чтобы они представили квадрат. Теперь, вместо того, чтобы создавать Мешь в 3D- студии и экспортировать его в M3G, затем загружать его в нашу программу, мы намного легче и быстрее создадим Мешь в коде. Если Вы помните последнюю обучающую программу, модель состоит из граней, которые непосредственно составлены из 3D-точек или вершин. Так для создания Квадрата мы нуждаемся в четырех точках, одной для каждого угла. В M3G модель описана классом Mesh, который держит все виды информации типа вершин, координат текстуры, граней, режима рендеринга многоугольника и т.д. Мы будем создавать класс Mesh в коде. Для создания класса Mesh нужны три вещи для отображения модели: VertexBuffer(Буфер вершин), IndexBuffer(Буфер индексов) и Appearance(Внешность). Давайте по очереди посмотрим как мы создадим каждый из них.

The VertexBuffer

Этот класс - очень удобный. Это содержит много информации о модели, включая вершины, координатах текстуры, нормалях и цветах. Для нашей очень простой модели, мы будем нуждаться в вершин и координатах текстуры. На сей раз мы не будем использовать нормали или цвета, так как мы не нуждаемся в них. VertexBuffer хранит и информацию о вершинах и координатах текстур в классе названном VertexArray. VertexArray - довольно простой класс, который внутренне хранит значения в массиве. Когда Вы создаете его, Вы определяете, сколько элементов каждая из ваших точек имеет и сколько байтов займет каждый элемент. Теперь Вы могли бы задаваться вопросом, почему я выбираю число элементов? А не 3D- координаты из обычной тройки координат; x, y и z? Хорошо, Вы правильно думаете, 3D- координаты всегда помещаются по трем осям и действительно имеют три элемента. Однако, есть и другие координаты, которые являются также интересными - типа координат текстуры. В этом примере мы будем использовать простую модель координат текстуры, которая только использует пару координат. Теперь, прежде, чем мы фактически начнем создавать наш VertexArrays, давайте посмотрим на координаты. Вот - вершины нашей модели (простой ограниченный план с четырьмя углами).

```
// Вершины плана
 short vertrices[] = new short[] {-1, -1, 0,
 1, -1, 0,
 1, 1, 0,
 -1, 1, 0};
```

Поскольку Вы можете видеть, наш план установлен с углами на x и y-осях плана. Здесь ничего сложного, основные 3D- координаты. Прежде, чем я покажу Вам, координаты текстур, я должен буду рассказать Вам, как координаты текстур используются в M3G. Текстура наносится на многоугольник, по заданным двум координатам. Это - немного сложная тема, к которой мы обратимся в более поздних частях обучающей программы, но я хотел бы сказать Вам немного об этом уже сейчас. Вообразите, что Вы имеете резак для печенья и хотите делать печенье из изображения. Координаты

текстуры фактически говорят Вам о размере вашего резака печенья, означая, что Вы используя координаты текстуры говорите M3G системе о том, какую часть текстуры Вы хотите нанести. Вы можете даже преобразовать координаты текстуры, если Вы хотите - вращать ваш резак печенья. Хотя мы не будем делать этого сегодня. Так, что углы текстуры - (0, 0), (0, 255), (255, 0) и (255, 255), и эти координаты говорят 3D- движку, что мы хотим использовать полную текстуру на нашем многоугольнике. Знание этого – знание того, что мы будем использовать полную текстуру, мы может создать очень простые координаты текстуры.

```
// Координаты текстуры плана
 short texCoords[] = new short[] {0, 255,
 255, 255,
 255, 0,
 0, 0};
```

Хорошо, это не было настолько трудно, не так ли? Теперь все, что мы должны сделать - наполнить классы VertexArray нашими вершинами и координатами текстуры и поместить их в VertexBuffer. Тогда мы на полпути к цели! Давайте посмотрим, как мы делаем это в коде:

```
// Создание вершин модели
 vertexArray = new VertexArray(vertrices.length/3, 3, 2);
 vertexArray.set(0, vertrices.length/3, vertrices);
```

Позвольте мне теперь объяснить, что мы делаем здесь. Прежде всего, мы создаем VertexArray. Конструктор VertexArray берет три вещи; число вершин (или точек, если Вы желаете), которые массив будет содержать, число элементов(прим.координат) на каждую вершину(допустимо 2, 3 или 4) и размер элемента в байтах, который говорит о размере памяти массива для хранения каждого компонента вершины. Различие здесь в том, что, если Вы устанавливаете это значение 2 (2 байта на компонент), то будем использовать короткие целые числа(short integers), чтобы хранить компоненты и если Вы устанавливаете значение 1 (1 байт на компонент), то будем для хранения использовать байты(bytes).
Так, давайте рассмотрим, что мы делаем: сначала мы устанавливаем число вершин, которое является нашей длиной массива вершин, разделенной на три (помните, вершина имеет три компонента). Затем, мы устанавливаем число компонентов, которое является 3 для координат вершин и наконец, мы устанавливаем число байтов(прим. длина компонента). Мы используем здесь два байта для хранения каждого компонента. Помните, что использование большего количества байтов также потребляет большее количество памяти, так если можете используйте один байт для вершин.
Теперь, когда наш VertexArray создан, мы можем установить в него значения. Метод установки реально прост и нуждается в трех аргументах. Первый аргумент - начальный индекс массива вершин. Это - конечно 0, так как мы раньше не помещали никаких вершин в массив. Второй аргумент - число вершин, чтобы копировать - является снова длиной нашего исходного массива, разделенного на три. Третий аргумент – фактический массив для копирования.

Смотри, просто! То же самое подходит для создания VertexArray, который содержит координаты текстуры. Вот - код для этого, и чем вы думаете он отличается от вышеупомянутого примера и почему. Помните то, что я сказал о координатах текстуры.

```
// Создание координат текстур модели
 texArray = new VertexArray(texCoords.length / 2, 2, 2);
 texArray.set(0, texCoords.length / 2, texCoords);
```

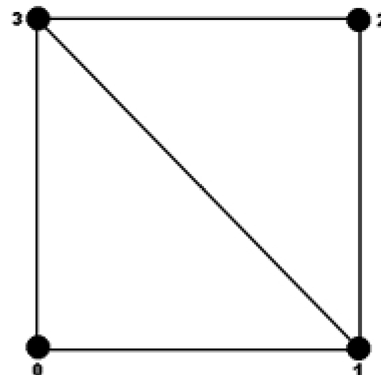
Хорошо, мы сделали все созданием пространственных координат и координат текстуры для нашей модели, и теперь мы должны определить, из каких граней модель состоит. Помните грани из второй обучающей программы? Хорошо, мы должны создать треугольники, из которых наш план будет состоять.

The IndexBuffer

M3G система хранить информацию о гранях(треугольниках) в классе называемом IndexBuffer. Мы говорили о таком классе во второй части обучающей программы, так что Вы должны знать что он делает, но я так или иначе освежу вашу память. IndexBuffer содержит индексы в массиве вершин и описывает грани(треугольники), из которых состоит наша модель. Так как наша модель - простой план (квадрат), то он состоит из двух треугольников. Так как IndexBuffer класс - абстрактный класс, мы должны использовать класс, который наследует IndexBuffer. Этот класс - TriangleStripArray. По его названию Вы можете сказать, он хранит треугольники в массиве, остро а? Есть два режима описать треугольники в TriangleStripArray: явный и неявный. Явное описание - самое общее, Вы

определяете координаты вершины, из которых состоят треугольники. Однако, неявная форма может также может быть очень полезна. Различие - то, что в явной форме Вы должны снабдить массив всеми координатами треугольников, в то время как в неявной форме Вы только снабжаете массив первой вершиной первого треугольника, и TriangleStripArray вычисляет остальные, предполагая, что каждая следующая вершина на одну больше предыдущей.

В этом примере из-за простоты мы будем использовать явную форму. TriangleStripArray требует, чтобы треугольники в массиве были определены специальным путем, который состоит в том, что Вы сначала определяете три индекса для первого треугольника, а затем только доопределяете еще один индекс для каждого последующего треугольника(прим. Два последних индекса предыдущего треугольника являются двумя первыми индексами последующего треугольника). Это - то, что вообще называется полосой треугольников. Вы могли бы задаться вопросом, как это работает? Хорошо, как Вы знаете, складывая треугольники рядом друг с другом в модели (типа нашего простого плана) мы видим, что в результате фактически все треугольники разделяют две точки с другим треугольником (помните наш куб из обучающей программы 2?). Это означает, что мы можем фактически определить треугольник, используя две точки от предыдущего треугольника, и одной дополнительной точки. Вы вероятно помните о наших координатах плана и что мы создали наш план, против часовой стрелки начиная с точки(-1,-1, 0). Вот - треугольники, которые мы хотим создать:



Как Вы видите, мы сокращаем мешь в двух точках: 3 и 1, оставляя нам два треугольника. Эти треугольники - (0, 1, 3) и (1, 3, 2). Вы можете уже видите образец? Первый треугольник фактически разделяет пункты 1 и 3 со вторым треугольником, так что мы можем описать их подобно этому: (0, 1, 3, 2) и TriangleStripArray поймет, что мы имеем два треугольника, которые разделяют средние индексы. Давайте посмотрим, как мы делаем это в коде.
//

```
// Создать индексы граней и длину полосы
int indices[] = new int[] {0, 1, 3, 2};
int[] stripLengths = new int[] {4};

// Создать модели треугольников
triangles = new TriangleStripArray(indices, stripLengths);
```

Так что массив индекса – это то, о чем мы говорили, содержит наши треугольники. Массив stripLengths - кое-что, о чем я не упомянул, но это довольно просто. Он только определяет длину полосы (в индексах). Так как мы определяем два треугольника, наша длина в индексах будет четыре. Это - полезная переменная для определения полос сразличными длинами в одном TriangleStripArray, но мы сегодня не будем беспокоиться об этом. Это - тема более поздних обучающих программ. Только знайте, что сегодня это - четыре. Так, имея все это, мы можем легко создать TriangleStripArray, используя явный конструктор, который берет массив (массив треугольников) и массив stripLengths. Очень легко!

Все, что нам теперь осталось сделать, это создать класс Appearance(Внешность), который в основном говорит M3G системе, как рендерить грани, содержавшиеся в классах IndexBuffer и VertexBuffer.

Appearance(Внешность)
Класс Appearance - большой класс, который содержит много информации о том как рендерить модель. Альфа смешивание(Alpha blending), текстуры, отбор(culling;процесс исключения треугольников из рендеринга) - Вы называете его. Мы будем нуждаться в одном из них, если мы должны создать нашу модель. Сегодня мы будем использовать только несколько из функций.

Сначала давайте начнем с отбора. Отбор – техника используемая для ускорения рендеринга, говорящая системе (прим.M3G), что не надо рендерить некоторые части модели, типа ее или фронтальной стороны. Для движка частиц например, мы действительно никогда не хотим видеть заднюю часть частицы, так что мы можем всегда использовать отбор тыльной стороны на наших частицах. Доступные методы отбора:

```
PolygonMode.CULL_BACK
PolygonMode.CULL_FRONT
PolygonMode.CULL_NONE
```

CULL_BACK определяет отбор тыльной стороны, означает, что задняя часть многоугольника никогда не будет рендериться. CULL_FRONT делает то же самое, но для фронтальных частей многоугольника. Последний отменяет отбор в целом.

Назначения отбора доступны в классе названном PolygonMode, который является компонентом класса Appearance. Так, чтобы устанавить отбор на что - нибудь, используем переменную cullFlags и сделаем следующее:

```
Appearance appearance = new Appearance();
 PolygonMode pm = new PolygonMode();
 pm.setCulling(cullFlags);
 appearance.setPolygonMode(pm);
```

Это было легко? Да. Мы определили форму отбора (обычно это - CULL_BACK). Следующая вещь на очереди- создание текстуры и установка ее в класс Appearance(помните, Внешность также хранит информацию текстуры). Это сделано в очень простой манере, создаем Image2D, который содержит изображение. Мы, ранее в этой обучающей программе, уже говорили о том, как сделать это, так что я не буду снова объяснять. Однако, Image2D сохранен в классе Texture2D, который я объясню.

Текстуры

Для создания текстуры, используемой моделью, Вы нуждаетесь в двух вещах; Image2D, содержащий картинку и Texture2D класс. Texture2D класс держит всю связанную с текстурой информацию, включая данные изображения и преобразование текстуры, которое преобразовывает наш резак печенья, помните? Он даже содержит другие полезные вещи типа смешивание (alpha blending), обертывание (wrapping; если текстура обертывается поперек многоугольника) и фильтрования (filtering; качество проецирования текстуры). Мы будем использовать все вышеперечисленное сегодня, но смешивание будет объяснено немного позже. Давайте начнем шаг за шагом. Сначала мы создаем Image2D и связанный объект Texture2D:

```
// Открытие картинки
 Image texImage = Image.createImage(texFilename);
 Texture2D theTexture = new Texture2D(new Image2D(Image2D.RGBA, texImage));
```

Это было довольно просто, и действительно не требуется никакое объяснение. Вы создаете Image2D и только вручаете это конструктору Texture2D. Теперь давайте поговорим о смешивании.
Вы можете смешать поверхности с цветом вашего многоугольника (да, модели могут иметь цвета) многими различными режимами. Я не буду входить в детали всех различных методов смешивания, вместо этого я прошу, чтобы Вы посмотрели документацию на M3G API и на класс Texture2D. Знайте, что есть различные режимы смешивания вашей текстуры, исходя из ее альфы и цвета, с основным цветом самой модели. Без смешивания вообще, мы сделали бы следующее:

```
// Заменить первоначальные цвета меши (без смешивания)
 theTexture.setBlending(Texture2D.FUNC_REPLACE);
```

Вышеупомянутая часть кода фактически говорит системе о игнорировании все основных цветов модели и очень полезна для многих вещей: так, она накладывая прозрачные части изображения текстуры, делает прозрачными части модели! Хорошо, не так ли? Вы не будете сейчас делать алфа-смешивание текстуры с моделью. Мы смешаем текстуры в этой обучающей программе позже, но пока я хочу, чтобы Вы просто знали о вышеупомянутом методе.
Последние две вещи, которые мы сделаем - обертывание и фильтрование. Это - довольно простые процедуры, и я не буду входить в их детали. Снова, если Вы хотите увидеть, какие различные виды фильтрования и обертывания Вам доступны, посмотрите документацию на M3G API. Здесь, мы не используем никакого обертывания и это означает, что текстура наложена только один раз на многоугольник; мы используем самое простое фильтрование, которое дает самое низкое качество, но

также обеспечивает самую высокую скорость(прим. Скорость рендеринга). Помните, телефоны с M3G - все еще далеки от сегодняшних графических карт на персоналках, так что мы должны жертвовать качеством изображения ради скорости прорисовки.

```
// Установка обертывания и фильтрования
 theTexture.setWrapping(Texture2D.WRAP_CLAMP, Texture2D.WRAP_CLAMP);
 theTexture.setFiltering(Texture2D.FILTER_BASE_LEVEL, Texture2D.FILTER_NEAREST);
```

Теперь все, что осталось, это добавить текстуру к Appearance. Да, я сказал, добавить, а не установить. Это - то, потому что модель может фактически иметь множество текстур! Это - очень хорошая техника для многих вещей подобно мультипликации, наложения света, и т.д. Это - продвинутые обучающие программы, и мы не будем рассматривать их сегодня. Вместо этого я детализирую эти предметы в более поздней части серии. Чтобы добавить текстуру, Вы просто используете метод setTexture (да, глухое название) и передаете индекс Texture2D. Так, чтобы добавить только одну текстуру, Вы делаете:

```
// Добавление текстуры к Appearance
 appearance.setTexture(0, theTexture);
```

Все хорошо, теперь мы все сделали! Мы создали нашу Appearance(внешность) и все другие вещи, жизненно важные для Меши. Все, что осталось, это мы должны фактически составить Мешь из созданных частей.

Мешь

Создание Меши очень просто. Все, что мы должны сделать – поставить созданные ранее части и составить Мешь. Это сделано в единственной строке кода, подобно этому:

```
// Финальное создание Меши
 Mesh mesh = new Mesh(vertexBuffer, triangles, appearance);
```

Вот именно! Теперь мы имеем текстурированный план, который мы можем рендерить в нашу сцену.

Первое впечатление от вышеупомянутого метода создания Меша - он мог показаться Вам очень сложным, но фактически - нет. После неоднократного использования метода, Вы поймете, что он является фактически очень легким и быстрым, но также и очень интуитивным. Теперь мы можем рендерить наш Мешь, используя режим непосредственного рендеринга, о котором я уже говорил, и передать (прим. M3G) матрицу Transform(Преобразование), чтобы разместить Мешь в определенном месте 3D- пространства.
Прежде, чем мы начнем кодировать с использованием класса ParticleEffect, который сделает кое-что прохладное (cull; кульное), давайте сначала поговорим немного о альфа-смешивании текстур и многоугольников.

Альфа-Смешивание

Часто Вы хотите смешать модель и ее текстуру в композит, создавая полупрозрачные модели. Это может использоваться для многих различных вещей, типа окон, водной поверхности, движков частиц, и т.д. Список - фактически бесконечен и только ваше воображение, устанавливает пределы. В M3G этот процесс очень прост, и сегодня я покажу Вам, как сделать это в нашем движке частиц. Мы хотим, чтобы наши частицы из темных становились все более прозрачными, поскольку они прогрессируют в их жизни и полностью исчезали при смерти, так что для этого мы будем нуждаться в альфа-смешивании. Это сделано в двух шагах.

Первый шаг - установка альфа-смешивания в Меше, говорящего о том, насколько он должен исчезнуть(стать прозрачным), делая это на основе логического перемножения(AND) значения альфа и цвета модели. Это сделано в классе VertexBuffer Меши, помните это? Это - метод, названный setDefaultColor и используется для плоской(flat) окраски (у всей модели один и тот же цвет). M3G также поддерживает цвета вершин и гладкую штриховку цвета (прим. интерполяцию цветов между вершинами), что означает, что Вы можете дать различным частям модели различные цвета, но мы не будем сегодня этого делать. Вместо этого, мы только хотим снабдить (прим. VertexBuffer Меша) единственным цветом с альфа- значением для всей модели и смешать с ее средой. Чтобы сделать это, мы только снабжаем метод setDefaultColor цветом в формате 0xAARRGGBB.

```
mesh.getVertexBuffer().setDefaultColor(0x80FF0000);
```

Вышеупомянутая строка кода возьмет VertexBuffer у Меши, установит его полу-прозрачным (значение альфа 0x80, или 128, - полупрозрачность) для ярко красного цвета (0xFF или 255 красный). Если бы Мешь не имела текстуры, то это была бы ярко красная модель, которая является полупрозрачной. Однако, мы также хотим смешать с моделью нашу текстуру. Для чего? Хорошо, если мы смешиваем также и нашу текстуру, мы можем фактически для чего-нибудь изменить цвет текстуры, не желая использовать для этого многослойные текстуры. Для движка частиц это очень полезно, так как мы используем единственную текстуру и только смешиваем, это с цветом по-умолчанию Меша, чтобы получить частицы всех различных цветов. Остро, не так ли? Так, чтобы смешать текстуру мы должны возвратиться к классу Appearance(Внешность), и одному из его признаков - CompositingMode.

CompositingMode говорит классу Appearance как делать композицию(или смесь) модели с ее средой. Есть множество режимов смешать модель с ее средой, и все производят различные результаты на битах (прим. пикселях). Мы будем использовать самый общий и самый простой метод, АЛЬФА-смешивание. Это - то, если бы Вы установили в CompositingMode АЛЬФА- смешивание и добавили бы это к Appearance(Внешности) Меши.

```
CompositingMode cm = new CompositingMode();
 cm.setBlending(CompositingMode.ALPHA);
 m.getAppearance(0).setCompositingMode(cm);
```

Смотрите, как было просто? Если Вы хотите знать о других режимах смешивания (ALPHA_ADD, MODULATE, MODULATE_X2 and REPLACE), проверьте документацию на класс CompositingMode M3G API.

Мы почти установили настройки нашего смешивания. Все, что осталось, это изменить текстуру. Помните, когда мы создавали текстуру и устанавливали режим смешивания FUNC_REPLACE, который полностью игнорировал цвет основной Меши и прозрачность? Хорошо, мы не можем использовать этот режим, чтобы смешать нашу текстуру с нашим Мешем, так что режим должен быть изменен. Есть три режима смешивания текстуры с основной моделью, и они - FUNC_REPLACE (который мы уже видели), FUNC_BLEND (что фактически смешивает), FUNC_DECAL и FUNC_MODULATE, которые являются специальными режимами, о которых Вы будете должны прочитать в документации API или подождать более поздней части этой серии обучающей программы. Сегодня мы будем использовать FUNC_BLEND. Он просто смешает нашу текстуру (и альфа-значения текстуры) с цветами и альфа-значениями модели. Чтобы сделать это, мы должны восстановить текстуру меши (или текстуры, если их много) и установить правильное смешивание. Чтобы наложить первую текстуру модели (которую мы сегодня сделаем), мы сделаем типа этого:

```
m.getAppearance(0).getTexture(0).setBlending(textureBlending);
```

Смотрите, как это просто! Теперь мы настроили смешивание для нашей модели и можем показать ее в разнообразии цветов и прозрачности. Прежде, чем мы сделаем что - нибудь, давайте поговорим о виде изображения текстуры, в которой мы нуждаемся для смешивания.

Смешивание Изображения

При смешивании изображения текстуры с основной моделью, Вы не можете использовать любые виды текстуры. Хорошо, фактически Вы можете, но Вы не получите желаемые результаты. Например; если текстура полностью белая, нет режима смешивания с текстурой, так как мы используем алгоритм, который складывает цвета модели и цвета текстуры. Это - проблема для белых текстур, так как белый - самый высокий цвет (255) и независимо от того сколько Вы добавляете к нему, Вы не будете иметь возможности произвести почти белый. Для лучшего смешивания и полного цветного управления с цветом по умолчанию Меша, используйте полностью черную текстуру. Этим путем Вы можете установить любой цвет, который Вы хотите у цвета по умолчанию Меша, и текстура также имеет тот же самый цвет.

Вот - картина текстуры, которую мы будем использовать в игре. Как Вы можете видеть, это полностью черное расплывающееся пятно, для создания хорошего и гладкого эффекта. Если Вы хотите, можете не использовать черный, но только помните, что цвет текстуры всегда смешивается с цветом Меши, так что Вы могли бы вывести полностью различный цвет, чем Вы имеете в виду.

Управление Альфой
Теперь, чтобы фактически управлять альфой (прозрачностью)модели и текстуры, Вы только управляете цветом по умолчанию Меша (который находится в VertexBuffer Меша, помните?). Ниже Уменьшая Альфа-значение, находящееся в цвете по умолчанию, Мешь будет более прозрачным. Чтобы построить цвет с определенным альфа-значением, Вы могли бы сделать это:

```
int color = (alpha << 24) | (red << 16) | (green << 8) | blue;
```

Вышеупомянутая часть кода комбинирует альфа-компонент с красным, зеленым и синим цветами и создаст тип цвета, который VertexBuffer ожидает в его методе setDefaultColor. Так, изменяя альфа-переменную, Вы также изменяете прозрачность. Позже Вы увидите практический пример этого.

Окончание Системы Частиц
Теперь, когда мы можем фактически создать Мешь из кода, давайте создадим сервисную функцию, которая сделает это для нас. Сначала давайте сделаем кое-что, что будет делать любую Мешь смешиваемой. Это выглядит подобно этому:

```
/ ** Установение альфа-смешивания Меша.
* Только означает, если мешь уже – альфа-смешена
*/
public static void setMeshAlpha(Mesh m, int alpha)
{
 m.getVertexBuffer().setDefaultColor(alpha);
 }

 /**
 *
 * @param m Мешь для конвертирования в смешиваемый
 * @param alpha Альфа цвет для  смешивания
 * @param textureBlending Параметр смешивания текстуры.
 */
 public static void convertToBlended(Mesh m, int alpha, int textureBlending)
 {
 // Установка альфа
 setMeshAlpha(m, alpha);

 // Установка режима смешивания
 CompositingMode cm = new CompositingMode();
 cm.setBlending(CompositingMode.ALPHA);
 m.getAppearance(0).setCompositingMode(cm);
 m.getAppearance(0).getTexture(0).setBlending(textureBlending);
 }
```

Эти два метода могут конвертировать любой Мешь, чтобы быть смешивающейся с остальной частью нашей сцены. Это всегда хорошо иметь. Теперь давайте получим метод, который фактически делает процесс создания:

```
/**
 * Создание плана текстуры.
 * @param texFilename The name of the texture image file
 * @param cullFlags The flags for culling. See PolygonMode.
 * @return The finished textured mesh
 */
 public static Mesh createPlane(String texFilename, int cullFlags)
 {
 // Вершины плана
 short vertrices[] = new short[] {-1, -1, 0,
 1, -1, 0,
 1, 1, 0,
 -1, 1, 0};
```

```
// координаты текстуры плана
  short texCoords[] = new short[] {0, 255,
  255, 255,
  255, 0,
  0, 0};

  // Классы
  VertexArray vertexArray, texArray;
  IndexBuffer triangles;
// Создание вершин модели
  vertexArray = new VertexArray(vertrices.length/3, 3, 2);
  vertexArray.set(0, vertrices.length/3, vertrices);

  // Создание координат текстуры модели
  texArray = new VertexArray(texCoords.length / 2, 2, 2);
  texArray.set(0, texCoords.length / 2, texCoords);

  // Смешивание предыдущих вершин  из VertexBuffer и координат текстуры
  VertexBuffer vertexBuffer = new VertexBuffer();
  vertexBuffer.setPositions(vertexArray, 1.0f, null);
  vertexBuffer.setTexCoords(0, texArray, 1.0f/255.0f, null);

  // создание индексов и длины поверхности
  int indices[] = new int[] {0, 1, 3, 2};
  int[] stripLengths = new int[] {4};

  // Создание полосы треугольников модели
  triangles = new TriangleStripArray(indices, stripLengths);
// Создание Внешности
  Appearance appearance = new Appearance();
  PolygonMode pm = new PolygonMode();
  pm.setCulling(cullFlags);
  appearance.setPolygonMode(pm);
// Создание и установка текстуры
  try
  {
  // Открытие файла изображения
  Image texImage = Image.createImage(texFilename);
  Texture2D theTexture = new Texture2D(new Image2D(Image2D.RGBA, texImage));

  // Замена оригинального цвета Меша (без смешивания)
  theTexture.setBlending(Texture2D.FUNC_REPLACE);

  // Установка обертывания и фильтрации
  theTexture.setWrapping(Texture2D.WRAP_CLAMP, Texture2D.WRAP_CLAMP);
  theTexture.setFiltering(Texture2D.FILTER_BASE_LEVEL, Texture2D.FILTER_NEAREST);
// Добавить текстуру к Внешности
  appearance.setTexture(0, theTexture);
}
  catch(Exception e)
  {
  // Кое-что пошло не так, как надо
  System.out.println("Failed to create texture");
  System.out.println(e);
  }

  // Наконец создаем Мешь
  Mesh mesh = new Mesh(vertexBuffer, triangles, appearance);
// All done
  return mesh;
  }
```

Большой код, но он легкий. Проверьте его, вернитесь к части обучающей программы, где я объяснил код по созданию Меша, и Вы поймете как все это просто.

Теперь, когда мы можем создавать Меши, мы должны создать класс ParticleEffect, чтобы использовать его с нашим ParticleSystem и получать некоторый эффект. Я стремлюсь к эффекту фонтана, который Вы можете вращать вокруг экрана столько, сколько Вы хотите. Хорошим упражнением после этой обучающей программы должно было бы создание вашего собственного класса ParticleEffect, который делает некоторый "кульный" эффект. Пробуйте создать круглый взрыв (подобно фейерверку) или возможно даже кое-что продвинутое, как мерцающее пламя. Во всяком случае, сначала мы будем использовать некоторое хорошее объектно-ориентируемое программирование для абстрактного поведения, которое много будут иметь ParticleEffects; использование плоского Меша как частицы. Этот класс выглядит:

```
/**
 * Представляет эффект частицы, что использует bitmap.
 */
 public abstract class BitmapParticleEffect implements ParticleEffect
 {
 // Мешь
 Mesh mesh = null;

 // Матрица трансформации
 Transform trans = new Transform();

 // Масштабирование
 float scale = 1.0f;

 /** Инициализация bitmap, используемой для рендеринга частиц */
 public BitmapParticleEffect(String filename, float scale)
 {
 // Загрузка плана с желаемой текстурой
 mesh = MeshFactory.createAlphaPlane(filename, PolygonMode.CULL_BACK, 0xffffffff);

 // Устанавливаем масштаб
 this.scale = scale;
 }

 /**
 * @смотри ParticleEffect#render(Particle, Graphics3D)
 */
 public void render(Particle p, Graphics3D g3d)
 {
 // Вычисление альфа
 int alpha = (int)(255 * p.getLife());

 // Создание цвета
 int color = p.getColor() | (alpha << 24);

 // Установка альфа
 MeshOperator.setMeshAlpha(mesh, color);

 // Трансформация
 trans.setIdentity();
 trans.postScale(scale, scale, scale);
 float[] pos = p.getPos();
 trans.postTranslate(pos[0], pos[1], pos[2]);

 // Рендеринг
 g3d.render(mesh, trans);
 }
}
```

Как Вы видите, это - абстрактный класс, который уже много для нас устанавливает, типа - непосредственный режим рендеринга. Проверьте метод рендеринга, и Вы увидите то, о чем я говорил ранее в этой обучающей программе. Как Вы видите, мы также устанавливаем альфу Меши исходя из жизни частицы. Так как каждая частица уже имеет цвет в формате 0xRRGGBB все, что мы делаем - вставляем значение альфы и устанавливает это как цвет Меша по - умолчанию. Метод

MeshFactory.createAlphaPlane, который используется в конструкторе - очень прост. Он сначала вызывает наш метод createPlane, который я ранее показал, и затем использует наши сервисные функции, чтобы делать Мешь смешиваемым.

Вышеупомянутый класс делает много работы для нас, так что все, что мы должны сделать теперь - фактически сделать круговое движение частиц. Я хотел создавать фонтан и вот - результат:

```
/**
 * Создает хороший эффект фонтана для частиц, который стреляет частицами
 * в некотором направлении, определенном его углом.
 *Угол может быть изменен в реальном времени.
 */
 public class FountainEffect extends BitmapParticleEffect
 {
 // Угол эмиссии частицы
 частный int угол = 90;

 // Синус и косинус текущего угла
 private int angle = 90;

 // Синус и косинус текущего угла
 private float[] trig = {1.0f, 0.0f};

 // Точка испускания
 private float[] pos = {0.0f, 0.0f, 0.0f};

 // Случайность
 Random rand = null;

 /**
 * @param angle Угол эмиссии частицы
 */
 public FountainEffect(int angle)
 {
 // Инициализация bitmap
 super("/res/particle.png", 0.05f);

 // Установить угол
  setAngle(angle);

 // Получить случайное число
 rand = new Random();
 }
 /**
   * @смотри ParticleEffect#init(Particle)
   */
   public void init(Particle p)
   {
   // Установка жизни частицы
   p.setLife(1.0f);

   // Установка позиции частицы
   p.setPos(pos);

   // Создание скорости(velocities) частицы
   float[] vel = new float[3];

   // Мы хотим скорости от 0.2f до 1.0f
   float xyvel = rand.nextFloat() * 0.8f + 0.2f;

   // Мы хотим, чтобы частица умерла медленно
   p.setDegradation(xyvel / 18);

   // Установка скорости согласно тригонометрии с маленьким отклонением(deviation)
```

```
vel[0] = xyvel * trig[1] + rand.nextFloat() * 0.125f - 0.0625f;
vel[1] = xyvel * trig[0] + rand.nextFloat() * 0.125f - 0.0625f;

// Никакого движения в глубину
vel[2] = 0.0f;

// Установка скорости
p.setVel(vel);

// Установка случайного цвета
int r = (int)(120 * rand.nextFloat()) + 135;
int g = (int)(120 * rand.nextFloat()) + 135;
int b = (int)(120 * rand.nextFloat()) + 135;
int col = (r << 16) | (g << 8) | b;
p.setColor(col);
}
/**
* @see ParticleEffect#update(Particle)
*/
public void update(Particle p)
{
// Просто обновление позиции
float[] ppos = p.getPos();
float[] vel = p.getVel();
ppos[0] += vel[0];
ppos[1] += vel[1];
ppos[2] += vel[2];

// Обновление жизни
p.setLife(p.getLife() - p.getDegradation());

// Проверка жизни. Если смерть, мы только переинициализируем
if(p.getLife() < -0.001f)
{
init(p);
}
}

/**
* @param angle Установка угла.
*/
public void setAngle(int angle) {
this.angle = angle;
trig[0] = (float)Math.sin(Math.toRadians(angle));
trig[1] = (float)Math.cos(Math.toRadians(angle));
}
/**
* @return Возврат угла.
*/
public int getAngle() {
return angle;
}
/**
* @param pos Установка позиции.
*/
void setEmittingOrigin(float[] pos) {
this.pos = pos;
}
/**
* @return Возврат позиции.
*/
float[] getEmittingOrigin() {
return pos;
```

```
   }
}
```

Я не желаю детализировать это, так как это - действительно основной материал тригонометрии. Вы можете выбросить тот класс и заменить это собственным классом, который делает некоторый “кульный” эффект частиц. Все, что это делает - рандомизирует скорость частицы в зависимости от угла фонтана, и затем перемещает ее вперед, пока она не умрет. Вы можете изменить угол в реальном времени, чтобы иметь возможность вращать фонтан. Я добавил это к главному шагу цикла игры, и это выглядит так:

```
// Проверка управления для вращения фонтана
 if(key[LEFT])
 fx.setAngle(fx.getAngle() + 5);
 if(key[RIGHT])
 fx.setAngle(fx.getAngle() - 5);
```

Объект fx - фактически ссылка на FountainEffect, в котором мы изменяем угол. Если нажата клавиша “Налево”, то увеличиваем угол против часовой стрелки, а если нажата клавиша “Направо” -уменьшаем угол по часовой стрелке.

Настраивание Сцены для Режима Непосредственного рендеринга

Теперь, когда мы все сделали, я только покажу Вам, как я настроил режим непосредственного рендеринга. Вы уже должны знать это, так как я ранее прошел это в этой части обучающей программы, так что я только повторю информацию. Здесь -как мы строим холст:

```
/** Конструирование холста
 */
 public M3GCanvas(int fps)
 {
 // Мы не желаем перехватывать нажатия клавиш обычным способом
 super(true);

 // Мы желаем полноэкранный холст
 setFullScreenMode(true);

 // Загрузка камеры
 loadCamera();

 // Загрузить наш фон
 loadBackground();

 // Настройка 3D- графики
 setUp();
 }
```

Метод loadCamera,очень прост,- создает камеру, которую будет использовать наша 3D- система. Все, что он делает - создает камеру по умолчанию. Вы можете видеть то, что он делает ниже в листинге кода. Метод loadBackground создает фон и устанавливает его цвет в черный (0x0). Также очень простой. Последний метод, setup, заканчивает инициализацию, фактически добавляя окружающий (ambient) свет к нашему контексту рендеринга. Это выглядит так:

```
/ ** Готовит движок Graphics3D к непосредственному режиму рендеринга, добавляя свет */
 private void setUp()
 {
 // Получить инстанцию
 g3d = Graphics3D.getInstance ();

 // Добавить свет к нашей сцене, чтобы мы могли видеть что-нибудь
  g3d.addLight(createAmbientLight(), identity);
 }


 /** Создает простой окружающий(ambient) свет */
 private Light createAmbientLight()
```

```
{
Light l = new Light();
l.setMode(Light.AMBIENT);
l.setIntensity(1.0f);
return l;
}
```

Там действительно ничего нового. Мы создавали окружающие огни прежде много раз, и Вы должны знать это наизусть.
Так, теперь наша сцена готова к рендерингу и все, что мы должны сделать - испускать некоторые частицы на экран. Как я уже упоминал, это делается, объекта ParticleSystem. Мы создаем ParticleSystem, добавляем к нему ParticleEffect (наш FountainEffect), и только затем мы вызываем метод emit в каждой итерации цикла. Это - то, что я поместил в главный цикл игры:

```
// Окутать все пробующим/ловящим блоком на всякий случай
try
{
// Получить Graphics3D контекст
g3d = Graphics3D.getInstance();

// Сначала свяжите графический объект.
// Мы используем наши предопределенные намеки рендеринга.
g3d.bindTarget(g, true, RENDERING_HINTS);

// Очистка фона
g3d.clear(back);

// Связать камеру с фиксированной позицией в origo
g3d.setCamera(cam, identity);

// Инициализация частиц
if(ps == null)
{
fx = new FountainEffect(90);
ps = new ParticleSystem(fx, 20);
}

// Эмиссия частиц
ps.emit(g3d);

// Проверка управления для вращения фонтана
if(key[LEFT])
fx.setAngle(fx.getAngle() + 5);
if(key[RIGHT])
fx.setAngle(fx.getAngle() - 5);

// Выход, если пользователь нажимает огонь(fire)
if(key[FIRE])
TutorialMidlet.die();
}
catch(Exception e)
{
reportException(e);
}
finally
{
// Всегда не забудьте отвязать!
g3d.releaseTarget();
}
```

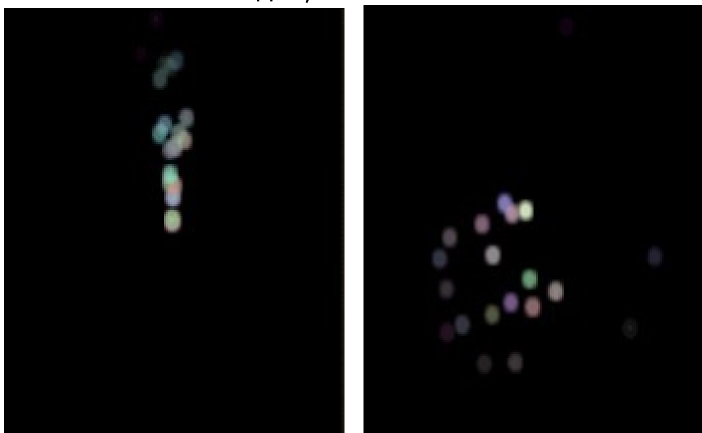
Чтобы понять это, давайте разобьем этот код на части. Вначале базовые операции, мы получаем инстанцию объекта Graphics3D и связываем ее с нашим объектом Graphics (вышеупомянут метод draw(Graphics g), так что мы уже имеем объект Graphics) . После этого, мы очищаем фон и буфер глубины(depth), вызывая g3d.clear на нашем Background объекте. Это дает нам чистый, черный холст. Теперь мы должны установить нашу камеру в мир с его собственной, трансформацией (неуклюже делать это в каждом повторении цикла игры, но я делаю это здесь для ясности). Мы

будем всегда использовать identity(идентичность), трансформацию, так как мы не хотим перемещать камеру в этой обучающей программе. Затем, мы проверяем, инициализирована ли ранее наша система частиц, и если необходимо , то инициализируем ее. Мы только создаем хороший эффект фонтана, который начинается под углом 90 градусов(указывающий прямо вверх) и систему частиц, которая содержит 20 частиц. После того, как это сделано, мы вызываем emit(испускающий) метод, и частицы будут обновлены и прорендерены в наш мир. Когда все сделано, мы проверяем некоторые клавиши, типа клавиши джойстика для вращения нашей хорошей системы частиц, и клавишу Fire для выхода из приложения.

Все! Это было не так трудно? Как я сказал, для практики, Вы можете фактически заменить FountainEffect класс кое-чем, что Вы написали сами, получив свой эффект частиц. Вы можете также попробовать загрузить много различных Мешей в память с большим количеством различных текстур, так как теперь Вы имеете методы, которые делают эту работу для Вас.

Заключение

Так это все выглядит, вот - несколько screenshots кода в действии. Довольно, симпатичные цвета!



Есть наша система частиц! Обратите внимание, как прекрасно это, когда частицы уже исчезают, старые еще добираются. Я также решил рандомизировать цвета частиц от 128 до 255, так, чтобы мы не получили никаких темных и капризных цветов. Мы только хотим теплое и яркое счастье! Вот – листинг кода. Вы можете даже загрузить полный исходный пакет, включая ресурсы по ссылке внизу документа.

TutorialMidlet

```
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
public class TutorialMidlet extends MIDlet implements CommandListener
{
    // Переменная, которая держит уникальный display
 private Display display = null;

 // Холст
 private M3GCanvas canvas = null;

 // Ссылка в MIDlet-е на себя самого
 private static MIDlet self = null;
 / ** Вызывается, когда приложение стартует, и когда возобновлено.
  * Мы игнорируем резюме(resume) здесь и помещаем данные для нашего
  *(прим. лучше бы в классе холста)
  * в startApp методе. Это - вообще то очень плохая практика.
  */
 protected void startApp() throws MIDletStateChangeException
 {
```

```
    // Выделение
  display = Display.getDisplay(this);
  canvas = new M3GCanvas(30);

  // Добавить к холсту команду выхода
  // Эта команда не будет видна так  как мы
  // работаем в полноэкранном режиме,
  // но всегда хорошо иметь команду выхода
  canvas.addCommand(new Command("Quit", Command.EXIT, 1));
   // Установка листенера мидлета
  canvas.setCommandListener(this);

  // Старт холста
  canvas.start();
  display.setCurrent(canvas);

 // Установка ссылки на самого себя
  self = this;
 }

 /** Вызывается, когда игра должна делать паузу */
 protected void pauseApp()
 {

 }
 /** Вызывается когда приложение должно быть закрыто */
 protected void destroyApp(boolean unconditional) throws MIDletStateChangeException
 {
    // Метод закрывает MIDlet
  notifyDestroyed();
 }
 /** Слушает команды и обрабатывает */
    public void commandAction(Command c, Displayable d) {
        // Если мы получаем команду  EXIT(ВЫХОД), мы уничтожаем приложение
        if(c.getCommandType() == Command.EXIT)
            notifyDestroyed();
    }

   / ** Статический метод, который выходит из приложения
     * используя статическую переменную  ' self' */
      public static void die()
    {
        self.notifyDestroyed();
    }
}
```

M3GCanvas

```
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.game.GameCanvas;
import javax.microedition.m3g.Background;
import javax.microedition.m3g.Camera;
import javax.microedition.m3g.Graphics3D;
import javax.microedition.m3g.Light;
import javax.microedition.m3g.Transform;
public class M3GCanvas  extends GameCanvas  implements Runnable {

    // управление нитью(Thread-control)
    boolean running = false;
    boolean done = true;

    // Если игра закончилась
    public static boolean gameOver = false;

    // Намеки рендеринга
```

```
    public static final int STRONG_RENDERING_HINTS = Graphics3D.ANTIALIAS |
Graphics3D.TRUE_COLOR | Graphics3D.DITHER;
    public static final int WEAK_RENDERING_HINTS = 0;
    public static int RENDERING_HINTS = STRONG_RENDERING_HINTS;

    // Массив клавиш
    boolean[] key = new boolean[5];

    // Константы клавиш
    public static final int FIRE = 0;
    public static final int UP = FIRE + 1;
    public static final int DOWN = UP + 1;
    public static final int LEFT = DOWN + 1;
    public static final int RIGHT = LEFT + 1;

    // Глобальная матрица идентичности
    Transform identity = new Transform();

    // Глобальный объект Graphics3D
    Graphics3D g3d = null;

    // Глобальный объект камеры
    Camera cam = null;

    // Система частиц
    ParticleSystem ps = null;
    FountainEffect fx = null;

       /** Конструктор холста
     */
    public M3GCanvas(int fps)
    {
        // Мы не хотим захватить клавиши обычным путем
        super(true);

        // Мы желаем полноэкранный холст
        setFullScreenMode(true);

        // Загружаем камеру
        loadCamera();

        // Загружаем фон
        loadBackground();

        // Устанавливаем 3D-графику
        setUp();
    }

/ ** Готовит движок Graphics3D к непосредственному режиму рендеринга, добавляя свет */
 private void setUp()
 {
 // Получить инстанцию
 g3d = Graphics3D.getInstance ();

 // Добавить свет к нашей сцене, чтобы мы могли видеть что-нибудь
  g3d.addLight(createAmbientLight(), identity);
 }


 /** Создает простой окружающий(ambient) свет */
 private Light createAmbientLight()
 {
 Light l = new Light();
 l.setMode(Light.AMBIENT);
```

```
l.setIntensity(1.0f);
return l;
}




  / ** Когда установлен полноэкранный режим,
    * некоторые устройства будут вызывать
    * этот метод уведомляя нас о новой ширине/высоте.
    * Однако, мы в этой обучающей программе
    * действительно не заботимся о ширине/высоте
    * так что мы позволяем этому быть
    */
   public void sizeChanged(int newWidth, int newHeight)
   {

   }



   /** Загрузка камеры */
    private void loadCamera()
   {
      // Создание новой камеры
      cam = new Camera();
   }

   /** Загрузка фона */
   private void loadBackground()
   {
      // Создание нового фона и установка его цвет черным
      back = new Background();
      back.setColor(0);
   }
   /** Чертим на экране
    */
   private void draw(Graphics g)
   {
     // Окутать все пробующим/ловящим блоком на всякий случай
try
{
// Получить Graphics3D контекст
g3d = Graphics3D.getInstance();

// Сначала свяжите графический объект.
// Мы используем наши предопределенные намеки рендеринга.
g3d.bindTarget(g, true, RENDERING_HINTS);

// Очистка фона
g3d.clear(back);

// Связать камеру с фиксированной позицией в origo
g3d.setCamera(cam, identity);

// Инициализация частиц
if(ps == null)
{
fx = new FountainEffect(90);
ps = new ParticleSystem(fx, 20);
}

// Эмиссия частиц
```

```
ps.emit(g3d);

// Проверка управления для вращения фонтана
if(key[LEFT])
fx.setAngle(fx.getAngle() + 5);
if(key[RIGHT])
fx.setAngle(fx.getAngle() - 5);

// Выход, если пользователь нажимает огонь(fire)
if(key[FIRE])
TutorialMidlet.die();
}
catch(Exception e)
{
reportException(e);
}
finally
{
// Всегда не забудьте отвязать!
g3d.releaseTarget();
}

/** Стартует холст, разжигая нить(thread)
   */
  public void start() {
     Thread myThread = new Thread(this);

     // Сделайте чтобы  мы знали, что мы запущены
     running = true;
     done = false;

     // Старт
     myThread.start();
  }

  /** Управляемый, управляется целой нитью. Также сохраняет постоянный FPS
   */
  public void run() {
     while(running) {
        try {
           // Вызываем метод process(вычисляем клавиши)
           process();

           // Чертим все
           draw(getGraphics());
           flushGraphics();

           // Спим для предотвращения starvation
           try{ Thread.sleep(30); } catch(Exception e) {}
        }
        catch(Exception e) {
           reportException(e);
        }
     }

     // Уведомление о завершении
     done = true;
  }

  /**
   * @param e
   */
  private void reportException(Exception e) {
     System.out.println(e.getMessage());
```

```
      System.out.println(e);
      e.printStackTrace();
   }
   /** Пауза в игре
    */
   public void pause() {}

   /** Останавливает игру
    */
   public void stop() { running = false; }

   /** Процесс обработки клавиш     */
   protected void process()
   {
      int keys = getKeyStates();

      if((keys & GameCanvas.FIRE_PRESSED) != 0)
         key[FIRE] = true;
      else
         key[FIRE] = false;

      if((keys & GameCanvas.UP_PRESSED) != 0)
         key[UP] = true;
      else
         key[UP] = false;

      if((keys & GameCanvas.DOWN_PRESSED) != 0)
         key[DOWN] = true;
      else
         key[DOWN] = false;

      if((keys & GameCanvas.LEFT_PRESSED) != 0)
         key[LEFT] = true;
      else
         key[LEFT] = false;

      if((keys & GameCanvas.RIGHT_PRESSED) != 0)
         key[RIGHT] = true;
      else
         key[RIGHT] = false;
   }
     /** Проверка запуска Нити
    */
   public boolean isRunning() { return running; }

   /** Проверка комплектности выполнения если нить финишировала
    */
   public boolean isDone() { return done; }
}
```

MeshFactory

```
import javax.microedition.lcdui.Image;
import javax.microedition.m3g.Appearance;
import javax.microedition.m3g.Image2D;
import javax.microedition.m3g.IndexBuffer;
import javax.microedition.m3g.Mesh;
import javax.microedition.m3g.PolygonMode;
import javax.microedition.m3g.Texture2D;
import javax.microedition.m3g.TriangleStripArray;
import javax.microedition.m3g.VertexArray;
import javax.microedition.m3g.VertexBuffer;
/**
 * Статический класс, который управляет созданием сгенерированных кодом Мешей
```

```
 */
public class MeshFactory
{
   /** Создает план текстуры, который альфа-смешан
    *
    * @param texFilename Имя файла изображения текстуры
    * @param cullFlags Флаги для куллинга. Смотри PolygonMode.
    * @param alpha Альфа –значение для смешивания. Формат полного цвета - 0xAARRGGBB
    * @return Законченный текстурированный мешь
    */
   public static Mesh createAlphaPlane(String texFilename, int cullFlags, int alpha)
   {
      // Создать нормали меша
      Mesh mesh = createPlane(texFilename, cullFlags);

      // Сделать его смешивающимся
      MeshOperator.convertToBlended(mesh, alpha, Texture2D.FUNC_BLEND);
      return mesh;
   }

   /**
    * Создает план текстуры
    * @param texFilename Имя файла изображения текстуры
    * @param cullFlags Флаги для куллинга. Смотри PolygonMode.
    * @return Законченный текстурированный мешь     */
       public static Mesh createPlane(String texFilename, int cullFlags)
       {
          // координаты плана
        short vertrices[] = new short[] {-1, -1, 0,
                                 1, -1, 0,
                                 1, 1, 0,
                                 -1, 1, 0};
      // Координаты текстуры плана
      short texCoords[] = new short[] {0, 255,
                                255, 255,
                                255, 0,
                                0, 0};

      // Классы
      VertexArray vertexArray, texArray;
      IndexBuffer triangles;
      // Создание  вершин модели
      vertexArray = new VertexArray(vertrices.length/3, 3, 2);
      vertexArray.set(0, vertrices.length/3, vertrices);

      // Создание  Координат текстуры модели
      texArray = new VertexArray(texCoords.length / 2, 2, 2);
      texArray.set(0, texCoords.length / 2, texCoords);

      // Смешивание предыдущих вершин  из VertexBuffer и координат текстуры
      VertexBuffer vertexBuffer = new VertexBuffer();
      vertexBuffer.setPositions(vertexArray, 1.0f, null);
      vertexBuffer.setTexCoords(0, texArray, 1.0f/255.0f, null);

      // Создание индексов и длины полосы
      int indices[] = new int[] {0, 1, 3, 2};
      int[] stripLengths = new int[] {4};

      // Создание полосы треугольников модели
      triangles = new TriangleStripArray(indices, stripLengths);
      // Создание Внешности
      Appearance appearance = new Appearance();
      PolygonMode pm = new PolygonMode();
```

```
        pm.setCulling(cullFlags);
        appearance.setPolygonMode(pm);
        // Создание и установка текстуры
        try
        {
           // Открытие файла изображения
           Image texImage = Image.createImage(texFilename);
           Texture2D theTexture = new Texture2D(new Image2D(Image2D.RGBA, texImage));

           // Замена оригинального цвета Меша (без смешивания)
           theTexture.setBlending(Texture2D.FUNC_REPLACE);

           // Установка обертывания и фильтрации
           theTexture.setWrapping(Texture2D.WRAP_CLAMP, Texture2D.WRAP_CLAMP);
           theTexture.setFiltering(Texture2D.FILTER_BASE_LEVEL, Texture2D.FILTER_NEAREST);
           // Добавить текстуру к Внешности
           appearance.setTexture(0, theTexture);
        }
        catch(Exception e)
        {
           // Кое-что пошло не так, как надо
           System.out.println("Failed to create texture");
           System.out.println(e);
        }

        // Наконец создаем Мешь
        Mesh mesh = new Mesh(vertexBuffer, triangles, appearance);
        // All done
        return mesh;
    }
}
```

MeshOperator

```
import javax.microedition.m3g.CompositingMode;
import javax.microedition.m3g.Mesh;
 /**
  * Исполняет некоторые основные действия на Мешь- объектах */

public class MeshOperator
{
  / ** Устанавливает альфу у смешиваемой меши.
 *  Значима только когда мешь уже - альфа смешиваема    */
    public static void setMeshAlpha(Mesh m, int alpha)
    {
        m.getVertexBuffer().setDefaultColor(alpha);
    }

   /**
   *
   * @param m Мешь для конвертирования в смешиваемый
   * @param alpha Альфа цвет для  смешивания
   * @param textureBlending Параметр смешивания текстуры.
   */
    public static void convertToBlended(Mesh m, int alpha, int textureBlending)
    {
        // Установка альфа
        setMeshAlpha(m, alpha);

        // Установка режима смешивания
        CompositingMode cm = new CompositingMode();
        cm.setBlending(CompositingMode.ALPHA);
        m.getAppearance(0).setCompositingMode(cm);
        m.getAppearance(0).getTexture(0).setBlending(textureBlending);
    }
```

```
}
```

Particle

```
/ **
 * Держит всю информацию частицы.
 * Альфа частицы управляет непосредственно ее жизнью. Его альфа - всегда
 * жизнь * 255.
 */

public class Particle
{
    // Жизнь частицы. Изменяется от 1.0f до 0.0f
    private float life = 1.0f;

    // Деградация частицы.
    private float degradation = 0.1f;

    // Скорости частицы.
    private float[] vel = {0.0f, 0.0f, 0.0f};

    // Позиция частицы частицы.
    private float[] pos = {0.0f, 0.0f, 0.0f};

    // Цвет частицы. (RGB формат 0xRRGGBB)
    private int color = 0xffffff;

    /** Пустая инициализация */
    public Particle()
    {

    }

    /**
     * Инициализация частицы.
     * @param velocity Установка скорости
     * @param position Установка позиции
     * @param color Установка цвета (не alpha)
     */
    public Particle(float[] velocity, float[] position, int color)
    {
        setVel(velocity);
        setPos(position);
        this.setColor(color);
    }
 /**
   * @param life Установка жизни.
   */
   void setLife(float life) {
   this.life = life;
   }

 /**
   * @return Возврат жизни.
   */
   float getLife() {
   return life;
   }

 /**
   * @param vel Установка скорости
   */
   void setVel(float[] tvel) {
   System.arraycopy(tvel, 0, vel, 0, vel.length);
```

```
}

/**
 * @return Возврат скорости.
 */
 float[] getVel() {
 return vel;
 }
/**
 * @param pos Установка позиции.
 */
 void setPos(float[] tpos) {
 System.arraycopy(tpos, 0, pos, 0, pos.length);
 }
/**
 * @return Возврат позиции.
 */
 float[] getPos() {
 return pos;
 }
/**
 * @param color Установка цвета.
 */
 void setColor(int color) {
 this.color = color;
 }
/**
 * @return Возврат цвета.
 */
 int getColor() {
 return color;
 }
/**
 * @param degradation Установка деградации.
 */
 public void setDegradation(float degradation) {
 this.degradation = degradation;
 }
/**
 * @return Возврат.
 */
 public float getDegradation() {
 return degradation;
 }
 }
```

ParticleEffect

```
import javax.microedition.m3g.Graphics3D;
/ **
 * Интерфейс, который определяет эффекты для показа,
 * которые производит двигатель частицы покажет.
 * ParticleEffect класс также содержит информацию об используемой bitmap
 * для показа частицы (если любой)
 */
 public interface ParticleEffect
 {
 // Инициализация частицы
 public void init(Particle p);

 // Модернизация частицы
 public void update(Particle p);

 // Рендеринг частицы
 public void render(Particle p, Graphics3D g3d);
 }
```

BitmapParticleEffect

```
import javax.microedition.m3g.Graphics3D;
import javax.microedition.m3g.Mesh;
import javax.microedition.m3g.PolygonMode;
import javax.microedition.m3g.Transform;
/**
 * Представляет эффект частицы, что использует bitmap.
 */
 public abstract class BitmapParticleEffect implements ParticleEffect
 {
 // Мешь
 Mesh mesh = null;

 // Матрица трансформации
 Transform trans = new Transform();

 // Масштабирование
 float scale = 1.0f;

 /** Инициализация bitmap, используемой для рендеринга частиц */
 public BitmapParticleEffect(String filename, float scale)
 {
 // Загрузка плана с желаемой текстурой
 mesh = MeshFactory.createAlphaPlane(filename, PolygonMode.CULL_BACK, 0xffffffff);

 // Устанавливаем масштаб
 this.scale = scale;
 }

 /**
 * @смотри ParticleEffect#render(Particle, Graphics3D)
 */
 public void render(Particle p, Graphics3D g3d)
 {
 // Вычисление альфа
 int alpha = (int)(255 * p.getLife());

 // Создание цвета
 int color = p.getColor() | (alpha << 24);

 // Установка альфа
 MeshOperator.setMeshAlpha(mesh, color);

 // Трансформация
 trans.setIdentity();
 trans.postScale(scale, scale, scale);
 float[] pos = p.getPos();
 trans.postTranslate(pos[0], pos[1], pos[2]);

 // Рендеринг
 g3d.render(mesh, trans);
 }
}
```

FountainEffect

```
import java.util.Random;
/**
 * Создает хороший эффект фонтана для частиц, который стреляет частицами
 * в некотором направлении, определенном его углом.
 *Угол может быть изменен в реальном времени.
 */
 public class FountainEffect extends BitmapParticleEffect
```

```
{
// Угол эмиссии частицы
частный int угол = 90;

// Синус и косинус текущего угла
private int angle = 90;

// Синус и косинус текущего угла
private float[] trig = {1.0f, 0.0f};

// Точка испускания
private float[] pos = {0.0f, 0.0f, 0.0f};

// Случайность
Random rand = null;

/**
* @param angle Угол эмиссии частицы
*/
public FountainEffect(int angle)
{
// Инициализация bitmap
super("/res/particle.png", 0.05f);

// Установить угол
 setAngle(angle);

// Получить случайное число
rand = new Random();
}
/**
  * @смотри ParticleEffect#init(Particle)
  */
  public void init(Particle p)
  {
  // Установка жизни частицы
  p.setLife(1.0f);

  // Установка позиции частицы
  p.setPos(pos);

  // Создание скорости(velocities) частицы
  float[] vel = new float[3];

  // Мы хотим скорости от 0.2f до 1.0f
  float xyvel = rand.nextFloat() * 0.8f + 0.2f;

  // Мы хотим, чтобы частица умерла медленно
  p.setDegradation(xyvel / 18);

  // Установка скорости согласно тригонометрии с маленьким отклонением(deviation)
  vel[0] = xyvel * trig[1] + rand.nextFloat() * 0.125f - 0.0625f;
  vel[1] = xyvel * trig[0] + rand.nextFloat() * 0.125f - 0.0625f;

  // Никакого движения в глубину
  vel[2] = 0.0f;

  // Установка скорости
  p.setVel(vel);

  // Установка случайного цвета
  int r = (int)(120 * rand.nextFloat()) + 135;
  int g = (int)(120 * rand.nextFloat()) + 135;
  int b = (int)(120 * rand.nextFloat()) + 135;
```

```
  int col = (r << 16) | (g << 8) | b;
  p.setColor(col);
  }
 /**
  * @see ParticleEffect#update(Particle)
  */
  public void update(Particle p)
  {
  // Просто обновление позиции
  float[] ppos = p.getPos();
  float[] vel = p.getVel();
  ppos[0] += vel[0];
  ppos[1] += vel[1];
  ppos[2] += vel[2];

  // Обновление жизни
  p.setLife(p.getLife() - p.getDegradation());

  // Проверка жизни. Если смерть, мы только переинициализируем
  if(p.getLife() < -0.001f)
  {
  init(p);
  }
  }

 /**
  * @param angle Установка угла.
  */
  public void setAngle(int angle) {
  this.angle = angle;
  trig[0] = (float)Math.sin(Math.toRadians(angle));
  trig[1] = (float)Math.cos(Math.toRadians(angle));
  }
 /**
  * @return Возврат угла.
  */
  public int getAngle() {
  return angle;
  }
 /**
  * @param pos Установка позиции.
  */
  void setEmittingOrigin(float[] pos) {
  this.pos = pos;
  }
 /**
  * @return Возврат позиции.
  */
  float[] getEmittingOrigin() {
  return pos;
  }
}
```

ParticleSystem

```
import javax.microedition.m3g.Graphics3D;
/**
  * Управление эмиссией частиц в Вашем 3D мире
  */
  public class ParticleSystem
  {
  // Эффект
  private ParticleEffect effect = null;

  // Частицы
```

```
Particle[] parts = null;

  / **
* Создает систему частиц, которая испускает частицы согласно определенному эффекту.
* @param effect  эффект, который управляет поведением частиц
* @param numParticles число частиц, чтобы испустить
*/
public ParticleSystem(ParticleEffect effect, int numParticles)
{
// Копирование эффекта
setEffect(effect);

// Инициализация частиц
parts = new Particle[numParticles];
for(int i = 0; i < numParticles; i++)
{
parts[i] = new Particle();
effect.init(parts[i]);
}
}

  / ** метод, который делает все. Он вызывается периодически каждый шаг цикла игры */
public void emit(Graphics3D g3d)
{
for(int i = 0; i < parts.length; i++)
{
getEffect().update(parts[i]);
getEffect().render(parts[i], g3d);
}
}
/**
* @param effect установка эффекта.
*/
public void setEffect(ParticleEffect effect) {
this.effect = effect;
}
/**
* @return Возврат эффекта.
*/
public ParticleEffect getEffect() {
return effect;
}
}
```

О Redikod

Redikod, из Мальмо(Malmo) в Швеции, - разработчик с 1997 сетевых и мобильных игр, и эта маленькая компания - теперь один из лидеров в скандинавской промышленности игр. Его непропорциональное влияние происходит от стратегических инициатив типа Скандинавского Оригинал статьиПотенциала Игр, ежегодной конференции, и скандинавского участия в E3 2006. Redikod уполномочен проектировать скандинавскую общественную систему поддержки и финансирования разработок для развития игр, включая мобильный телефон, что, ожидаемое заключительное принятие этой системы произойдет этой осенью и войдет в силу в 2006. Но разработка 3D- и мульти-плеерных для мобильного телефона - их ежедневная работа. Более подробно на WEB-сайте Redikod>>.

P.S.

Оригинал статьи   © Перевод, Сергей Кузнецов, 2007