3D графика для Java мобильных устройств,
Часть 1: M3G's непосредственный режим(immediate mode).

Создание 3D сцен с помощью JSR 184

Уровень: Вводный
Claus Höfele(mailto:Claus.Hoefele@gmail.com?subject=M3G's immediate mode),
Автор, Внештатный журналист
11 Oct 2005

Эта статья, первая в серии из двух статей, описывает Мобильный 3D Графический программный интерфейс приложения(Mobile 3D Graphics API JSR 184). Автор введет Вас в 3D- программирование для Явы ™ мобильных устройств и покажет, как Вы можете работать с источниками света, камерами, и материалами.

Игра в игры на мобильных устройствах - забавное времяпрепровождение. Вплоть до сих пор, скорость работы аппаратных средств телефонов, позволяла выполнять лишь классические игры с использованием захватывающей логики игры, но простой графикой. Сегодня, Tetris и Pac-Man все более и более дополнены двухмерными играми –действиями с улучшенной графикой. Следовательно, следующий шаг должен быть - к 3D-графике. PlayStation Portable от Sony показывает, что мощную графику Вы можете поместить в мобильное устройство. Хотя средний мобильный телефон - технологически позади этой специализированной игровой машины, Вы можете видеть, куда идет рынок. Мобильный 3D Графический программный интерфейс приложения(Mobile 3D Graphics API; коротко M3G), определенный в Требованиях Спецификации Явы 184(Java Specification Request; JSR 184), является усилием промышленности создать стандартный 3D API для мобильных устройств с поддержкой программирования на Яве.
M3G's API может быть разделен грубо в две части: непосредственный и сохраненный режимы.
В непосредственном режиме, Вы определяете и отображаете индивидуальные 3D- объекты.
В сохраненном режиме, Вы определяете и показываете полный мир 3D- объектов, включая информацию относительно их внешности. Вы можете вообразить непосредственный режим как доступ низкого уровня к 3D-функциям, а сохраненный режим как более абстрактный, но также и более удобный, режим показа 3D- графики. В этой статье, я объясню непосредственный режим API. Во второй статье этой серии покажу, как использовать сохраненный режим.

Альтернативы M3G
M3G не один.

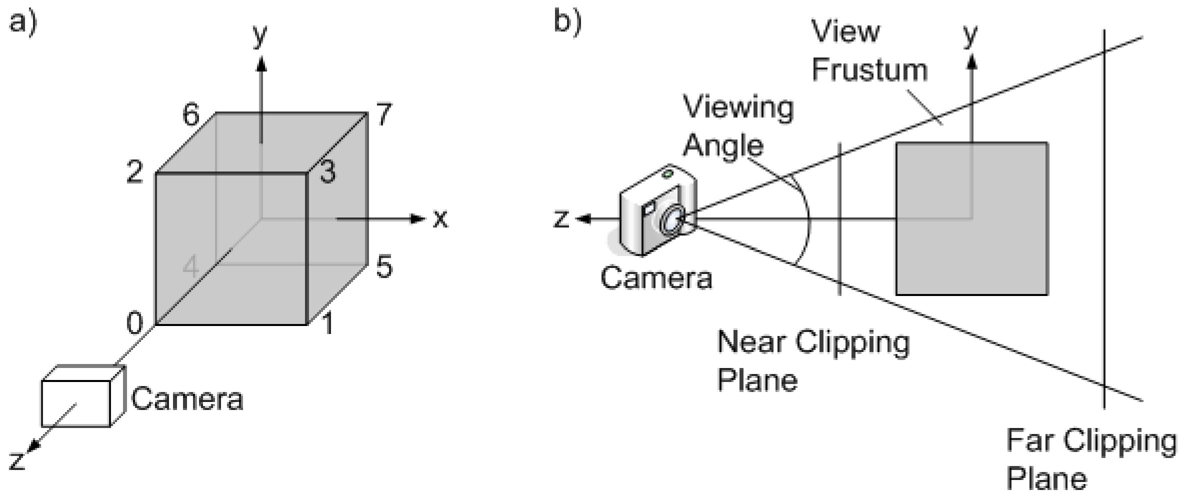
Mascot Capsule(Капсула Талисмана) API Корпорации популярен в Японии, где все три главных оператора используют это в различных инкарнациях, и за границей. Sony Ericsson, например, поставляет телефоны и с M3G и с Mascot Capsule HI проприетарным API. Прикладные разработчики сообщают на Вебсайте Sony Ericsson, что Mascot Capsule API - устойчивая и быстрая 3D-окружающая среда.

API интерфейса Явы для OpenGL ES(JSR 239, Java Bindings for OpenGL ES), предназначена для устройств подобно M3G. OpenGL ES - поднабор широко известной библиотеки OpenGL 3D и становится фактическим стандартом для родного 3D - выполнения на ограниченных устройствах. JSR 239 определяет Java API, который напоминает API OpenGL ES -интерфейс с языком C, в максимально возможной степени облегчая портацию, существующего OpenGL содержания. С сентября 2005, JSR 239 находится все еще в раннем статусе проекта. Я могу только размышлять, возымеет ли это воздействие на мобильные телефоны. Однако, в то время как не совместимый с API M3G, API OpenGL ES действительно влиял на архитектуру M3G: экспертная группа по JSR 184 предусмотрела, что быть возможным эффективно осуществить M3G на вершине OpenGL ES. Если Вы знаете OpenGL, Вы обнаружите много его особенностей в M3G.

Несмотря на альтернативы, M3G имеет поддержку всех главных изготовителей телефонов и операторов. Хотя я упомянул игры, поскольку это главная привлекательность, M3G - программный интерфейс приложения общего назначения, который Вы можете использовать, чтобы создать все виды 3D- содержания. Это будет 3D API, чтобы использовать для мобильных телефонов в течение многих последующих лет.

Ваш первый 3D объект
В качестве первого примера, Вы создадите куб как показано на Рисунке 1.

Рисунок 1. Типовой куб: a) Передний вид с индексами вершины, b) Вид со стороны с планами обрезания(Фронт, Сторона)



Куб живет в системе координат, которую M3G определяет как систему координат правой руки. Если Вы берете вашу правую руку и вытягиваете ваши большой, указательный и средний пальцы так, чтобы каждый палец находится в прямом угле к другим двум пальцам, то большой палец - ваша ось X, указательный палец ось Y, и средний палец ось Z. Попытайтесь выровнять ваш большой и указательный пальцы с осями X и Y в соответствеии с Рисунком 1a; ваш средний палец(ось Z)должен тогда указывать на Вас. Я использовал восемь вершин(точки углов куба) и поместил центр куба в начало системы координат.
Как Вы видите на Рисунке 1, камера, которая снимает 3D-сцену, смотрит по отрицательной оси Z, располагаясь перед кубом. Положение камеры и ее свойства определяют, что позднее будет показано на экране.
Рисунок 1b показывает представление стороны той же самой сцены(профиля сцены), так что Вы легко видите, какая часть 3D- мира видима камерой. Один фактор ограничения - угол обзора, который определяется линзой камеры: теле-фото-графическая линза имеет более узкий обзор, чем широко-угловая линза. Угол обзора определяет как широко вы видите стороны. В отличие от реального мира, 3D- вычисление дает Вам еще две границы вида(плана) мира : ближний и дальний урезанные планы(виды). Вместе, угол обзора и планы урезания определяют то, что называют планом усечения(*view frustum*). Все что находится внутри планов усечения - видимо, все что снаружи - нет.

Все это реализовано в классе VerticesSample, члены которого Вы видите в Листинге 1.

Листинг 1. Пример, показывающий куб, Часть 1: члены Класса

```
package m3gsamples1;

import javax.microedition.lcdui.*;
import javax.microedition.m3g.*;


/**
 * Пример, показывающий куб, определенный восьмью вершинами, которые связаны
 * треугольниками.
 *
 * @author Claus Hoefele
 */
public class VerticesSample extends Canvas implements Sample
{
 /** координаты вершин куба(x, y, z).  */
 private static final byte[] VERTEX_POSITIONS = {
   -1, -1,  1,    1, -1,  1,   -1,  1,  1,    1,  1,  1,
   -1, -1, -1,    1, -1, -1,   -1,  1, -1,    1,  1, -1
 };

  / ** Индексы, которые определяют, как соединить вершины, чтобы строить
  * треугольники. */
 private static int[] TRIANGLE_INDICES = {
```

```
  0, 1, 2, 3, 7, 1, 5, 4, 7, 6, 2, 4, 0, 1
 };

/ ** данные вершин куба. */
private VertexBuffer _cubeVertexData;

/ ** треугольники куба, определенные как полосы из треугольников. */
private TriangleStripArray _cubeTriangles;



 / ** Графический singleton, используемый для рендеринга. */
private Graphics3D _graphics3d;
```

Класс VerticesSample наследован от Canvas, чтобы иметь возможность непосредственно рисовать на экране. Он также имеет встроенный интерфейс Sample, который я определил для удобства структурирования различных тематических примеров кода этой статьи.
VERTEX_POSITIONS определяет восемь вершин в том же самом порядке, что и на Рисунке 1а. Например, вершина 0 определена с координатами(-1,-1, 1). Помните, что я поместил центр куба в начало системы координат; поэтому длина граней куба будет две единицы. Положение камеры и угол обзора позже определят количество пикселов, которые единица длины займет на экране.

Одних координат вершин - недостаточно, однако; Вы также должны сказать, какую стереометрию Вы хотите построить. Подобно рисованию от точки-к-точке, Вы должны соединить вершины линиями, пока Вы не увидите результат черчения. M3G налагает одно ограничение: Вы должны строить стереометрию из треугольников. Треугольники -популярные примитивы 3D - реализации, так как любой многоугольник Вы можете определить как набор треугольников. Построение треугольника - основная базовая операция черчения, на основе которой Вы можете строить более абстрактные операции.

К сожалению, если бы Вы должны были описать куб с одними треугольниками, Вы нуждались бы в 36 вершинах = 6 сторон * 2 треугольника * 3 вершины в каждом; это было бы тратой памяти, так много вершин дублировано.
Чтобы уменьшить память, Вы сначала отделяете вершины от определений треугольников.
Массив TRIANGLE_INDICES(массив индексов треугольников) определяет стереометрию, используя массив индексов вершин VERTEX_POSITIONS, строя его(массив индексов треугольников) таким образом чтобы избежать большого повторного использования индексов вершин. Затем, для уменьшения числа индексов, используем полосы треугольников вместо треугольников. С использованием полосы треугольников, новый треугольник повторно использует последние два индекса предыдущего треугольника. Например, полоса треугольника(0, 1, 2, 3) транслируется в два треугольника(0, 1, 2) и(1, 2, 3). Когда Вы следуете по TRIANGLE_INDICES(полосе; массиву) за определениями треугольников,(смотри Рисунок 1а, где я отметил каждый угол соответствующим индексом), Вы будете видеть, что треугольники дико подскакивают между сторонами куба. Это - только пример, чтобы избежать определения нескольких полос. Я управился с восьмью вершинами куба одной единственной полосой треугольников с 14 индексами.
Остальную часть членов класса Вы используете для черчения куба. Листинг 2 показывает их инициализацию.

Листинг 2. Пример показа куба, Часть 2: Инициализация

```
/**
  * Вызывается когда пример показывается на экране.
  */
 public void showNotify()
 {
   init();
 }

 /**
  * Инициализация примера.
  */
 protected void init()
 {
   // Получение экземпляра(singleton) для 3D рендеринга.
   _graphics3d = Graphics3D.getInstance();
```

```
  // Создание данных вершин.
  _cubeVertexData = new VertexBuffer();

  VertexArray vertexPositions =
     new VertexArray(VERTEX_POSITIONS.length/3, 3, 1);
  vertexPositions.set(0, VERTEX_POSITIONS.length/3, VERTEX_POSITIONS);
  _cubeVertexData.setPositions(vertexPositions, 1.0f, null);

  // Создание треугольников определяющих куб; индексы
  // указывают на вершины в VERTEX_POSITIONS.
  _cubeTriangles = new TriangleStripArray(TRIANGLE_INDICES,
     new int[] {TRIANGLE_INDICES.length});

  // Создание камеры с переспективной проекцией.
  Camera camera = new Camera();
  float aspect =(float) getWidth() /(float) getHeight();
  camera.setPerspective(30.0f, aspect, 1.0f, 1000.0f);
  Transform cameraTransform = new Transform();
  cameraTransform.postTranslate(0.0f, 0.0f, 10.0f);
  _graphics3d.setCamera(camera, cameraTransform);
 }
```

Первым шагом в init(), Вы получаете графический контекст для 3D-черчения. Graphics3D - экземпляр(singleton), и _graphics3d содержит ссылку на него для последующего использования.
Затем, Вы создаете VertexBuffer, чтобы управлять данными вершин. Вы позже увидите , что Вы можете назначить несколько видов информации для вершин, и VertexBuffer будет содержать все эти виды. В настоящее время, единственная информация в VertexBuffer, в которой Вы нуждаетесь - координаты вершин в VertexArray, которая установлена при помощи _cubeVertexData.setPositions(). Конструктор VertexArray берет параметры: число вершин(восемь), число координат в вершине(x, y, z), и размер каждой координаты(один байт). Поскольку куб мал, одного байта достаточно, чтобы хранить одну координату.
Если Вы хотите создать большие объекты, Вы можете создать VertexArray с использованием short значений(два байта). Однако, Вы не можете использовать реальные числа, только целые числа. Затем, Вы инициализуете TriangleStripArray индексами из TRIANGLE_INDICES.

Заключительная часть кода инициализации - установка камеры. В setPersective(), Вы устанавливаете угол обзора(viewing angle), коэффициент сжатия(aspect ratio), и обрезающие планы(clipping planes). Заметьте, что значения коэффициента сжатия и обрезающих планов- значения с плавающей точкой. M3G требует поддержки плавающей точки, которая стала доступна в Виртуальной машине Ява(JVM) с CLDC 1.1.

После установки перспективы, Вы отодвигаете камеру от куба, так что бы Вы имели полный обзор объекта. Вы делаете это преобразованием(трансформацией), которую я объясню более подробно в секции о Преобразованиях(Transformations). Пока, поверьте мне, что postTranslate() с положительным третьим параметром, переместит камеру по оси Z.
После инициализации, Вы рендерите сцену на экран. Листинг 3 показывает это.

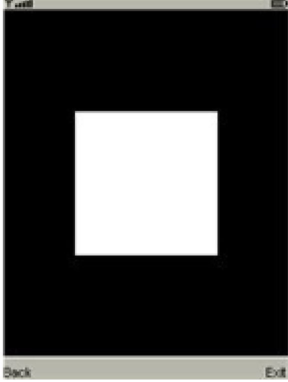
Листинг 3. Пример показа куба, Часть 3: Черчение

```
 /**
  * Рендеринг примера на экран.
  *
  * @param graphics графический объект для черчения.
  */
 protected void paint(Graphics graphics)
 {
  _graphics3d.bindTarget(graphics);
  _graphics3d.clear(null);
  _graphics3d.render(_cubeVertexData, _cubeTriangles,
     new Appearance(), null);
  _graphics3d.releaseTarget();
 }
```

bindTarget(), в paint(),назначает графический контекст Холста(Canvas's) объекту Graphics3D. Это позволяет рендерить 3D объекты до тех пор пока не вызван releaseTarget(). После вызова clear(),фон стерт, Вы чертите куб, используя данные вершин и треугольников, созданные в init().
Многие из Graphics3D's методов бросают непроверенные исключения, но я решил обойтись без пробующегося/ловящегося(try/catch) блока, потому что большинство ошибок невосстанавливаемо. Вы можете найти полный исходный код этого примера в VerticesSample.java.
Чтобы показать пример на экране телефона , я написал простой MIDlet. Вы можете получить это в секции download section вместе с полным исходным кодом. Вы можете увидеть результат загрузки примера на Рисунке 2.

О коде примера
Если для примеров в этой статье вы желаете изготовить мидлеты и запустить их, Вы должны загрузить исходные коды из секции download. Я использовал набор инструментов Sun's Java Wireless Toolkit 2.2 и сконфигурировал мой проект на использование MIDP 1.0, CLDC 1.1, и естественно M3G. Я построил каждый пример статьи как отдельный класс. Я также построил простой интерфейс для выбора и выполнения каждого примера. Файл readme.txt, включенный в wi-m3gsamples1.zip содержит много подробностей.

Рисунок 2. Пример куба



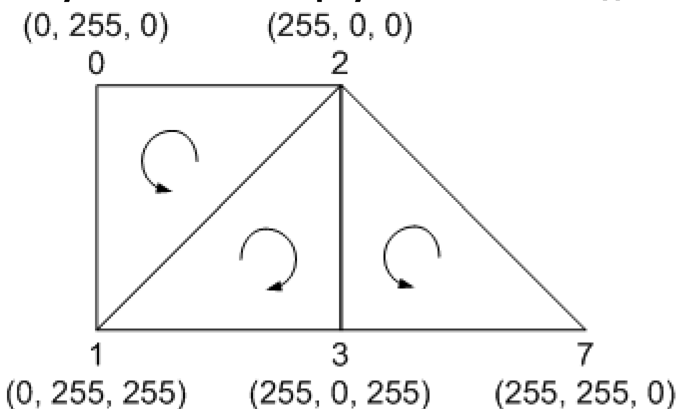
Трудно утверждать, что прямоугольник на экране является кубом, потому что я поместил камеру прямо перед ним, и это положение походит на положение у белой стены. Почему это бело? Я не установил никаких цветов, а белый – цвет по умолчанию. Следующая секция устранит это.

Цвета вершин

Когда я создавал VertexBuffer, я упоминал, что Вы можете назначить несколько видов информации для вершины, например цвета. Графические аппаратные средства обрабатывают вершины на конвейере подобно фабрике, собирающей автомобили на сборочной линии. Так и вершина за вершиной проходит через различные шаги обработки, пока каждая не достигает экрана. В этой архитектуре, все данные для каждой вершины должны быть доступны в одно и то же время. Вообразите, как все замедлилось бы, если рабочий, который собирает капот, должен был для различного места каждый раз получить винты.

Рисунок 3 показывает первые пять вершин куба, развернутых в плоскость, и включающих информацию о цвете в(R, G, B) формате. Числа в углах - снова индексы, используемые в полосе треугольников.

Рисунок 3. Полоса треугольников с индексами, цветами вершин, и ориентациями



Если Вы назначаете цвета для вершин то, что случается с пикселами внутри треугольников? Одна возможность состоит в том, чтобы использовать один цвет вершины для целого треугольника; другая - интерполировать цвета между двумя вершинами и создавать цветовой скат(ramp).

M3G позволяет Вам выбирать между обоими вариантами.
В плоско штрихующем режиме(flat-shading mode), Вы используете цвет третьей вершины треугольника для всего треугольника. Если Вы определяете первый треугольник На Рисунке 3 как(0, 1, 2), то цвет будет красный(255, 0, 0). В плавно штрихующем режиме(smooth-shading), каждый пиксел внутри треугольника получает свой собственный цвет интерполяцией. Пикселы между вершинами 0 и 2 начались бы с зеленого цвета(0, 255, 0), который плавно изменяется на красный. Несколько треугольников имеют общие вершины с номерами 2 и 3, что означает, что они также имеют те же цвета, потому что одна вершина может иметь только один цвет.

На Рисунке 3, я также указал порядок, в котором индексы определены. Например,(0, 1, 2) определяет вершины первого треугольника против часовой стрелки(counter-clockwise); второй треугольник(1, 2, 3) определен по часовой стрелке(clockwise). Это называют проветриванием многоугольника*(winding of a polygon)*. Вы это используете для определения стороны - внешней(фронтальной) или изнаночной(обратной). Смотря на куб спереди, Вы могли бы предположить, что Вы всегда видите только наружную сторону, а если бы Вы могли открыть коробку(куб)? Вы хотели бы видеть также внутреннюю(изнаночную) часть стороны. Каждая грань куба имеет две стороны: внешнюю и изнаночную. По умолчанию, порядок вершин против часовой стрелки(counter-clockwise), указывает внешнюю(фронтальную) сторону.

Есть одна небольшая проблема: как показано на Рисунке 3, изменение проветривания в полосе при переходе от треугольника к каждому следующему треугольнику. Соглашение состоит в том, что первый треугольник полосы определяет его проветривание. Когда я обертывал одну полосу треугольника вокруг целого куба в Листинге 1, я начал с треугольника, который определяется против часовой стрелки(0, 1, 2). Этим путем, я неявно определил внешней поверхностью куба фронтальную сторону, а внутренней поверхностью - обратную часть.
В зависимости от ваших потребностей, Вы можете указать M3G рендерить только фронтальные стороны, только изнаночные стороны, или обе вместе. Последнее полезно, если куб имеет полуоткрытую крышку, и Вы можете видеть в одно и то же самое время и внутри и снаружи куба.

Если возможно, Вы должны отменить стороны(disable faces), которые Вам не нужны для просмотра, так как это ускоряет рендеринг. Процесс исключения треугольников из рендеринга называют отбрасыванием(вырезанием,куллингом; *culling*).

Листинг 4 показывает, как использовать цвета вершин.

Листинг 4. Куб с цветами вершин, Часть 1: Инициализация цветов

```
/** Цвета вершин куба(R, G, B). */
 private static final byte[] VERTEX_COLORS = {
   0,(byte) 255, 0,             0,(byte) 255,(byte) 255,
  (byte) 255, 0, 0,            (byte) 255, 0,(byte) 255,
  (byte) 255,(byte) 255, 0,   (byte) 255,(byte) 255,(byte) 255,
   0, 0,(byte) 128,             0, 0,(byte) 255,
 };

 /**
  * Инициализация примера.
  */
 protected void init()
 {
   // Получение экземпляра(singleton)  для 3D рендеринга.
   _graphics3d = Graphics3D.getInstance();

   // Создание данных вершин.
   _cubeVertexData = new VertexBuffer();

   VertexArray vertexPositions =
       new VertexArray(VERTEX_POSITIONS.length/3, 3, 1);
   vertexPositions.set(0, VERTEX_POSITIONS.length/3, VERTEX_POSITIONS);
   _cubeVertexData.setPositions(vertexPositions, 1.0f, null);

   VertexArray vertexColors =
       new VertexArray(VERTEX_COLORS.length/3, 3, 1);
   vertexColors.set(0, VERTEX_COLORS.length/3, VERTEX_COLORS);
   _cubeVertexData.setColors(vertexColors);
```

```
  // Создание треугольников определяющих куб; индексы
  // указывают на вершины в VERTEX_POSITIONS.
  _cubeTriangles = new TriangleStripArray(TRIANGLE_INDICES,
      new int[] {TRIANGLE_INDICES.length});

  // Определить объект Внешность, и установить режим многоугольника.
  // значения по умолчанию: SHADE_SMOOTH, CULL_BACK, и WINDING_CCW.
  _cubeAppearance = new Appearance();
  _polygonMode = new PolygonMode();
  _cubeAppearance.setPolygonMode(_polygonMode);

   // Создание камеры с переспективной проекцией.
  Camera camera = new Camera();
  float aspect =(float) getWidth() /(float) getHeight();
  camera.setPerspective(30.0f, aspect, 1.0f, 1000.0f);
  Transform cameraTransform = new Transform();
  cameraTransform.postTranslate(0.0f, 0.0f, 10.0f);
  _graphics3d.setCamera(camera, cameraTransform);
 }

 /**
  * Рендеринг примера на экран.
  *
  * @param graphics графический объект для черчения.
  */
 protected void paint(Graphics graphics)
 {
  _graphics3d.bindTarget(graphics);
  _graphics3d.clear(null);
  _graphics3d.render(_cubeVertexData, _cubeTriangles,
      _cubeAppearance, null);
  _graphics3d.releaseTarget();

  drawMenu(graphics);
 }
```

В секции членов класса, я определяю каждый цвет вершины в VERTEX_COLORS.
Я поместил цвета в VertexArray в init() и установил ссылку на них в VertexBuffer-е вызовом setColors().
Я также инициализировал объект Внешность(Appearance), названный _cubeAppearance, для использования его _graphics3d.render(), чтобы изменить внешность куба.
Часть _cubeAppearance(объекта Внешность) – объект PolygonMode, который содержит методы изменения атрибутов уровня многоугольника, включая атрибуты показа сторон куба(внешней(фронтальной) или изнаночной(обратной)).
Для интерактивного изменения этих атрибутов, я добавил метод keyPressed(), который Вы видите в Листинге 5.

Листинг 5. Куб с цветами вершин, Часть 2: Обработка нажатий клавиш.

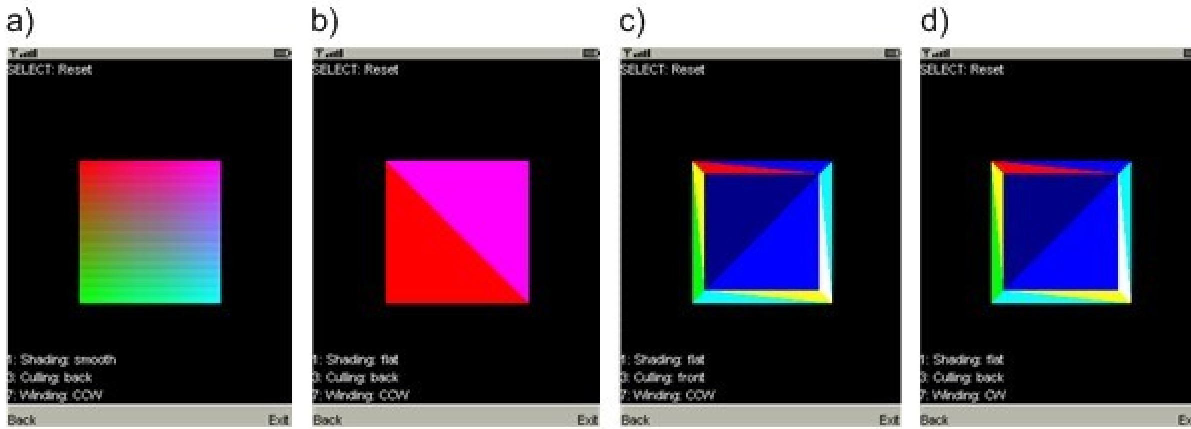
```
/**
  * Обработка нажатий клавиш.
  *
  * @param keyCode код клавиши.
  */
 protected void keyPressed(int keyCode)
 {
  switch(getGameAction(keyCode))
  {
    case FIRE:
      init();
      break;

    case GAME_A:
      if(_polygonMode.getShading() == PolygonMode.SHADE_FLAT)
      {
```

```
      _polygonMode.setShading(PolygonMode.SHADE_SMOOTH);
    }
    else
    {
      _polygonMode.setShading(PolygonMode.SHADE_FLAT);
    }
    break;

  case GAME_B:
    if(_polygonMode.getCulling() == PolygonMode.CULL_BACK)
    {
      _polygonMode.setCulling(PolygonMode.CULL_FRONT);
    }
    else
    {
      _polygonMode.setCulling(PolygonMode.CULL_BACK);
    }
    break;

  case GAME_C:
    if(_polygonMode.getWinding() == PolygonMode.WINDING_CCW)
    {
      _polygonMode.setWinding(PolygonMode.WINDING_CW);
    }
    else
    {
      _polygonMode.setWinding(PolygonMode.WINDING_CCW);
    }

    break;

  // по умолчанию – нет.
 }

 repaint();
}
```

Нажимая соответствующую клавишу, изменяете один из трех атрибутов: режим штриховки(плоская или плавная; flat or smooth), отбор(culling; показывать ли вместе изнаночную и лицевую стороны куба; прим. При только лицевой - уменьшение объемов обработки), и проветривание(треугольники против часовой стрелки это индикаторы лицевой или изнаночной части). Рисунок 4 показывает эти различные варианты. VertexColorsSample.java – полый исходный код этого примера.

Карта соответствия клавиш
Пример использует обработку нажатий игровых клавиш(soft-клавиш) MIDP. Какой физической клавише соответствует игровая клавиша зависит от конкретного устройства, на котором вы выполняете пример. Sun's Java Wireless Toolkit отображает LEFT, RIGHT, UP, DOWN, и FIRE на джойстик. GAME_A – на Кл.1, GAME_B – на Кл.3, GAME_C – на Кл.7, и GAME_D – на Кл.9.

Рисунок 4. Цветной куб: a) Плавная штриховка, b) Плоская штриховка, Изнаночные стороны отброшены, c) Лицевые стороны отброшены, проветривание против часовой стрелки,
d) Изнаночные стороны отброшены, проветривание по часовой стрелке



Преобразования

Вначале, я использовал объект Transform(Преобразование), чтобы переместить камеру назад, так что бы я мог видеть целый куб. Тем же самым путем, Вы можете преобразовать любой 3D объект. Вы можете математически выразить преобразования как матричные операции. Например, вектор - положение камеры, умноженный на правильную матрицу для перевода становится вектором, который перемещается в системе координат. Объект Transform предоставляет такую матрицу. Для самых общих преобразований, M3G обеспечивает три удобных в работе интерфейса, которые скрывают основную математику:

- Transform.postScale(float sx, float sy, float sz): Масштабирует(Scale) 3D-объект в x, y, и z направлениях. Значения больше 1 масштабируют объект этим фактором; значения между 0 и 1 уменьшают его. Отрицательные значения масштабируют и в то же самое время зеркалируют.
- Transform.postTranslate(float tx, float ty, float tz): Перемещает(Translate) 3D-объект, добавляя значения к x, y, и z координатам. Отрицательные значения перемещают объект в направлении отрицательной оси координат.
- Transform.postRotate(float angle, float ax, float ay, float az): Вращает(Rotate) объект с данным углом вокруг оси, которая проходит через(0, 0, 0) и(ax, ay, az). Положительные значения для угла вращают объект по часовой стрелке, если Вы смотрите вдоль положительной оси вращения. Например, postRotate(30, 1, 0, 0) повернет объект на 30 градусов вокруг оси X.

Все имена операций начинаются с " post", что означает, что текущий Transform объект, доумножен справа заданной матрицей преобразования – показывает порядок действий с матрицами. Если Вы поворачиваете на 90 градусов направо и перемещаете два шага, то Вы окажетесь в различной позиции по сравнению с тем, что если бы Вы сначала переместились на два шага и затем повернулись.
Вы можете достигнуть вышеупомянутых инструкций перемещения, вызывая поочередно два post-метода: postRotate() and postTranslate(). Порядок вызова определяет, какой инструкции перемещения Вы достигли.
Из-за постумножения, преобразование Вы используете последним.
M3G имеет класс Transform и интерфейс Transformable. Все API непосредственного режима берут объект Transform в качестве параметра, который изменяет с ним связанный 3D-объект. С другой стороны, Вы используете интерфейс Transformable для преобразования узлов(nodes) – части 3D-мира в сохраненном режиме. Я обсуждаю это во 2 статье этой серии.

В Листинге 6 Вы можете видеть пример, демонстрирующий преобразования.

Листинг 6: Преобразования

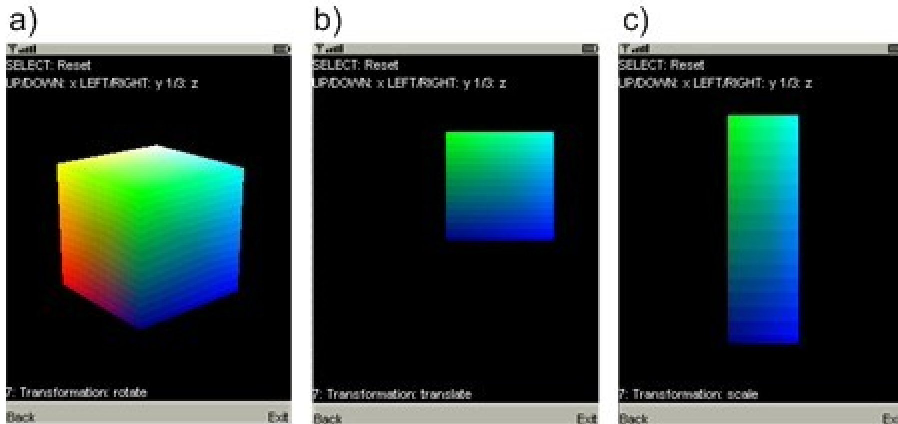
```
/**
 * Рендеринг примера на экран.
 *
 * @param graphics графический объект для черчения.
 */
protected void paint(Graphics graphics)
{
  _graphics3d.bindTarget(graphics);
  _graphics3d.clear(null);
  _graphics3d.render(_cubeVertexData, _cubeTriangles,
     new Appearance(), _cubeTransform);
  _graphics3d.releaseTarget();
```

```
  drawMenu(graphics);
}

/**
 * Обработка нажатий клавиш
 *
 * @param keyCode код клавиши.
 */
protected void keyPressed(int keyCode)
{
  switch(getGameAction(keyCode))
  {
    case UP:
      transform(_transformation, TRANSFORMATION_X_AXIS, false);
      break;

    case DOWN:
      transform(_transformation, TRANSFORMATION_X_AXIS, true);
      break;

    case LEFT:
      transform(_transformation, TRANSFORMATION_Y_AXIS, false);
      break;

    case RIGHT:
      transform(_transformation, TRANSFORMATION_Y_AXIS, true);
      break;

    case GAME_A:
      transform(_transformation, TRANSFORMATION_Z_AXIS, false);
      break;

    case GAME_B:
      transform(_transformation, TRANSFORMATION_Z_AXIS, true);
      break;

    case FIRE:
      init();
      break;

    case GAME_C:
      _transformation++;
      _transformation %= 3;
      break;

    // no default
  }

  repaint();
}

/**
 * Трансформации куба заданными параметрами.
 *
 * @param transformation - трансформация(поворот, перемещение, масштабирование)
 * @param axis - ось перемещения(x, y, z)
 * @param positiveDirection - true для увеличения, false для уменьшения
 *                          значения.
 */
protected void transform(int transformation, int axis,
    boolean positiveDirection)
{
  if(transformation == TRANSFORMATION_ROTATE)
```

```
  {
    float amount = 10.0f *(positiveDirection ? 1 : -1);

    switch(axis)
    {
      case TRANSFORMATION_X_AXIS:
        _cubeTransform.postRotate(amount, 1.0f, 0.0f, 0.0f);
        break;

      case TRANSFORMATION_Y_AXIS:
        _cubeTransform.postRotate(amount, 0.0f, 1.0f, 0.0f);
        break;

      case TRANSFORMATION_Z_AXIS:
        _cubeTransform.postRotate(amount, 0.0f, 0.0f, 1.0f);
        break;

      // по умолчанию - нет
    }
  }
  else if(transformation == TRANSFORMATION_SCALE)
  {
    float amount = positiveDirection ? 1.2f : 0.8f;

    switch(axis)
    {
      case TRANSFORMATION_X_AXIS:
        _cubeTransform.postScale(amount, 1.0f, 1.0f);
        break;

      case TRANSFORMATION_Y_AXIS:
        _cubeTransform.postScale(1.0f, amount, 1.0f);
        break;

      case TRANSFORMATION_Z_AXIS:
        _cubeTransform.postScale(1.0f, 1.0f, amount);
        break;

      // по умолчанию – нет
    }
  }
  else if(transformation == TRANSFORMATION_TRANSLATE)
  {
    float amount = 0.2f *(positiveDirection ? 1 : -1);

    switch(axis)
    {
      case TRANSFORMATION_X_AXIS:
        _cubeTransform.postTranslate(amount, 0.0f, 0.0f);
        break;

      case TRANSFORMATION_Y_AXIS:
        _cubeTransform.postTranslate(0.0f, amount, 0.0f);
        break;

      case TRANSFORMATION_Z_AXIS:
        _cubeTransform.postTranslate(0.0f, 0.0f, amount);
        break;

      // по умолчанию – нет
    }
  }
}
```

paint() - метод теперь имеет объект Transform, _cubeTransform, в качестве четвертого параметра в вызове _graphics3d.render().Измененный метод keyPressed() содержит код для изменения видов трансформаций в интерактивном режиме используя transform().
Клавиша GAME_C переключает между вращением, перемещением, и масштабированием куба. Клавиши UP/DOWN изменяют ось X текущего преобразования, LEFT/RIGHT -ось Y, и GAME_A/GAME_B - ось Z. Нажмите клавишу FIRE(ОГОНЬ), чтобы повторно установить куб в первоначальное положение. Вы можете найти полный исходный код в TransformationsSample.java.

Рисунок 5. Пример куба: a) Поворот, b) Перемещение, и c) Масштабирование



Буфер глубины и проецирование

Используя преобразования, я хочу продемонстрировать два понятия, которые я уже использовал, но не объяснил: *проецирование*, которое определяет, как 3D- объекты отображаются на 2D- экране; *буфер глубины*, который определяет правильный порядок рендеринга объектов, согласно их расстоянию от камеры.
Чтобы видеть прорендированное изображение с точки зрения(расположения) камеры, Вы должны конвертировать 3D- мир к месту расположения камеры, принимая во внимание положение камеры и ориентацию.
В предыдущем примере, я вызывал Graphics3D.setCamera() с объектами Camera и Transform. Вы можете думать о последнем как о трансформации камеры или как указание M3G о том, как конвертировать мировые координаты в координаты камеры - оба определения корректны. Наконец, трехмерные объекты показаны на двухмерном экране. Так, Camera.setPerspective() я указал M3G сделать перспективное проецирование при конвертировании от 3D-пространства в 2D. Перспективное проецирование работает подобно реальному миру: когда Вы смотрите вдаль на длинную, прямую дорогу, то кажется, что границы дороги сходятся в точке на горизонте. Объекты на дороге по мере удаления от камеры, становятся меньше.
Вы можете также игнорировать перспективу и чертить все объекты того же самого размера, независимо от их расположения от камеры. Это может иметь смысл для приложений типа программ автоматизированного проектирования(CAD), где легче работать с чертежами без перспективы. Для выключения перспективного проецирования, замените Camera.setPerspective() на Camera.setParallel().

В пространстве камеры, координата z объекта определяет ее расстояние до камеры.
Если бы Вы рендерите несколько 3D- объектов с различными координатами z, Вы конечно ожидаете, что объекты ближние к камере заслонят отдаленные. Используя буфер глубины, объекты рендерются корректно.
Буфер глубины имеет ту же самую ширину и высоту, что и экран, но содержит координаты z вместо значений цвета.
Он хранит расстояния до камеры для всех пикселов, выводимых на экран. Однако, M3G чертит пиксел только тогда, если пиксел ближе к камере, чем существующий в той же самой позиции. Вы можете проверить это, сравнивая координату z поступающего пиксела со значением Z в буфере глубины.
Таким образом, включенный буфер глубины рендерит объекты согласно их 3D- положения, независимо от порядка вызова команд Graphics3D.render(). Наоборот, если Вы выключаете буфер глубины, Вы должны обращать внимание на порядок, в котором Вы чертите ваши 3D- объекты. Вы включаете или выключаете буфер глубины, когда Вы связываете(bind) Graphics3D с графическим контекстом. Когда Вы используете перегруженную версию bindTarget(), который берет один параметр(только контекст графики), Вы по умолчанию включаете буфер глубины. При использовании bindTarget() с тремя параметрами, Вы можете явно включать или выключать буфер глубины, используя значение булевой переменной в качестве второго параметра.

Вы можете изменить эти два свойства, буфер глубины и проецирование, как показано в Листинге 7:

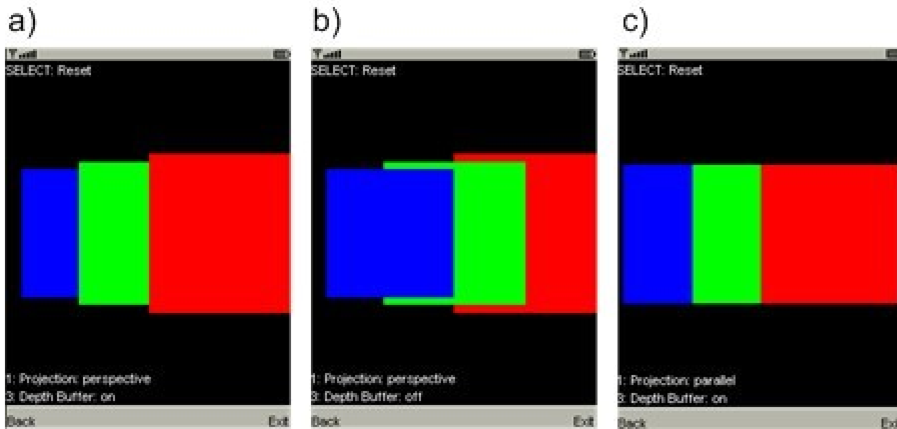
Листинг 7. Буфер глубины и проецирование

```
/**
 * Инициализация примера.
 */
protected void init()
{
   // Получение экземпляра для 3D рендеринга.
  _graphics3d = Graphics3D.getInstance();

  // Создание данных вершин.
  _cubeVertexData = new VertexBuffer();

  VertexArray vertexPositions =
     new VertexArray(VERTEX_POSITIONS.length/3, 3, 1);
  vertexPositions.set(0, VERTEX_POSITIONS.length/3, VERTEX_POSITIONS);
  _cubeVertexData.setPositions(vertexPositions, 1.0f, null);

  // Создание треугольников определяющих куб; индексы
  // указывают на вершины в VERTEX_POSITIONS.
  _cubeTriangles = new TriangleStripArray(TRIANGLE_INDICES,
     new int[] {TRIANGLE_INDICES.length});

  // Создание камер с переспективной и параллельной проекциями.
  _cameraPerspective = new Camera();

  float aspect =(float) getWidth() /(float) getHeight();
  _cameraPerspective.setPerspective(30.0f, aspect, 1.0f, 1000.0f);
  _cameraTransform = new Transform();
  _cameraTransform.postTranslate(0.0f, 0.0f, 10.0f);

  _cameraParallel = new Camera();
  _cameraParallel.setParallel(5.0f, aspect, 1.0f, 1000.0f);

  _graphics3d.setCamera(_cameraPerspective, _cameraTransform);
  _isPerspective = true;

  // Включить Буфер глубины.
  _isDepthBufferEnabled = true;
}

/**
 * Рендеринг примера на экран.
 *
 * @param graphics графический объект для черчения..
 */
protected void paint(Graphics graphics)
{
  // Создание Трансформ-объектов для куба.
  Transform origin = new Transform();
  Transform behindOrigin = new Transform(origin);
  behindOrigin.postTranslate(-1.0f, 0.0f, -1.0f);
  Transform inFrontOfOrigin = new Transform(origin);
  inFrontOfOrigin.postTranslate(1.0f, 0.0f, 1.0f);

  // Включение или выключение Буфера глубины, когда M3G
  //связывается с графическим контекстом.
  _graphics3d.bindTarget(graphics, _isDepthBufferEnabled, 0);
  _graphics3d.clear(null);

   // Чертим все кубы: от ближнего(фронтального)  и далее. Если буфер глубины включен,
  // они будут начерчены согласно их координате z. Иначе,
```

```
  // согласно порядка вызова команд рендеринга.
  _cubeVertexData.setDefaultColor(0x00FF0000);
  _graphics3d.render(_cubeVertexData, _cubeTriangles,
    new Appearance(), inFrontOfOrigin);
  _cubeVertexData.setDefaultColor(0x0000FF00);
  _graphics3d.render(_cubeVertexData, _cubeTriangles,
    new Appearance(), origin);
  _cubeVertexData.setDefaultColor(0x000000FF);
  _graphics3d.render(_cubeVertexData, _cubeTriangles,
    new Appearance(), behindOrigin);

  _graphics3d.releaseTarget();

  drawMenu(graphics);
}

/**
 * Обработка нажатий клавиш.
 *
 * @param keyCode Код клавиши.
 */
protected void keyPressed(int keyCode)
{
  switch(getGameAction(keyCode))
  {
    case GAME_A:
      _isPerspective = !_isPerspective;
      if(_isPerspective)
      {

        _graphics3d.setCamera(_cameraPerspective, _cameraTransform);
      }
      else
      {
        _graphics3d.setCamera(_cameraParallel, _cameraTransform);
      }
      break;

    case GAME_B:
      _isDepthBufferEnabled = !_isDepthBufferEnabled;
      break;

    case FIRE:
      init();
      break;

    // по умолчанию нет
  }

  repaint();
}
```

Используете GAME_A ключ для переключения между перспективным и параллельным проецированием. GAME_B включает или выключает буфер глубины. Вы можете найти полный исходный код в DepthBufferProjectionSample.java. На Рисунке 6, Вы можете видеть эффекты от различных установок.

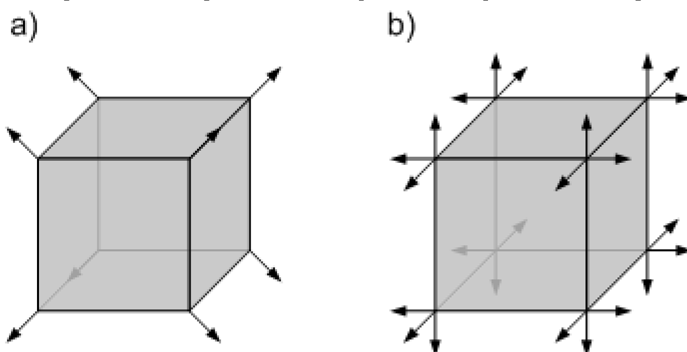
Рисунок 6. Кубы:
a) Порядок рендеринга согласно их расстояний до камеры с включенным буфером глубины,
b) Порядок рендеринга согласно очередности вызова операций рендеринга при выключенном буфере глубины,
c) Рендеринг с параллельным(вместо перспективного) проецированием.



Освещение

В комнате без света, все кажется черным.
Тогда удивительно, что Вы могли видеть что - нибудь в предыдущих примерах, которые не содержали свет.
Цвета вершин и, как Вы будете видеть позже, текстуры, не нуждаются в свете; они будут всегда отображаться в определенном цвете. Однако, свет может изменить их; свет добавляет глубину к сцене.
Направление света, отраженного от объекта зависит от его выравнивания. Если Вы направляете прожектор на зеркало, которое является перпендикулярно к Вам, свет будет отражаться назад к Вам. Если зеркало наклонено, угол падения света будет равен углу отражения.
Общая идея в том, что Вы нуждаетесь в векторе направления, который перпендикулярен к освещенной поверхности. Этот вектор называют *вектором нормали* или коротко *нормалью*.
M3G вычисляет затенение(shading) на основе нормалей, расположений источника света и камеры.
К тому же, нормали – атрибуты вершин, и штриховка пикселей между вершинами или интерполирована(PolygonMode.SHADE_SMOOTH) или взята от третьей вершины треугольника(PolygonMode.SHADE_FLAT).
Поскольку куб имеет восемь вершин, один способ его снабдить нормалями - определить векторы, которые указывают от центра куба к его углам, как показано на Рисунке 7a. Однако, в результате мы получили бы неправильно заштрихованный куб.
Цвет штриховки был бы разделен среди трех сторон, делая грани, невидимыми и делая куб с виду круглым. Что было бы прекрасно для сферы, не работает для куба.
Рисунок 7b показывает, как Вы можете создать жесткие грани, используя 24 нормали, по 4 на сторону. Поскольку одна вершина может только иметь одну нормаль, Вы должны также дублировать вершины.

Рисунок 7. Куб с векторами нормалей: a) 8 вершин, b) 24 вершины(четыре на сторону)



После того, как Вы можете вычислить освещение, используя нормали, Вы должны сказать M3G, какой свет Вы хотите. Свет исходит от источников различных форм: лампочки(bulbs), солнце(sun), прожектор(flashlight) и от других источников. Их синонимы в M3G называются: рассеянный(omnidirectional), направленный(directional), и прожекторный(spotlight).

- Рассеянный(omnidirectional) свет исходит из одной точки и испускается одинаково во всех направлениях. Лампочка без абажура производит такой свет.
- Направленный(directional) свет испускает параллельные лучи в одном направлении. Солнце - так далеко, что Вы можете рассматривать его лучи как параллельные. Направленный свет не имеет координат местоположения, только направление.
- Прожекторный(spotlight). свет сопоставим прожектору или пятну, используемому в театре. Его свет бросает форму конуса и освещает объекты, где конус встречается с поверхностью.

В реальном мире, свет также отражается от объектов и освещает поблизости другие объекты. Если Вы включаете свет в спальне, немного света будет и некоторых местах под кроватью, даже если свет от лампочки напрямую туда не попадает.
Лучи - трассировщики рендерят феноменально реалистические изображения по пути от камеры к источнику света – и требуют феноменально большого времени. Для интерактивных скоростных сменяющихся фреймов нужна простая модель источника света: окружающий(ambient) свет.
Окружающий свет освещает объекты от каждого направления в постоянной доле. Вы могли бы смоделировать предыдущую сцену кровати с окружающим(ambient) светом, чтобы осветить все объекты до некоторой степени и создать дополнительный рассеянный(omnidirectional) свет.
Листинг 8 демонстрирует, как установить различные источники света.

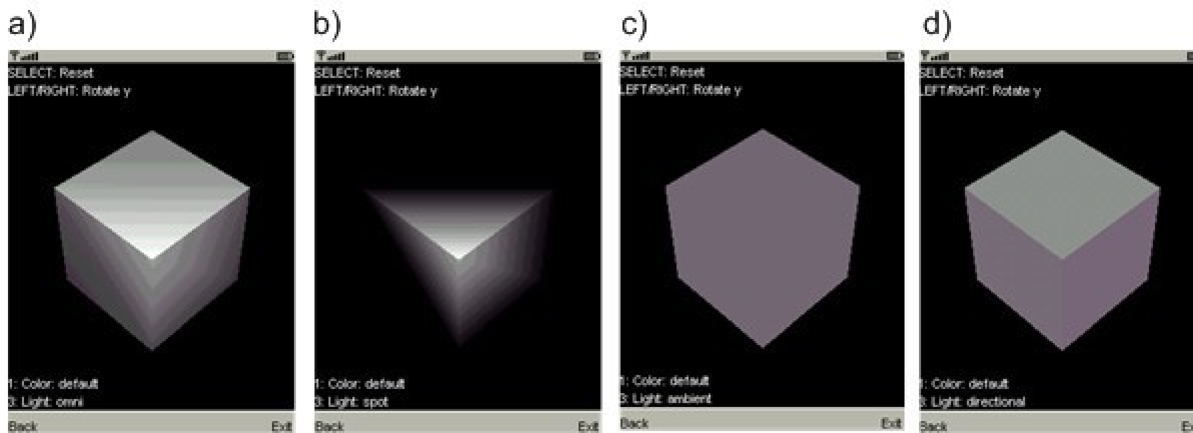
Листинг 8. Установка светового режима

```
// Создание источника света.
   _light = new Light();
   _lightMode = LIGHT_OMNI;
   setLightMode(_light, _lightMode);
   Transform lightTransform = new Transform();
   lightTransform.postTranslate(0.0f, 0.0f, 3.0f);
   _graphics3d.resetLights();
   _graphics3d.addLight(_light, lightTransform);

 /**
  * Установка светового режима.
  *
  * @param light объект Light(Свет), который модифицируется.
  * @param mode вид света.
  */
 protected void setLightMode(Light light, int mode)
 {
   switch(mode)
   {
     case LIGHT_AMBIENT:
       light.setMode(Light.AMBIENT);
       light.setIntensity(2.0f);
       break;

     case LIGHT_DIRECTIONAL:
       light.setMode(Light.DIRECTIONAL);
       light.setIntensity(1.0f);
       break;

     case LIGHT_OMNI:
       light.setMode(Light.OMNI);
       light.setIntensity(2.0f);
       break;

     case LIGHT_SPOT:
       light.setMode(Light.SPOT);
       light.setSpotAngle(20.0f);
       light.setIntensity(2.0f);
       break;

     // по умолчанию нет
   }
 }
```

Вы можете видеть результат освещения каждого типа на Рисунке 8. Куб примера освещен от всех четырех типов света. В каждом случае, свет белый, направлен фронтально от камеры на три стороны куба.

Рисунок 8. Куб освещен светом a) рассеянным(omnidirectional), b) прожекторным(spot),

c) окружающим(ambient), and d) направленным(directional)



Рассеянный свет наиболее силен в вершине, стоящей перед светом и медленно постепенно исчезает. Прожекторный свет, с другой стороны, производит резкое изменение между светом и темнотой, в том месте где пятно конуса заканчивается. Если бы я определил Прожекторный свет с большим конусом, результат был бы подобен рассеянному свету.
Окружающий(ambient) свет освещает куб от каждого направления. Куб выглядит плоским, потому что изображение испытывает недостаток в оттенках.
Наконец, направленный(directional) свет дает каждой стороне различный цвет. В пределах стороны, цвет остается одним и тем же из-за параллельных лучей света.
Освещение не точно; иначе, конус света испускаемый прожектором освещал бы все вокруг.
Это - из-за сложности вычислений освещения, которое из-за малой мощности мобильного телефона упрощено.
Мы могли помочь качеству, добавляя больше треугольников к каждой стороне куба. Хотя треугольники не определяют внешний вид геометрии, они дадут M3G больше контрольных точек(и больше данных для вычисления).

Материалы

Свет может вызывать различные эффекты. Блестящий серебряный шар отражает свет по - другому по сравнению с листом бумаги. M3G моделирует эти характеристики материала следующими атрибутами:

- Окружающее отражение(Ambient reflection): свет, который отражен от окружающего источника света.
- Диффузионное отражение(Diffuse reflection): отраженный свет рассеян одинаково во всех направлениях.
- Эмиссионный свет(Emissive light): объект может испускать свет, подобно пылающим объектам.
- Зеркальное отражение(Specular reflection): свет, который отражается от объектов с блестящей поверхностью.

Вы можете установить цвет для каждого атрибута материала. Цвет диффузного отраженного света от блестящего серебряного шара был бы серебряный и и имел бы зеркальный белый компонент. Чтобы получить заключительный цвет объекта, цвет материала смешивается с цветом света. Если Вы указываете синий свет к серебряному шару, то он отразится синеватым.

Листинг 9 показывает, как использовать материалы:

Листинг 9. Установка материала.

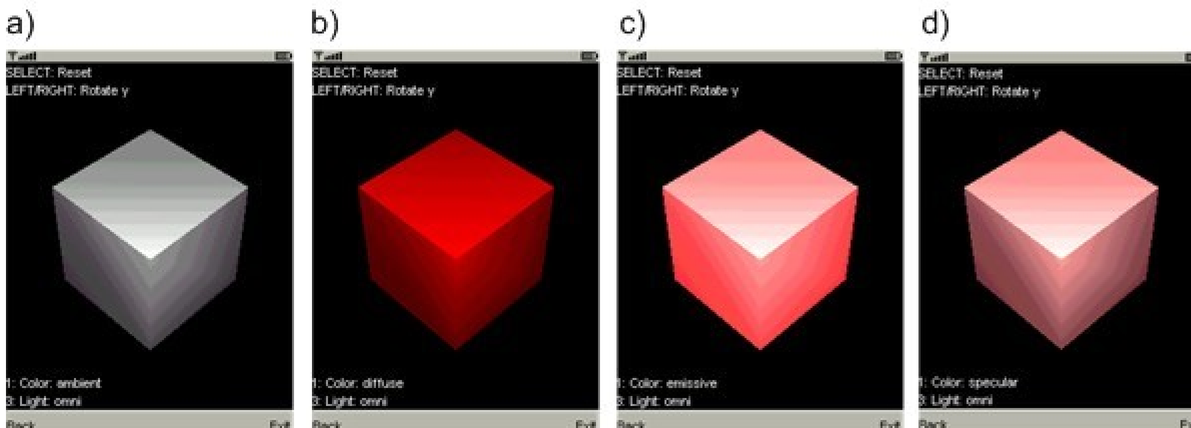
```
// Создание объектов внешности и материала.
   _cubeAppearance = new Appearance();
   _colorTarget = COLOR_DEFAULT;
   setMaterial(_cubeAppearance, _colorTarget);


 /**
  * Установка материала в соответствии с требуемым цветом
  *
  * @param appearance - внешность объектов, что модифицируется.
  * @param colorTarget - требуемый цвет.
  */
 protected void setMaterial(Appearance appearance, int colorTarget)
 {
   Material material = new Material();
```

```
  switch(colorTarget)
  {
    case COLOR_DEFAULT:
      break;

    case COLOR_AMBIENT:
      material.setColor(Material.AMBIENT, 0x00FF0000);
      break;


    case COLOR_DIFFUSE:
      material.setColor(Material.DIFFUSE, 0x00FF0000);
      break;

    case COLOR_EMISSIVE:
      material.setColor(Material.EMISSIVE, 0x00FF0000);
      break;

    case COLOR_SPECULAR:

      material.setColor(Material.SPECULAR, 0x00FF0000);
      material.setShininess(2);
      break;

    // по умолчанию нет
  }

  appearance.setMaterial(material);
 }
```

setMaterial() создает новый объект Материал и устанавливает цвета через setColor() используя соответствующий идентификатор компоненты цвета. Объекту Внешность, используемому в вызове Graphics3D.render(), назначен объект Материал. Хотя здесь не показано, для окружающего и диффузионного отражения, Вы вместо использования Material.setColor(), можете при помощи Material.setVertexColorTrackingEnable() использовать цвета вершин.
И источники света и материалы осуществлены в LightingMaterialsSample.java.
Используя ключи, Вы можете комбинировать различные источники света с материалами, чтобы экспериментировать с эффектами.

На Рисунке 9, показаны материалы с различными характеристиками, используя рассеянный свет. Каждый Снимок устанавливает единичный цветной компонент на красном, чтобы демонстрировать его эффект в изоляции.

Рисунок 9. Отдельные компоненты цвета: a) Окружающий(Ambient), b) Диффузный(Diffuse),
c) Эмиссионный(Emissive), и d) Зеркальный(Specular)

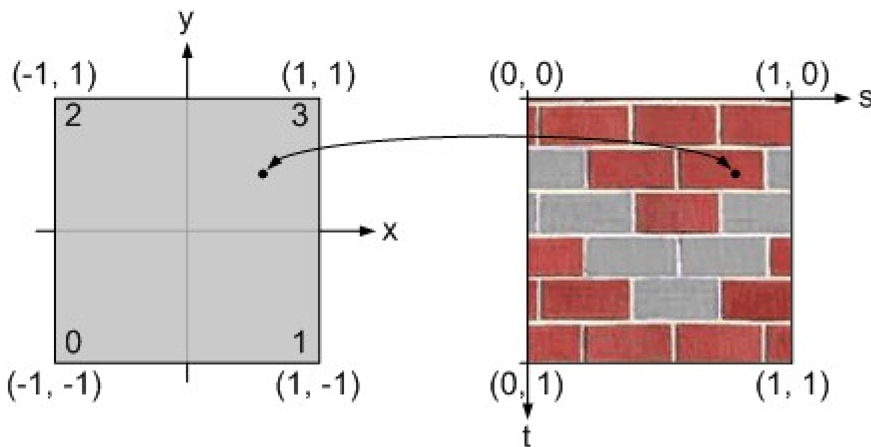


Окружающее отражение только реагирует на окружающий свет; таким образом, нет смысла использовать рассеянный(omnidirectional) свет. Диффузионный компонент материала создает матовую поверхность, в то время как эмиссионный компонент создает пылающий эффект. Зеркальный компонент цвета подчеркивает блеск. Опять же, Вы можете улучшить качество штриховки, используя больше треугольников.

Текстуры

Пока, я показал Вам два способа изменения внешности вашего куба: цвета вершин и материалы. Однако, результат выглядит искусственным.
В реальном мире, Вы имеете очень много деталей. В этом помогают текстуры. Текстуры - изображения, обернутые вокруг 3D- объектов подобно бумаге, обернутой вокруг подарка.
Вы должны правильно выбрать бумагу, подходящую к торжественному случаю и решить, как обернуть(выровнять или под углом). Вы должны выполнить те же действия для 3D-программирования.
Сейчас, Вы вероятно уже предположили, что я введу еще один признак для вершины. Для каждой вершины, координата текстуры определяет, какое положение текстуры будет использоваться. M3G наносит(мапирует) текстуру, заполняя ваш объект.
Вообразите, что Вы прикрепляете к вашему подарку в вершинах эластичную упаковочную бумагу. Пикселы текстуры, на которые эти координаты ссылаются, называют текселями*(texels)*.
Рисунок 10 показывает, как Вы можете нанести квадратную текстуру 128 x 128 текселей на фасад куба.

Рисунок 10: Соотнесение координат многоугольника(x, y) с координатами текстуры(s, t)



Координаты текстуры называют(s, t), чтобы отличить их от(x, y), используемых для позиций вершин(в литературе, вместо этого обычно используется(u, v)). Координаты(s, t) определены так(0, 0) - верхний левый угол текстуры и(1, 1) - нижний правый угол. Соответственно, если Вы хотите нанести нижний левый угол текстуры на нижний левый угол(-1,-1) лицевой стороны куба, Вы должны назначить координаты текстуры(0, 1) к вершине 0(прим.: координаты -1,-1).

Поскольку Вы определяете координаты текстуры относительно углов текстуры, изображения любого размера имеют те же самые координаты(s,t). M3G интерполирует значения между 0 и 1 к самому близкому техселю; например, 0.5 обратился бы к середине текстуры. Если координата текстуры - вне диапазона от 0 до 1, M3G позволяет Вам решать то, что получится. Координаты: - или обернуты вокруг(например, значение 1.5 та же самое как и 0.5), -или значения сжаты, что означает, что любое значение, которое меньше 0 принимается за 0 и любое значение больше 1 принимается за 1. Ширина текстуры и высота могут быть отличны, но должны быть кратными 2, типа 128 на Рисунке 1. Реализация M3G должна поддержать размеры текстуры по крайней мере до 256 - это является одним из дополнительных свойств в M3G.

Graphics3D.getProperties() возвращает Hashtable, заполненный определенными свойствами реализации M3G, типа максимального размера текстуры, или максимальное поддерживаемое число источников света.
Документация к getProperties() содержит список свойств и их минимальных требований.
Перед использованием особенностей, которые превышают эти значения, Вы должны сначала проверить, поддерживает ли реализация M3G вашего устройства выполнение их.
Вы можете видеть использование текстур в Листинге 10.

Листинг 10. Использование текстур, Часть 1: Инициализация

```
/** координаты вершин куба(x, y, z). */
 private static final byte[] VERTEX_POSITIONS = {
   -1, -1,  1,    1, -1,  1,   -1,  1,  1,    1,  1,  1, // фронт
    1, -1, -1,   -1, -1, -1,    1,  1, -1,   -1,  1, -1, // тыл
    1, -1,  1,    1, -1, -1,    1,  1,  1,    1,  1, -1, // справа
   -1, -1, -1,   -1, -1,  1,   -1,  1, -1,   -1,  1,  1, // слева
```

```
  -1,  1,  1,    1,  1,  1,   -1,  1, -1,    1,  1, -1, // верх
  -1, -1, -1,    1, -1, -1,   -1, -1,  1,    1, -1,  1  // низ
 };
/ ** Индексы, которые определяют, как соединить вершины, чтобы строить
  * треугольники. */
private static final int[] TRIANGLE_INDICES = {
    0,  1,  2,  3,   // фронт
    4,  5,  6,  7,   // тыл
    8,  9, 10, 11,   // справа
   12, 13, 14, 15,   // слева
   16, 17, 18, 19,   // верх
   20, 21, 22, 23,   // низ
};
/ ** Длины треугольника полос в TRIANGLE_INDICES. */
  private static int[] TRIANGLE_LENGTHS = {
   4, 4, 4, 4, 4, 4
 };

  / ** название Файла текстуры. */
private static final String TEXTURE_FILE = "/texture.png";

 / ** координаты текстуры(s, t), которые определяют, как нанести
  * текстуру на куб. */
private static final byte[] VERTEX_TEXTURE_COORDINATES = {
   0, 1,   1, 1,   0, 0,   1, 0,   // фронт
   0, 1,   1, 1,   0, 0,   1, 0,   // тыл
   0, 1,   1, 1,   0, 0,   1, 0,   // справа
   0, 1,   1, 1,   0, 0,   1, 0,   // слева
   0, 1,   1, 1,   0, 0,   1, 0,   // верх
   0, 1,   1, 1,   0, 0,   1, 0,   // низ
 };

 /** Первый цвет для смешивания. */
 private static final int COLOR_0 = 0x000000FF;

 /** Второй цвет для смешивания. */
 private static final int COLOR_1 = 0x0000FF00;

 /**
  * Инициализация примера.
  */
 protected void init()
 {

   // Получение экземпляра(singleton)  для 3D рендеринга.
   _graphics3d = Graphics3D.getInstance();

   // Создание данных вершин.
   _cubeVertexData = new VertexBuffer();

   VertexArray vertexPositions =
       new VertexArray(VERTEX_POSITIONS.length/3, 3, 1);
   vertexPositions.set(0, VERTEX_POSITIONS.length/3, VERTEX_POSITIONS);
   _cubeVertexData.setPositions(vertexPositions, 1.0f, null);

   VertexArray vertexTextureCoordinates =

       new VertexArray(VERTEX_TEXTURE_COORDINATES.length/2, 2, 1);
   vertexTextureCoordinates.set(0,
       VERTEX_TEXTURE_COORDINATES.length/2, VERTEX_TEXTURE_COORDINATES);
   _cubeVertexData.setTexCoords(0, vertexTextureCoordinates, 2.0f, null);

   // Установка цвета куба по умолчанию.
```

```
  _cubeVertexData.setDefaultColor(COLOR_0);

  // Создание треугольников определяющих куб; индексы
  // указывают на вершины в VERTEX_POSITIONS.
  _cubeTriangles = new TriangleStripArray(TRIANGLE_INDICES,
     TRIANGLE_LENGTHS);

  // Создание камеры с переспективной проекцией.
  Camera camera = new Camera();
  float aspect =(float) getWidth() /(float) getHeight();
  camera.setPerspective(30.0f, aspect, 1.0f, 1000.0f);
  Transform cameraTransform = new Transform();
  cameraTransform.postTranslate(0.0f, 0.0f, 10.0f);
  _graphics3d.setCamera(camera, cameraTransform);

  // Поворот куба, чтобы видеть 3 стороны.
  _cubeTransform = new Transform();
  _cubeTransform.postRotate(20.0f, 1.0f, 0.0f, 0.0f);
  _cubeTransform.postRotate(45.0f, 0.0f, 1.0f, 0.0f);

  // Определение объекта Внешности и
  //установка режима полигонов(polygon mode).
  _cubeAppearance = new Appearance();
  _polygonMode = new PolygonMode();
  _isPerspectiveCorrectionEnabled = false;
  _cubeAppearance.setPolygonMode(_polygonMode);

  try
  {
    // Загрузка образа текстуры и установка текстуры
   //в объекте Внешности:
    // значения по умолчанию: WRAP_REPEAT, FILTER_BASE_LEVEL/
    // FILTER_NEAREST, and FUNC_MODULATE.
    Image2D image2D =(Image2D) Loader.load(TEXTURE_FILE)[0];
    _cubeTexture = new Texture2D(image2D);
    _cubeTexture.setBlending(Texture2D.FUNC_DECAL);

    // Индекс 0 используется так как имеем только одну текстуру.
    _cubeAppearance.setTexture(0, _cubeTexture);
  }
  catch(Exception e)
  {
    System.out.println("Error loading image " + TEXTURE_FILE);
    e.printStackTrace();
  }
 }
```

В init(), я добавляю координаты текстуры, которые определены как статические члены класса VertexBuffer. Как в примере освещения, я использую четыре вершины на сторону куба и наношу на каждую вершину один угол текстуры. Обратите внимание, что я использую масштабирование 2.0 - третий параметр в _cubeVertexData.setTexCoords().Это говорит M3G о необходимости умножать все координаты текстуры на это значение. В действительности, текстура использует только четверть стороны куба. Я делаю это, чтобы показать Вам особенности M3G: сжатие и обертывание. Если текстура сжата, то она будет начерчена только в верхнем левом углу. Если обертывание текстуры, то она будет начерчена плиточно по целой стороне.

Сама текстура загружена Loader.load() и установлена в объект Texture2D. Вы могли также использовать MIDP-ный Image.createImage(), но класс Loader - самый быстрый путь, если Вы хотите читать структуру из Архива Явы(JAR) файла.

В окончание объект Texture2D установлен как текстура для объекта Внешность(Appearance) для куба.

При использовании текстурирования, Вы можете еще хотеть использовать цвета от освещения или непосредственно назначенные вершинам. Для этой цели, M3G поддерживает различные функции смешивания, которые установлены вызовом в _cubeTexture.setBlending().

В init(), я использую Texture2D.FUNC_DECAL, который смешивает текстуру с основным цветом вершины в зависимости от значения альфа. Серые биты в изображении текстуры На Рисунке 10 на

60 процентов прозрачны. Вместо установки цветов вершин или использовать цвет освещения, я установил цвет по умолчанию для куба с _cubeVertexData.setDefaultColor(), что означает, что все треугольники куба будут иметь тот же самый цвет. Со смешиванием, Вы можете также использовать многократные текстуры на вершине друг друга, чтобы достигнуть дополнительных эффектов.

Я остановлюсь на другой дополнительной особенности M3G. Поскольку Вы видели в секции освещения, качество предоставления зависит от числа треугольников, которые Вы используете - чем меньше расстояние между вершинами, тем лучше интерполяция. То же самое верно для текстур. Высокое качество в контексте к текстурам означает, что текстура наносится без искажения. Текстуры подобно той, что на Рисунке 10 уязвимы, потому что они содержат прямые линии, которые делают искажение очевидным. M3G обеспечивает дешевый путь, в терминах мощности обработки, для решения проблемы. Флаг Дополнительной корректировки перспективы может быть установлен вызовом PolygonMode.setPerspectiveCorrectionEnable(), как вы можете видеть в Листинге 11.

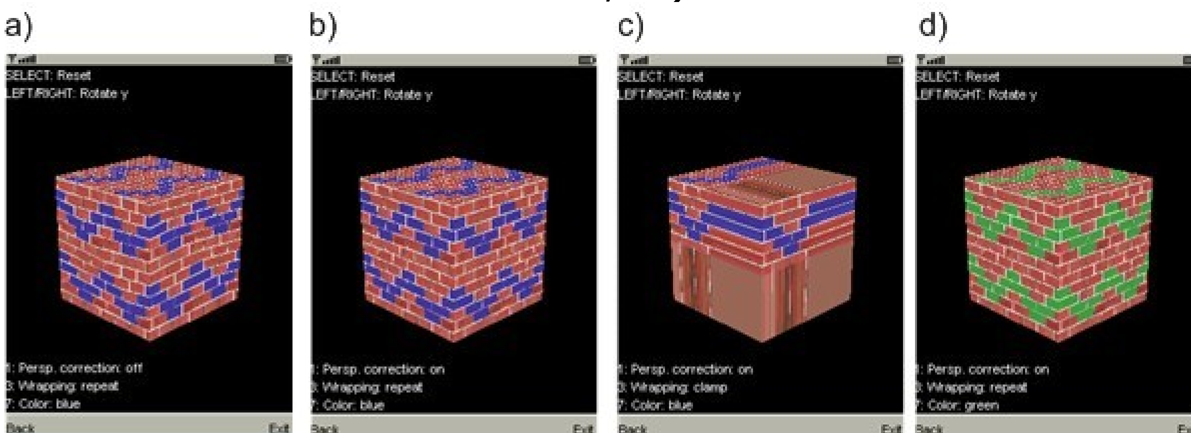
Листинг 11. Использование текстур, Часть 2: Изменение дополнительной корректировки перспективы, режима обертывания, и смешивания цвета

```
/**
  * Проверка поддержки корректировки перспективы.
  *
  * @return true, если поддерживается корректировка перспективы,
  * иначе false.
  */
 protected boolean isPerspectiveCorrectionSupported()
 {
   Hashtable properties = Graphics3D.getProperties();
   Boolean supportPerspectiveCorrection =
      (Boolean) properties.get("supportPerspectiveCorrection");

   return supportPerspectiveCorrection.booleanValue();
 }

 /**
  * Обработка нажатий клавиш.
  *
  * @param keyCode код клавиши.
  */
 protected void keyPressed(int keyCode)
 {
   switch(getGameAction(keyCode))
   {
     case LEFT:
       _cubeTransform.postRotate(-10.0f, 0.0f, 1.0f, 0.0f);
       break;

     case RIGHT:
       _cubeTransform.postRotate(10.0f, 0.0f, 1.0f, 0.0f);
       break;

     case FIRE:
       init();
       break;

     case GAME_A:
       if(isPerspectiveCorrectionSupported())
       {
         _isPerspectiveCorrectionEnabled = !_isPerspectiveCorrectionEnabled;
         _polygonMode.setPerspectiveCorrectionEnable(
             _isPerspectiveCorrectionEnabled);
       }
       break;

     case GAME_B:
       if(_cubeTexture.getWrappingS() == Texture2D.WRAP_CLAMP)
       {
```

```
        _cubeTexture.setWrapping(Texture2D.WRAP_REPEAT,
          Texture2D.WRAP_REPEAT);
      }
      else
      {
        _cubeTexture.setWrapping(Texture2D.WRAP_CLAMP,
          Texture2D.WRAP_CLAMP);
      }
      break;

    case GAME_C:
      if(_cubeVertexData.getDefaultColor() == COLOR_0)
      {
        _cubeVertexData.setDefaultColor(COLOR_1);
      }
      else
      {
        _cubeVertexData.setDefaultColor(COLOR_0);
      }
      break;

    // no default
  }

  repaint();
}
```

В примере, isPerspectiveCorrectionSupported() проверяет, поддерживает ли реализация M3G перспективную корректировку. Если да, то Вы можете в интерактивном режиме в keyPressed() включить или выключить флаг корректировки.
Я также добавил варианты, чтобы изменить порядок текстурирования куба(сжатый или повторяющийся;clamped or repeated;) и опцию изменения смешивания цвета.
Изменение смешивания цвета демонстрирует, как Вы можете легко смешать цвета с текстурами, чтобы создать дополнительные эффекты. Полный пример найдете в TexturesSample.java.
Рисунок 11 показывает текстурирование в различных вариантах.

Рисунок 11. Текстура: a) Без перспективной коррекции, b) С перспективной коррекцией, c) Сжатие вместо плиточного заполнения, и d) Смешивание с зеленым вместо синего



Что следующее?

Я осветил много основ в этой статье, включая детали создания кубов с использованием данных вершин, как снять картинку кубов на камеру, использование источников света и материалов кубов, и как создать реалистически-выглядящие кубы с текстурами. Много кубов.
Кубы – хороши для демонстрации понятий, но они - не показатели сложности 3D проекта.
Во введении, я подчеркнул игровой аспект 3D- графики. Сборка данных вершин вручную, как это сделано в примерах, делает утомительной задачу создания детальных игровых миров
Вы нуждаетесь в технологии: проектировании 3D- сцен в инструменте моделирования и импортировании этих данных в вашу программу. После того, как модель импортирована, Вы должны найти путь организации данных.

С подходом VertexBuffer, Вы должны запоминать все преобразования так же как отношения между объектами.

Предплечье и локоть руки соеденены. Они часть руки и связаны друг с другом. M3G упрощает эту задачу наличием API графа сцены, сохраненного режима, который Вы можете использовать, чтобы моделировать полный мир объектов и его свойств. Это - то, что я буду обсуждать во второй статье этой серии.

Загрузи

Описание	**Имя**	**Рамер**	**Метод загрузки**
Пример кода	wi-m3gsamples1.zip	97 KB	**FTP**

Ресурсы

Обучение

- Понимание особенностей устройств, которые поддерживают M3G на Вебсайтах изготовителей Sony Ericsson, Nokia, или Motorola.
- Серия из 4 частей J2ME 101(developerWorks, November 2003) обеспечивает введение в Java 2 Platform, Micro Edition и Mobile Information Device Profile(MIDP).
- Обучение Java 3D в "Java 3D Joy Ride"(developerWorks, November 2001). Java 3D – технология для Java 2 Platform, Standard Edition.

Получение продуктов и технологий

- Загрузите последнюю спецификацию 3D API:Mobile 3D Graphics API for J2ME(JSR 184).
- Используйте Sun-кий Java Wireless Toolkit версии 2.2 и выше для разработки Ваших M3G приложений.
- Java Wireless Toolkit может быть встроен в Eclipse IDE с помощью плугина EclipseME.
- NetBeans с пакетом Mobility Pack – другая open source IDE оболочка, которая поддерживает разработку Ваших приложений при помощи Java Wireless Toolkit.
- Mobile 3D Viewer от alphaWorks показывает 3D модели в Pocket PC.

Дискуссии

- Участие в форуме обсуждения
- Игры и 3D-связанные вопросы вокруг Явы могут быть обсуждены на Форуме Игр Sun's Java Games Forums.
- Вовлечение в сообщество developerWorks с блогом developerWorks blogs.

Об авторе

Claus Höfele - эксперт по беспроводным приложениям, имеет обширный опыт, работая в промышленности телекоммуникаций. Он - последователь технологии Явы во всех ее инкарнациях. Клаус живет в Токио и с ним можно контактировать по mailto:Claus.Hoefele@gmail.com?cc=.

P.S.

Оригинал статьи **© Перевод, Сергей Кузнецов, 2007**