Август 2005

Подарок от Redikod:
Учебное пособие по 3D- программированию для мобильных устройств, часть два: Теория 3D освещения и ориентации

Это – вторая часть из серии учебных пособий по Mobile Java 3D, созданных Mikael Baros, старшим программистом из Redikod, активным членом дискуссий на форуме разработчиков Sony Ericsson по Mobile Java 3D.

Ниже Вы можете загрузить и zip файлы исходного кода и пакета приложения для части два, так же можете освежить вашу память первой частью из учебного пособия.

Часть два: Теория 3D освещения и ориентации

Часть два путеводителя от Mikael-a.

Введение в теорию 3D освещения и ориентации

Это – вторая часть из серии учебного пособия по JSR 184 (M3G). Здесь я пройду некоторые основы 3D- теории, 3D- математики и затем закончу урок с простым демонстрационным примером ориентации. 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.

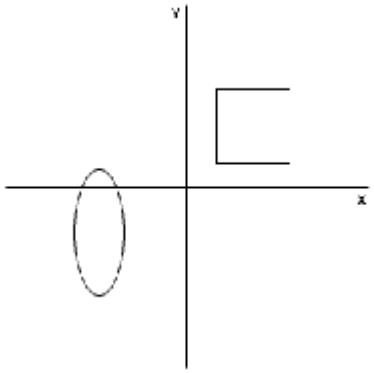
Цель этой обучающей программы состоит в том, чтобы дать Вам достаточно хорошее понимание 3D-математики, чтобы Вы могли использовать методы перемещения и ориентации из JSR 184's.
Я пройду 3D- систему координат, перемещение в 3D- пространстве месте и ориентацию(поворот) вокруг вектора. Также, Вы сможете использовать эти новые знания в конце обучающей программы, чтобы вращать меши в коде и размещать их в 3D- пространстве.Это однако не полностью освятит все что подразумевает название " Теория 3D освещения и ориентации". Более поздние главы этой серии учебных пособий будут содержать более продвинутые разделы по этой теме. 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.

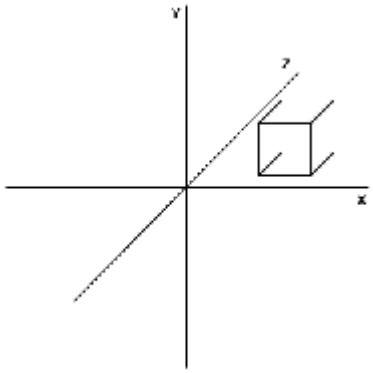
Что Вы должны знать

Прежде, чем Вы начнете читать это, Вы должны были прочитать первую обучающую программу этой серии и должны иметь основное понимание Исчисления и Линейной Алгебры. Знать математику не обязательно, но это поможет пониманию.

Трехмерная система координат

Трехмерная система координат много походит на двухмерную, исключая добавочную ось. Эту ось, ось Z, также обычно называют глубиной.





Двухмерная система координат. Трехмерная система координат.

Как Вы можете видеть на картинке выше, в 3D- объекты не только имеют ширину и высоту (x, y) но также и глубину(z).
Таким образом, точка как трехмерный объект всегда определяется с тремя координатами x, y и z. 3D-объекты состоят из массива вершин (vertrices), которые определяют лица модели. Куб имеет восемь углов и таким образом восемь вершин (хотя многие из них общие). Так что координаты для куба могли быть (в порядке x, y, z):

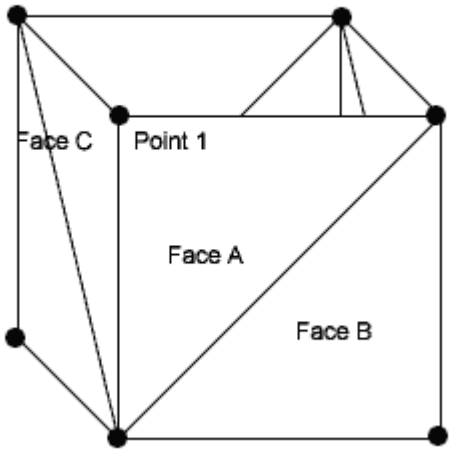
Передняя грань:

-1.0, -1.0, 1.0
-1.0, 1.0, 1.0
1.0, -1.0, 1.0
1.0, 1.0, 1.0

Задняя грань:

-1.0, -1.0, -1.0
-1.0, 1.0, -1.0
1.0, -1.0, -1.0
1.0, 1.0, -1.0

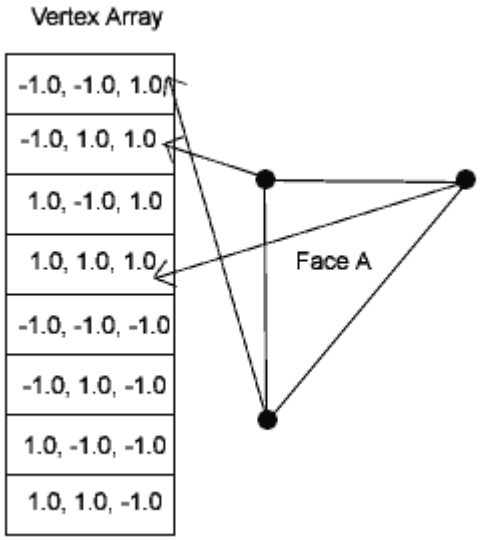
Выше мы имеем вершины куба с его центром в начале системы координат(0, 0, 0). Также обратите внимание, что задняя грань идентична передней грани, кроме координаты z (глубины), которая является +1 для передней грани и -1 для задней грани. Если Вы подумаете об этом, то это логично. Так как ось Z также называют глубиной, тогда логично, задняя грань имеет отрицательную глубину.
Однако, даже если вышеупомянутые вершины необходимы для куба, мы также должны знать грани. 3D-rasterizer не знает, что сделать с простыми 3D-вершинами, как они могут быть превращены в пикселы и в результате все закончится черчением на экране 8 пикселов, хотя мы фактически хотим куб.



Это - то, где грани пересекаются. Объект имеет список вершин (точно так же как выше упомянуто, для куба), из которого его грани состоят. Грань - поверхность 3D-модели и составлено из 3 или больше вершин. Обычно все модели состоят из большого количества разбитых на треугольники граней, то есть каждая грань – треугольник, состоящий из 3 вершин.

Так, например, наш куб нуждается в двух треугольниках для каждой стороны (так как два треугольника составляют квадрат), и куб имеет 6 сторон, так что мы в итоге имеем 6*2 = 12 треугольников. Так как каждый треугольник имеет 3 вершины, в конечном счете нужно 12 * 3 = 36 вершин, чтобы определить наш куб. Это - намного больше чем оригинал из 8 вершин, не так ли? Однако, это – конец истории. Поскольку Вы уже поняли, многие тех 12 треугольников будут иметь те же самые вершины - как показано на иллюстрации справа.

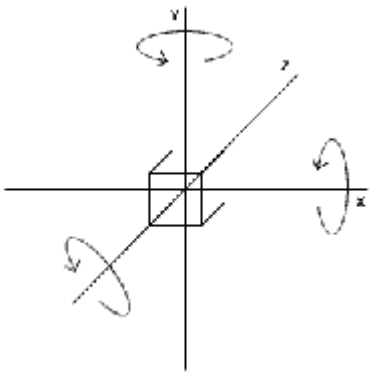
Как Вы можете видеть, вершина, помеченная "Вершина 1" фактически общая у грани A и грани C. Так что мы в действительности не нуждаемся в хранении всех 36 вершин. Фактически, достаточно первых восьми вершин, сохраненных в массиве, и они затем позволят треугольникам иметь индексы на все в наше множество вершины. Этим путем мы экономим 36 - 8 = 28 вершин. Поскольку комплекс моделирует это число в большее.
Направо - картина, которая иллюстрирует множество вершин и индексы треугольника.
Если Вы не заметили, показанная грань - то же самая грань нашего куба, и все восемь вершин в массиве вершин - наши восемь вершин куба.
Как Вы видите, каждый треугольник не должен отдельно хранить три вершины, состоящие из трех координат каждая, для превращения девяти переменных с плавающей точкой в треугольник.
Вместо этого, каждый треугольник только содержит три индекса из целых чисел. Это большая экономии памяти, особенно если Вы делаете в модели от 300 треугольников или больше.



Ориентация и Перемещение

Каждая вершина в модели(объекта) поворачивается и перемещается из собственной (объекта) систем координат в глобальные (мировые) координаты. Это для того, чтобы мы могли фактически показать модели в любой точке 3D-мира, а не только в системе координат модели. (Для ссылки, большинство моделей создано в системе координат модели. Если бы мы не переводили модели к мировым координатам, мы всегда рендерили бы все в координаты нашего мира. Скучно!).

Так что это занятие переводом очевидно довольно важно. Вы спрашиваете, как это сделать? Это довольно просто, действительно. Все, что Вы должны сделать, чтобы переместить модель, должны просто перевести все ее вершины.Например, наш куб в примере выше был создан в его собственной системе координат, но мы хотим, чтобы куб был показан в координатах 10.0, 0.0,-10.0. Это – на 10 единиц направо и на 10 единиц вглубь. Чтобы сделать это, мы просто возьмем наш массив вершин и только добавим 10.0 к координате x каждой вершины и вычтем 10.0 из координаты z каждой вершины. Просто!
Если Вы помните первую обучающую программу, мы переместили нашу камеру, чтобы двигаться вокруг в мире. Камера может также быть рассмотрена как 3D-объект, так что мы для перемещения камеры только добавили или вычли из координат вершин камеры.
Обычно, мы также хотим вращать наши объекты. Вращение немного труднее для понимания чем перевод, так как это более сложно в принципе.



Справа Вы видите нашу прежнюю картину куба, теперь немного измененную:
Как Вы видите, объект в 3D- пространстве может вращаться вокруг одной из этих трех осей. Вращение вокруг оси Y также называют отклонением от курса (the yaw), вращение вокруг оси X называют подачей (the pitch), и вращение вокруг оси Z называют перекличкой(the roll). Однако, в этой обучающей программе я просто назову их вращениями вокруг любой оси, о которой я говорю.

Вращение вокруг оси лучше всего показать взяв кубик сыра. (Да, пожалуйста пойдите и возьмите маленькую часть сыра из холодильника, и принесите три зубочистки также.) Этот сыр напомнит наш куб. Теперь, вонзите зубочистки в сыр, где оси вращения, и вращайте зубочистки, Вы получите

вращение вокруг осей. Вращение вокруг оси всегда выполняется против часовой стрелки для положительных углов и по часовой стрелке для отрицательных углов.

Вращение 3D- объекта действительно довольно просто, поскольку Вы можете вращать объект вокруг ваших трех осей и вот все ... или - нет? Прежде всего, Вы можете вращать ваш объект вокруг любого вектора, а не только вокруг этих трех осей. Например, если Вы воткнете зубочистку в часть сыра по диагонали от одного угла к другому, Вы создадите вектор, который - не ось, но тем не менее Вы можете крутить зубочистку и вращать сыр. Если Вы помните первую главу, я упоминал метод вращения, используемый в JSR 184. Это выглядит так:

nameOfRotation(float degrees, float x, float y, float z)

Три значения с плавающей точкой x, y и z, являются компонентами вектора, вокруг которого Вы желаете вращать. Теперь, чтобы вращать вокруг единственной оси Вы только снабжаете вектором оси X. Вектор оси X - (весьма логически) 1.0, 0.0, 0.0. То есть +1 на оси X и 0 на других осях.

Хотя есть некоторые проблемы с 3D- вращением. А именно, вращение чувствительно к порядку проведения вращений. Вращение объекта вокруг оси X, затем вокруг оси Y и затем вокруг оси Z полностью отличается от выполнения этого в любой другой последовательности.

Также, если Вы вращали объект вокруг одной оси, Вы также вращаете ее другие оси. Это создает другую проблему, куда ось X фактически перемещается, если Вы вращаете вокруг оси Y сначала, и таким образом вращение вокруг оси X не будет давать желательный эффект. Обычно простая 3D-игра не должна волноваться об этом, поскольку каждый обычно не вращает большинство моделей одновременно вокруг всех осей, и даже если Вы делаете, Вы можете обычно хранить угол вращения на каждой оси и использовать метод setRotation. Я не буду входить в глубину об Ориентации объекта и его осей здесь, это будет рассмотрено в одной из продвинутых обучающих программ позже. Не позвольте этой неприятности прямо сейчас изменить ваше мнение.

Другая вещь во вращении, о которой я упомяну - локальное вращение и мировое вращение. Есть различие, если Вы вращаете объект в его локальной системе координат или если Вы вращаете его после того, как он был переведен в мировые координаты. Различие – в том, что вращение в его локальной системе координат, происходит вокруг его локальных осей координат, а в мировой системе координат, он будет вращаться вокруг всемирных осей.

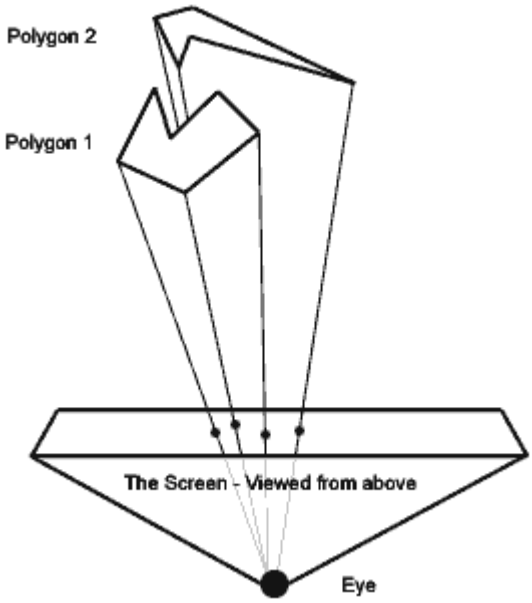
3D Вселенная

3D-понятие немного труднее для представления в голове чем 2D. Мы можем все же коснуться 2D-графики, где Вы имеете массив пикселов, которые Вы просто выводите на экран. Однако, 3D- модели и вселенная в 3D- игре не описаны так "физически"(прим. массивом пикселей) как 2D- изображение в PNG. В противовес этому все модели математически описаны с вершинами (vertrices) в 3D- системе координат. Эти вершины, перемещаются и поворачиваются(преобразуются) в их новое положение со всеми переводами и вращениями, которые Вы применили к 3D-модели, и наконец в графическом канале , подготавливаются пикселы для граней, определенных вершинами. Запутано все же?
Я не буду входить в формулы проецирования, или во что – нибудь из трудной алгебры, но я думаю, что Вы должны по крайней мере знать, что ваши модели не "реальны" или "видимы" до последней стадии рендеринга, где области (многоугольники из граней), определенные вершинами (vertrices) подготовлены черчения (прим. преобразованы) на плане (экране).

Вам поможет представить 3D-е проецирование рисунок справа.

Это изображение - очень упрощенное изображение проецирования, но достаточно Вам для представления в вашей голове. Как Вы видите, "линии", начерченные от многоугольников до глаза, - линии проецирования.

Проецирование может фактически быть представлено как пули, летящие от вершин модели, прямо в глаз наблюдателя. Пуля, продвигаясь к глазу, проникает через свинцовую пластину (экран) и отлетает этого, оставляя вмятину(пиксел) на экране. Также, если пуля от Многоугольника 2 должна пройти через Многоугольник 1, то она никогда не попадет в к экрану, так как Многоугольник 2 расположен более глубже в сцене. Конечно, процесс проецирования намного более сложен, чем делает эта упрощенная модель.



Проецирование двух многоугольников на экран.

We will be using the exact same canvas from the first tutorial, but we'll add some methods and change the constructor code. We don't want to load our world anymore, we'll load a cube and display it. This is a fairly easy procedure as we only change the file name from map.m3g to cube.m3g in our loader method from the first tutorial. More in-depth knowledge of the various rendering modes and models comes later.

Кодирование

Хорошо, чтобы помощи Вам в понимании вышеупомянутой части теории, мы создадим некоторый код с использованием этих новых знаний. Мы будем использовать точный тот же самый холст из первой обучающей программы, но мы добавим некоторые методы и изменим код конструктора. Мы не хотим больше загружать наш мир, а мы загрузим куб и покажем его. Это - довольно легкая процедура, поскольку мы только изменяем имя файла от map.m3g на cube.m3g в нашем методе загрузчика(loader) из первой обучающей программы. Более глубокое знание различных режимов рендеринга и моделей будет позже.

```
/** Загрузка мира */
   private void loadWorld()
   {
      try
      {
        // Загрузка мира очень проста. Обратите внимание, что я люблю использовать а
         // res-папка, что я удерживаю все файлы. Если Вы обычно только помещаете ваш
         // ресурсы в проектном корне, затем загрузите это от корня.
         Object3D[] buffer = Loader.load("/res/cube.m3g");

         // Find the world node, best to do it the "safe" way
         for(int i = 0; i < buffer.length; i++)
         {
            if(buffer[i] instanceof World)
            {
               world = (World)buffer[i];
               break;
```

```
            }
        }

        // Очистка объектов
        buffer = null;

        // Сейчас найдем куб и будем его вращать
        // Мы знаем  ID куба
        cube = (Mesh)world.find(13);
    }
    catch(Exception e)
    {
        // ERROR!
        System.out.println("Loading error!");
        reportException(e);
    }
}
```

Метод (прим. loadWorld()) в основном тот же самый, кроме последней строки cube = (Mesh)world.find(13);. Эта строка может показаться немного озадачивающей, так что давайте говорить об ней.Обычно объекты в графе сцены JSR 184 имеют ID's, который может их однозначно идентефицирует (хотя отдельные объекты, могут не иметь ID). Этими ID можно непосредственно манипулировать при экспорте модели из вашей любимой программы 3D- моделирования.
При выполнении find(найти)-метода с объектами мира, будет найден объект с этим ID, если он (ID) существует и связан с объектом.Самый легкий путь иллюстрации этого, это - HashMap, который имеет ключ - как ID объекта и значение – как сам объект. Find метод – метод входит в класс Object3D, так почти каждый класс в JSR 184 имеет этот метод(прим. наследует).

Теперь, когда я создавал куб, я экспортировал его так, чтобы его ID был 13. Поскольку я знаю это, я могу легко извлечь Мешь куба из Мира и манипулировать им (вращать, перемещать, масштабировать, и т.д ...). Класс Mesh(Меш;Петля) - очень универсальный класс, который содержит всю необходимую информацию, которую модель должна иметь; буфер вершин, буфер индексов треугольника, координаты текстуры, и т.д... Он даже содержит в классе Appearance(Внешность) продвинутые режимы для операций рендеринга(представления). Мы будем говорить больше об этом фантастическом классе позже. Пока, знайте, что Mesh представляет собой модель объекта и является подклассом Transformable(Трансформируемые).

Трансформируемые

Абстрактный класс (прим. Класс лишь наследуется другими классами) представляется как объект в 3D- пространстве, который может быть трансформирован некоторым способом (масштабирован, повернут или перемещен). Много объектов наследуют этот класс. Правило большого пальца - любой объект, который может двигаться в 3D-вселенной является подклассом Transformable.
 Если Вы хотите детальную информацию о классе Transformable, пожалуйста, смотрите документацию к API JSR 184. Он имеет много полезных методов, которые мы исследуем подробно в других частях этой обучающей программы. Пока мы будем использовать только один метод, preRotate(предвращащение. Это - метод, который просто вращает объект прежде, чем любые перемещения применены, и таким образом вращает его(прим.объект) вокруг его собственных пространственных осей. (Помните я говорил об этом ранее в этой обучающей программе.).

Так, использованием preRotate мы достигаем эффекта -мы всегда вращаем куб правильно, независимо от того где он помещен в нашем мире. preRotate методы - с четырьмя аргументами с плавающей точкой, подобно любому методу вращения. Первый – угол поворота, и последние три представляют вектор, вокруг которого мы желаем вращать. Давайте посмотрим на наш новый метод moveCube.

```
private void moveCube() {
    // Проверка клавиш
    if(key[LEFT])
    {
```

```
            cube.preRotate(5.0f, 0.0f, 0.0f, 1.0f);
        }
        else if(key[RIGHT])
        {
            cube.preRotate(-5.0f, 0.0f, 0.0f, 1.0f);
        }
        else if(key[UP])
        {
            cube.preRotate(5.0f, 1.0f, 0.0f, 0.0f);
        }
        else if(key[DOWN])
        {
            cube.preRotate(-5.0f, 1.0f, 0.0f, 0.0f);
        }
        // Если пользователь нажимает ключ FIRE , давайте выйдем из мидлета
        if(key[FIRE])
            M3GMidlet.die();
    }
```

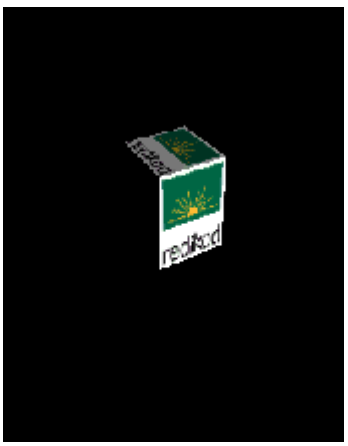
Как Вы видите, все, что мы делаем – это вызываем метод и preRotate с указанием угла поворота степени и соответствующая оси вращения. Теперь, левый и правый повороты вокруг оси Z (помните, ось Z - вектор со всеми 0 кроме координаты z = 1.0), и вращение вверх и вниз происходит вокруг оси X. Мы в демонстрационном примере не вращаем вокруг оси Y, но это - не трудно добавить в код. Рассматривайте это как задание Вам, сделать вращение куба вокруг оси Y при нажатии другой пары клавиш.

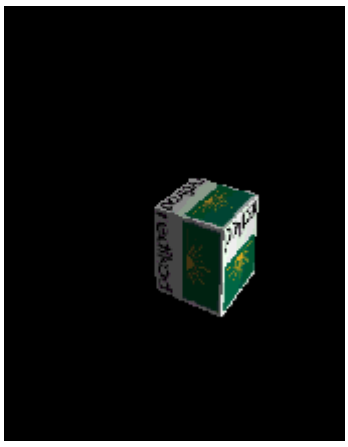
Мы вызываем метод moveCube один раз каждом шаге цикла игры для обновления вращения куба, также, как мы сделали с методом moveCamera в предыдущей обучающей программе.

Остальная часть кода в этом демонстрационном примере та же самая как в первом демонстрационном примере, так что я не буду говорить повторно об этом здесь. Внесите изменения в листинг кода как указано ниже, или загрузите исходный код, и лично убедитесь.

Заключение

Так это все выглядит, смотрите ниже - несколько screenshots кода в действии:





Хорошо, мы имеем наш вращающийся куб. Я надеюсь, что Вы теперь имеете большее понимание 3D-систем координат и 3D- вращения. Пожалуйста, продолжайте читать эту серию обучающих программ для более продвинутых тем разработки 3D- приложений с API JSR 184. Ниже листинг кода этого примера. Вы можете также разгрузить код как отдельный ZIP файл, содержащий все ресурсы, так что Вы можете играть с этим дома.

M3GMidlet

```
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 M3GMidlet extends MIDlet implements CommandListener
{
   // Переменная, которая держит уникальный экран
 private Display display = null;

 // Холст(canvas)
 private M3GCanvas canvas = null;

 // Ссылка мидлета на самого себя
 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
 {
    // Allocate
  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.Camera;
import javax.microedition.m3g.Graphics3D;
import javax.microedition.m3g.Group;
import javax.microedition.m3g.Loader;
import javax.microedition.m3g.Mesh;
import javax.microedition.m3g.Object3D;
import javax.microedition.m3g.Transform;
import javax.microedition.m3g.World;
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 мира
    World world = null;
```

```
    // Глобальный объект камеры
    Camera cam = null;

    // Вращение камеры
    float camRot = 0.0f;
    double camSine = 0.0f;
    double camCosine = 0.0f;

    // Подпрыгивание (Удар в голову)
    float headDeg = 0.0f;

    // Модель куба, которой желаем управлять
    Mesh cube = null;

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

        // Мы желаем полноэкранный холст
        setFullScreenMode(true);

        // Загружаем мир
        loadWorld();

        // Загружаем камеру
        loadCamera();
    }

/ ** 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()
    {
        // BAD!
        if(world == null)
            return;
       // Получить активную камеру от мира
        cam = world.getActiveCamera();

    }
/** Загрузка мира */
    private void loadWorld()
    {
        try
```

```
      {
         // Загрузка мира очень проста. Обратите внимание, что я люблю использовать а
         // res-папка, что я удерживаю все файлы. Если Вы обычно только помещаете ваш
         // ресурсы в проектном корне, затем загрузите это от корня.
         Object3D[] buffer = Loader.load("/res/map.M3G");

         // Найти мировой узел, лучше всего сделать этот "безопасный" путь
         for(int i = 0; i < buffer.length; i++)
         {
            if(buffer[i] instanceof World)
            {
               world = (World)buffer[i];
               break;
            }
         }

         // Очистка объектов
         buffer = null;

        // Сейчас найдем куб и будем его вращать
         // Мы знаем  ID куба
         cube = (Mesh)world.find(13);        }
      catch(Exception e)
      {
         // ERROR!
         System.out.println("Loading error!");
         reportException(e);
      }
   }

/** Чертим на экране
    */
   private void draw(Graphics g)
   {
      // Окутать все try/catch блоком на всякий случай
      try
      {
         // Переместить куб вокруг
            moveCube();

         // Получить Graphics3D контекст
         g3d = Graphics3D.getInstance();

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

       // Теперь, только отдайте мир. Просто как пирог!
       g3d.render(world);
      }
      catch(Exception e)
      {
         reportException(e);
      }
      finally
      {
        // Всегда не забудьте отвязать!
         g3d.releaseTarget();
      }
```

```
    }



    /**
     *
     */
    private void moveCube() {
        // Проверка клавиш
        if(key[LEFT])
        {
            cube.preRotate(5.0f, 0.0f, 0.0f, 1.0f);
        }
        else if(key[RIGHT])
        {
            cube.preRotate(-5.0f, 0.0f, 0.0f, 1.0f);
        }
        else if(key[UP])
        {
            cube.preRotate(5.0f, 1.0f, 0.0f, 0.0f);
        }
        else if(key[DOWN])
        {
            cube.preRotate(-5.0f, 1.0f, 0.0f, 0.0f);
        }
        // Если пользователь нажимает ключ FIRE , давайте выйдем из мидлета
        if(key[FIRE])
            M3GMidlet.die();
    }
/** 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; }
}
```

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 Потенциала 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.