3D графика для Java мобильных устройств, Часть 2: M3G's Сохраненный режим (retained mode).
Простое управление 3D объектами в графах сцен с помощью JSR 184

Уровень: Промежуточный

Claus Höfele (mailto:Claus.Hoefele@gmail.com?subject=M3G's retained mode), Автор, Внештатный журналист
20 Дек 2005

Mobile 3D Graphics API в сохраненном режиме позволяет Вам работать с графом сцены -представлением вашего 3D мира. Эта статья, вторая из серии (всего две статьи), описывает легкий способ управления Вашими 3D- объектами.

В первой статье серии я объяснил, как Вы можете создать 3D- сцены, **используя Mobile 3D Graphics API (M3G, описанный в JSR 184)в непосредственном режиме** (*immediate mode)* и выводить их непосредственно на экран, отсюда и название режима. Вы можете представить себе непосредственный режим как доступ низкого уровня(low-level access) к 3D-функциям.

Для более сложных задач, полезно иметь в памяти представление Вашего 3D- мира, такое, которое позволило бы Вам управлять данными (3D)структурированным способом. Для M3G, это называется **сохраненным режимом**. В сохраненном режиме, Вы определяете и показываете полный мир 3D- объектов, включая информацию о их внешнем виде. Представьте себе сохраненный режим более абстрактным, но также и более удобным, способом показать 3D-графику.
Сохраненный режим особенно удобен, когда Вы создаете вашу 3D- сцену в инструменте моделирования (на ПК) и импортируете данные в Ваше приложение. В этой статье, я покажу Вам как это делать.

Непосредственный режим против сохраненного режима.

Чтобы показать различия между непосредственным и сохраненным режимами, я хочу использовать часть кода непосредственного режима из первой статьи, в примере сохраненного режима. Вы помните белый куб? В классе, который наследуется от Canvas (Холста), я создавал куб с восьмью вершинами и поместил его в центр системы координат. Я также определил полосу треугольников, которая сказала M3G, как соединить вершины, чтобы строить геометрию куба. Камера с перспективной проекцией сделала снимок куба, который был окончательно отрисован(rendered) в paint(). В этом методе, Graphics3D.render (), вызываемом с параметрами: данными вершины и полосой треугольника.

Изменения для сохраненного режима происходят в init () и paint().Листинг 1 показывает их исполнение.

Листинг 1. Простой куб в сохраненном режиме

```
/**
 * Пример инициализации.
 */
protected void init()
{
  // Получение singleton для  3D рендеринга и создание мира.
  _graphics3d = Graphics3D.getInstance();
  _world = new World();

  // Создание данных вершин.
  VertexBuffer 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  полосы треугольников, что определяют куб;
    // индексы указывают на вершины в VERTEX_POSITIONS.
    TriangleStripArray cubeTriangles = new TriangleStripArray(
        TRIANGLE_INDICES, new int[] {TRIANGLE_INDICES.length});

    // Создание Каркаса(Mesh) куба и добавление к миру.
    Mesh cubeMesh = new Mesh(cubeVertexData, cubeTriangles, new Appearance());
    _world.addChild(cubeMesh);

    // Создание камеры с переспективной проекцией.
    Camera camera = new Camera();
    float aspect = (float) getWidth() / (float) getHeight();
    camera.setPerspective(30.0f, aspect, 1.0f, 1000.0f);
    camera.setTranslation(0.0f, 0.0f, 10.0f);
    _world.addChild(camera);
    _world.setActiveCamera(camera);
  }

  /**
   * Рендеринг примера на экран.
   *
   * @параметр graphics – графический объект для вывода.
   */
  protected void paint(Graphics graphics)
  {
    _graphics3d.bindTarget(graphics);
    _graphics3d.render(_world);
    _graphics3d.releaseTarget();
  }
```

Сначала, я получаю контекст 3D графики и создаю объект Мира, что представляет мою сцену. Инициализация VertexBuffer и TriangleStrip та же самая, что и в непосредственного режиме, но вместо того чтобы их сразу рендерить , я их включаю в Mesh. Мир имеет двух детей: каркас(mesh), который я только, что создал и камеру с переспективной проекцией. Метод _world.setActiveCamera() говорит M3G о необходимости использовании этой добавленной к Миру камеры для рендеринга. Метод paint() становится очень простым – рендеринг Мира в контекст графики. Полный листинг класа найдете в VerticesRetainedSample.java.
M3G работает в сохраненном режиме когда вы используете Graphics3D.render() с параметром Мир(World). Вы видите в Листинге 1, сохраненным режимом просто управлять как и непосредственным режимом, но каждый режим использует немного различные вызовы API. В Таблице 1 приведен список различий.

Таблица 1. API вызовы, используемые в непосредственном и сохраненном режимах

	Непосредственный режим	**Сохраненный режим**
Camera	Установка текущей камеры в Graphics3D перед вызовом рендеринга. • Graphics3D.setCamera(Camera camera, Transform transform)	Добавление одной или более Камер к Миру и активизация одной них. • World.addChild(Node child) • World.setActiveCamera(Camera camera)
Light	Добавление, замена или модификация источника света в Graphics3D перед вызовом рендеринга, очистка массива источников. • Graphics3D.addLight(Light light, Transform transform) • Graphics3D.setLight(int index, Light light, Transform transform) • Graphics3D.resetLights()	Добавление источника Света. • World.addChild(Node child)

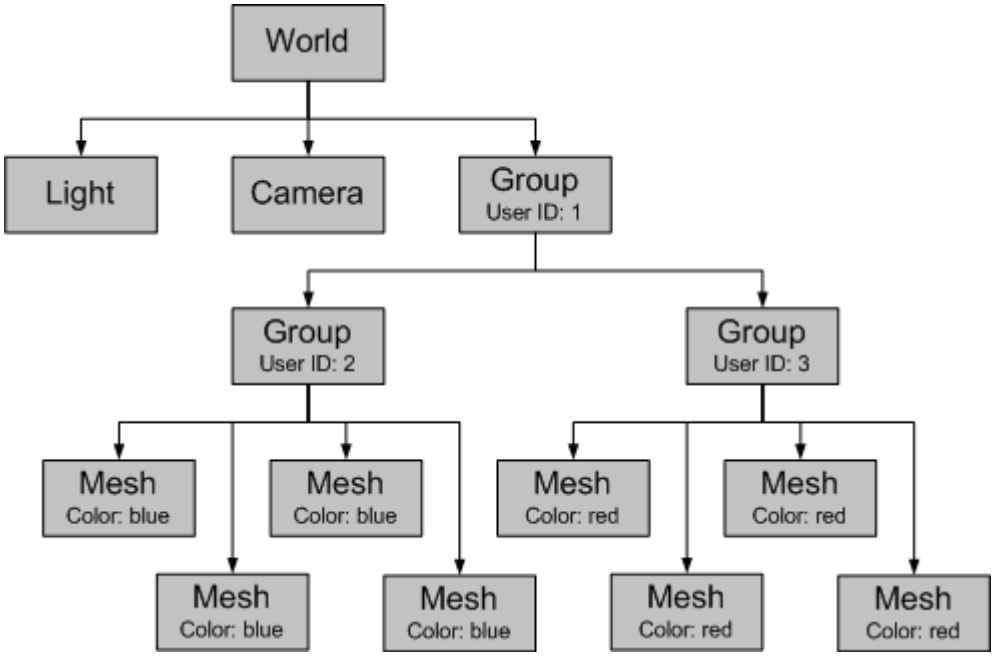
Transformation	Передача Transform объекта при вызове API, например: • Graphics3D.render(Node node, Transform transform)	Используем методы унаследованные от Transformable класса, например: • Node.setTransform(Transform transform)
Background	Установка Фона вызовом метода Graphics3D.clear(). Всегда вызывайте clear() перед рендерингом. • Graphics3D.clear(Background background)	Установка Фона в объекте Мира. Если не установлен то, фон очищается автоматически черным. • World.setBackground(Background background)
Rendering	Рендеринг узлов графа сцены, (Sprite3D, Mesh, Group, и их подклассов), и частей каркасов(submeshes) (VertexBuffer). • Graphics3D.render(Node node, Transform transform) • Graphics3D.render(VertexBuffer vertices, IndexBuffer triangles, Appearance appearance, Transform transform, int scope) • Graphics3D.render(VertexBuffer vertices, IndexBuffer triangles, Appearance appearance, Transform transform)	Рендеринг всего Мира, включая его детей. • Graphics.render(World world)

Вы можете комбинировать сохраненный и непосредственный режимы, смешивая соответствующие команды рендеринга. После использования Graphics3D.render(World), активная Камера и Свет из рендеруемого Мира автоматически заменяют текущие Камеру и Свет в Graphics3D объекте. Таким образом при последующем рендеринге в непосредственном режиме вы можете использовать тоже окружение. Вы можете также начертить (отрендерить) Мир в непосредственном режиме вызовом Graphics3D.render(Node, Transform) т.к. Мир (World) наследуется из узла.
В этом случае, Свет, Камеры, и Фон Мира игнорируются, но все другие дочерние узлы чертятся.
M3G's спецификация определяет сохраненный режим основываясь на типе параметра при вызове render(), что вы используете при черчении. API работает в сохраненном режиме когда Вы используете render(World).
Поскольку Вы можете рендерить Мир и его дочерние объекты как Узел в непосредственном режиме, то нет большого различия между любыми режимами.
Однако, обычно говоря о сохраненном режиме имеют в виду способность легко управлять группой 3D-объектов в структуре данных названной графом сцены, независимо от кода рендеринга.

Граф сцены

Преимущества графа сцены становятся очевидными, когда Вы определяете более сложный мир. В M3G, класс World(Мир) организовывает 3D- объекты в структуре дерева, узлы которого - классы, наследованные из Node(Узла).
Вы уже видели два подкласса Node(Узла) в предыдущем примере: Camera(Камеру) и Mesh (Каркас). Другие обычно используемые объекты включают Light(Свет), который освещает 3D-сцену, и Group(Группу), которая действует как контейнер для других узлов. Сгруппированные узлы можно рассматривать как единый объект.
Сам Мир – это Группа с дополнительными возможностями и используется как контейнер верхнего уровня для графа сцены.
Рисунок 1 показывает дерево мира, который включает Свет, Камеру, и несколько Каркасов, организованных в Группах. Я также включил определенные значения в узлы Группа и Каркас, их я объясню позже.

Рисунок 1. Пример дерева графа сцены



В графе сцены для M3G, один Node (Узел) должен принадлежать только одной Group (Группе). Граф должен также быть нециклическим. Вы свободны в моделировании вашего графа, который соответствует вашей структуре данных. Для моего примера, я решил иметь две Группы, четыре Каркаса в каждой. Первая синяя Группа содержит синие Каркасы, вторая красная Группа - красные Каркасы. Другая Группа - контейнер и для синих и для красных Групп.

Пользовательские Идентификаторы(ID), которые я назначил Узлам Групп, позже помогут мне идентифицировать их. Любой Узел может иметь пользовательский ID. Для моих целей, достаточно иметь возможность находить Узлы Групп. Я добавил узлы Свет и Камеру к корневому Объекту, но их местоположение не имеет значения. Вы можете добавить много камер, чтобы определить различные точки обзора, и переключать их в активное состояние с помощью World.setActiveCamera().Вы можете также иметь сразу несколько источников Света. (Максимальное число источников Света, поддержавших реализацией M3G, может быть получено с Graphics3D.getProperties ()).

Листинг 2 показывает инициализацию мира, показанного на Рисунке 1.

```
/**
 * Пример инициализации.
 */
protected void init()
{
 // Получение singleton для  3D рендеринга и создание мира.
     _graphics3d = Graphics3D.getInstance();
     _world = new World();

 // Создание камеры с переспективной проекцией.
  Camera camera = new Camera();
  float aspect = (float) getWidth() / (float) getHeight();
  camera.setPerspective(30.0f, aspect, 1.0f, 1000.0f);
  camera.setTranslation(0.0f, 0.0f, 10.0f);
  _world.addChild(camera);
  _world.setActiveCamera(camera);

  // Создание источника Света.
  Light light = new Light();
  light.setMode(Light.OMNI);
  light.setTranslation(0.0f, 0.0f, 3.0f);
  _world.addChild(light);

  // Создание двух наборов данных вершин: один для синих Каркасов
```

```
    // и один для красных Каркасов.
    VertexBuffer blueCubeVertexData = new VertexBuffer();
    blueCubeVertexData.setDefaultColor(0x000000FF); // синий
    VertexBuffer redCubeVertexData = new VertexBuffer();
    redCubeVertexData.setDefaultColor(0x00FF0000);  // красный

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

    VertexArray vertexNormals =
        new VertexArray(VERTEX_NORMALS.length/3, 3, 1);
    vertexNormals.set(0, VERTEX_NORMALS.length/3, VERTEX_NORMALS);
    blueCubeVertexData.setNormals(vertexNormals);
    redCubeVertexData.setNormals(vertexNormals);

    // Создание  полосы треугольников, что определяют куб;
    // индексы указывают на вершины в VERTEX_POSITIONS.
       TriangleStripArray cubeTriangles = new TriangleStripArray(
        TRIANGLE_INDICES, TRIANGLE_LENGTHS);

    // Создание материала.
    Material material = new Material();
    material.setVertexColorTrackingEnable(true);
    Appearance appearance = new Appearance();
    appearance.setMaterial(material);

    // Создание групп для группирования кубов.
    Group allMeshes = new Group();
    allMeshes.setUserID(USER_ID_ALL_MESHES);
    _world.addChild(allMeshes);
    Group blueMeshes = new Group();
    blueMeshes.setUserID(USER_ID_BLUE_MESHES);
    allMeshes.addChild(blueMeshes);
    Group redMeshes = new Group();
    redMeshes.setUserID(USER_ID_RED_MESHES);
    allMeshes.addChild(redMeshes);

    // Создание каждого куба в цикле.
    for (int i=0; i<8; i++)
    {
      Mesh cubeMesh = null;

      if ((i%2) == 0)
      {
        cubeMesh = new Mesh(blueCubeVertexData, cubeTriangles, appearance);
        blueMeshes.addChild(cubeMesh);
      }
      else
      {
        cubeMesh = new Mesh(redCubeVertexData, cubeTriangles, appearance);
        redMeshes.addChild(cubeMesh);
      }

      cubeMesh.setTranslation(1.5f * (float) Math.cos(i*Math.PI/4), 1.5f * (float)
        Math.sin(i*Math.PI/4), 0.0f);
      cubeMesh.setScale(0.4f, 0.4f, 0.4f);
    }
  }
```

Вначале - обычная установка Камеры и Света, которые я добавил к Миру. Для Каркасов, я создаю два VertexBuffers, один по умолчанию с синим цветом и другой по умолчанию с красным цветом. Хотя они - отдельные объекты, они имеют общее: положения вершин и

нормалей, что позволяет экономить память. Я также создаю материал, который необходим для освещенности и позволяет прослеживать цвета вершин по умолчанию и использовать их в вычислении освещенности Каркасов. Затем, я создаю узлы Групп и в цикле создаю восемь Каркасов и располагаю их по кругу. Каждый четный Каркас – синий и добавляется к синей Группе, каждый нечетный Каркас – красный и добавляется к красной Группе. Каждой Группе назначен пользовательский ID; Листинг 3 показывает, как их использовать.

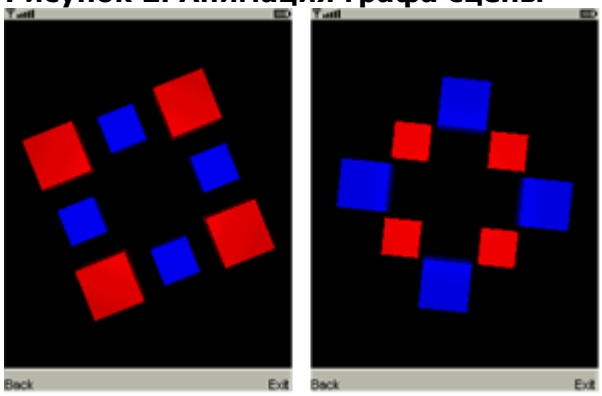
Листинг 3. Дерево графа сцены, часть 2: Анимация

```
/**
 * Пример показывает как инициализируется
 * и стартует Нить  анимации.
 */
public void showNotify()
{
  init();

  Thread thread = new Thread(this);
  _isRunning = true;
  thread.start();
}

/**
 * Ведение анимации.
 */
public void run()
{
  while(_isRunning)
  {
    // Поворот Каркасов.
    Group allMeshes = (Group) _world.find(USER_ID_ALL_MESHES);
    allMeshes.postRotate(2, 0.0f, 0.0f, 1.0f);

    _counter++;
    if ((_counter%COUNTER_MAX) == 0)
    {
      _scale *= -1;
    }

    // Масштабирование синих Каркасов.
    Group blueMeshes = (Group) _world.find(USER_ID_BLUE_MESHES);
    blueMeshes.scale(1 + _scale, 1 + _scale, 1 + _scale);

    // Масштабирование красных Каркасов.
    Group redMeshes = (Group) _world.find(USER_ID_RED_MESHES);
    redMeshes.scale(1 - _scale, 1 - _scale, 1 - _scale);

    repaint();

    try
    {
      Thread.sleep(50);
    }
    catch (Exception e){}
  }
}
```

Этот Пример выполняет Thread(Нить) кода, чтобы создать простую анимацию, с Transform(Преобразованием) Узлов Групп. Преобразования в сохраненном режиме работают немного по-другому по сравнению с непосредственным режимом. В непосредственном режиме Преобразование представляет одиночную матрицу, которая манипулирует 3D-объектом. В сохраненном режиме, вместо этого используются методы объекта Узла(Node), унаследовавшего их от класса Transformable(Преобразуемый). Класс Преобразуемый имеет четыре компонента преобразования: translation (T; перемещение), orientation (R; поворот), scale (S;

масштабирование), и generic 4x4 matrix (M; основное 4x4 матричное). Все четыре компонента могут быть установлены одновременно и вместе выполняют финальное преобразование. Вектор p, координаты вершины, преобразован в p', используя уравнение p' =TRSMp.
Самый легкий способ перевести преобразование из непосредственного режима в сохраненный режим состоит в том, чтобы использовать Transformable.setTransform () и таким образом изменять основную матрицу. Иначе, Вы можете использовать сохраненные три компонента, но обратите внимание, что порядок матричного умножения фиксирован и не может быть изменен. Преобразования также применимы к Группам Узлов.
Группировка - легкий способ обрабатывать несколько Узлов как один единый объект.
В Листинге 3, все Каркасы вращают вокруг оси Z, но синие Каркасы масштабированы по-другому по сравнению с красными Каркасами. Преобразование Каркаса определено ее собственным преобразованием и преобразованием всех его родителей. Два скриншота анимации можно посмотреть в Рисунке 2; полный исходный код находится в SceneGraphSample.java.

Рисунок 2. Анимация графа сцены



M3G's двоичный формат
Создание 3D- объектов вручную утомительно, и это объясняет, почему я использовал до сих пор простые кубы. В этой секции, я покажу Вам, как использовать 3D инструмент моделирования, чтобы создать 3D мир и затем импортировать его в приложение M3G.
Обмен данными между приложениями требует общего формата файла. Вместо того, чтобы изобретать ваш собственный, Вы можете использовать M3G's встроенный двойной формат, который ваше приложение может читать с помощью Loader.load ().
Много инструментов могут уже сохранять миры в формате M3G файлов, или Вы можете добавить эту функцию экспорта в формате M3G к инструментам третьих лиц. Из многих 3D инструментов, которые существуют, я выбрал Blender, который является мощным, open source пакетом моделирования. Blender - бесплатен, и Web содержит множество хорошей документации. Важно, что Blender может быть расширен скриптами, написанными на языке Python. Я нашел два скрипта, которые пишут M3G's формат файла. Я использовал скрипт, написанный Gerhard Völkl, потому что он имеет больше функциональности и доступен согласно лицензии GPL.
Resources – ссылки на упомянутые инструменты.

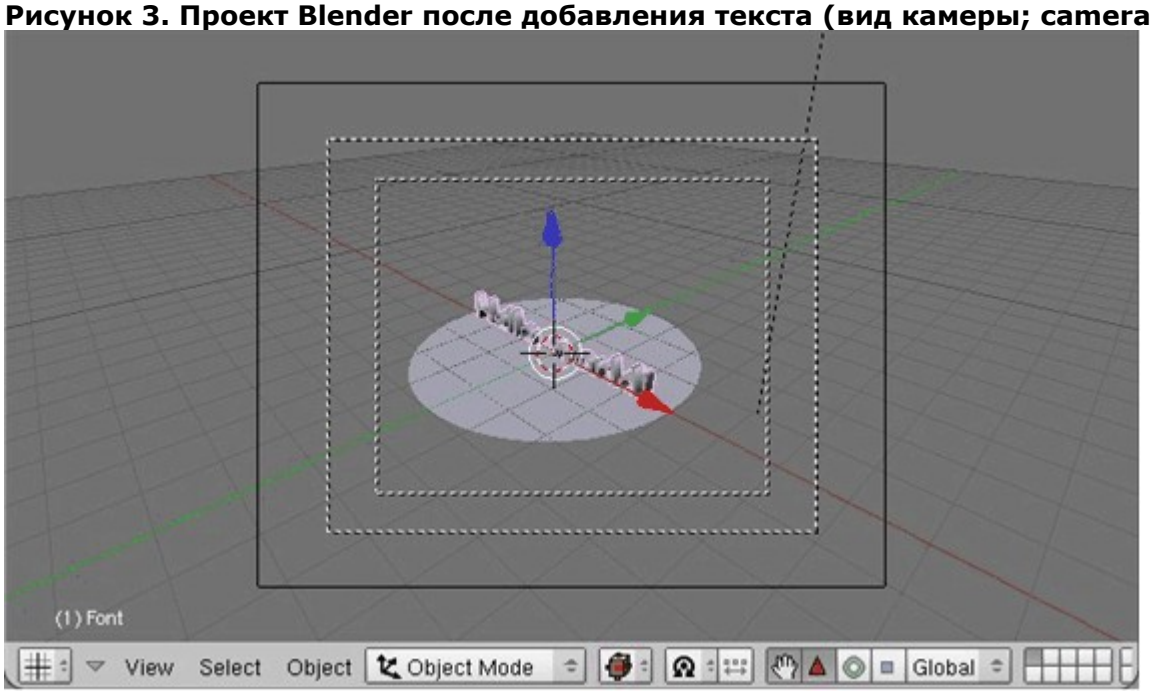
Сцена, которую я хочу создать – 3D-текст, который вращается вокруг начала системы координаты. Вы можете разгрузить законченный проект Blendera из секции Sample code, но вот - перечень шагов, чтобы создать модель самостоятельно:

1. Создай новый файл(Ctrl-X), удали куб, что Blender создал по умолчанию.
2. Помести курсор в начало системы координат и захвати курсором сетку в фронтальном обзоре(front view) и представлении стороны(side view) (Shift-S-3). Курсор - теперь точно в начале координат.
3. В фронтальном обзоре, добавьте текст (**Add > Text**), который создает редактируемый объект шрифта. Введи любой текст; я ввел "Hello, world!".
4. Переключи режим объекта(Tab) и увидишь контекст **Editing**(F9; Редактирования). Щелкни кнопку **Center New (Новый центр)** на панели **Curve and Surface (Кривая и Поверхность)**. Текст теперь отцентрирован горизонтально относительно начала координат.

5. В той же самой панели, введи 0.100 в поле **Ext1**, для вытеснения глубины(толщины букв; extrudes the depth) текста и получишь 3D –вид текста.
6. В верху фронтального обзора добавьте круг Безье(**Add > Curve** > **Bezier Circle;**), это ортогонально к тексту. Я буду использовать этот круг позже, чтобы согнуть текст, так что бы он следовал за дугой круга.
7. Круг выбран по умолчанию; увеличьте его(введитеS для масштабирования).
8. Blender создает свет и камеру по умолчанию. Оставьте их без изменения.

В этом пункте вы имеете модель мира, показанную на Рисунке 3.

Рисунок 3. Проект Blender после добавления текста (вид камеры; camera view)

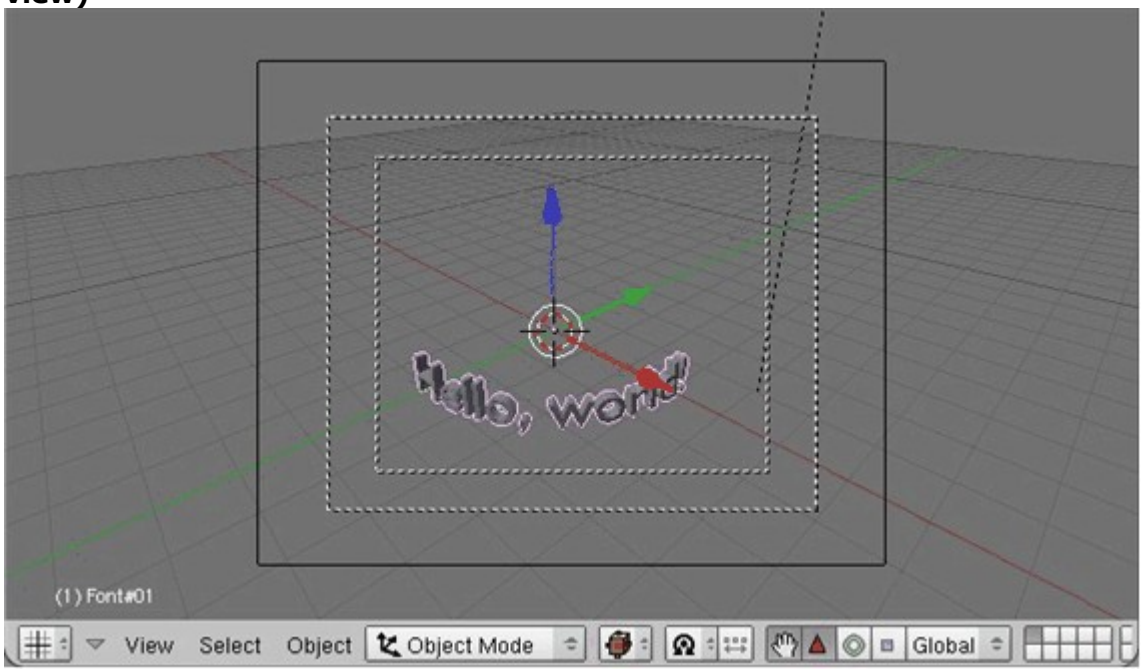


Поскольку скрипт экспорта может только обращаться с Каркасами(meshes), я должен конвертировать объект шрифта. Впоследствии, я согну результирующий Каркас вокруг круга.

1. В Object Mode (режиме объекта;переключается Tab), выберите текст правым щелчком мыши и Alt-C конвертируйте тип объекта. Сначала Font(Фонт) в Curve(Кривую), затем Curve конвертируйте в Mesh (Каркас).
2. Выберите текст, перейдите в контекст редактирования(**Editing**;F9) и в панели **Links and Materials(Ссылки и материалы)** с помощью кнопки **New** создайте новый материал со значением по умолчанию. Каркас должен иметь материал, иначе скрипт экспорта выдаст ошибку.
3. Выделите текст, измените выбираемый круг, и создайте родительско-дочернее отношение с помощью Ctrl-P. В открытом списке вариантов, выбери **curve deform(деформация кривой)**.
4. С выделенным текстом, перейдите в контекст **Object** (F7). Выбери **Track X(Дорожка X)** панели установки анимации. Это выбирает ось x для выравнивания между текстом и кругом. Теперь текст приятно изогнут в форме круга.
5. Текст теперь стал изогнутым по кругу. Трансформируйте текст по вашему выбору(вводя G для перемещения и R для вращения).
6. Используйте Ctrl-Shift-A для текста для добавления деформации к Каркасу и удаление круга за дальнейшей ненадобностью. Если виды текста сильно искажены после удаления круга, то нажмите дважды Tab для переключения между режимами объекта и редактирования.
7. Цвет фона может быть изменен в панели **World** (Мир; переключением в контекст **Shading**(Тени) с F5 выбором кнопки **World**).
8. В панели **Links and Materials** контекста **Editing**(F9), переименуйте имя текстового объекта в Font#01. Число в имени Каркаса будет пользовательским ID Каркаса.

Рисунок 4 показывает результат. Если Вы хотите иметь предварительно просмотреть модель, Blender может прорендерить ее с помощью клавиши F12.

Рисунок 4. Проект Blender после изгиба текста в круглой форме (вид камеры; camera view)



Заключительный шаг должен экспортировать 3D модель в M3G файл.

1. Установите скрипт экспорта копированием в папку .blender\scripts вашего установленного Blender.
2. Скрипт использует несколько возможностей импорта(операторов Python), которые установленный по умолчанию Python вместе с Blender-ом не имеет. Вы можете решить эту проблему инсталлируя отдельно интерпретатор Python. Вместо этого, я закомментировал несколько операторов импорта Python и скрипт работал прекрасно.
3. После перезагрузки Blender-а, загрузите вашу 3D модель и выполните скрипт выбором **File > Export > M3G in J2ME**.
4. Выберите **Export into *.m3g binary file**. Скрипт может также сгенерировать исходный код Java™- эквивалент вашей 3D модели. Это удобно если Вы в последующем хотите в ручную внести изменения. В моем случае, двоичный файл –наилучший выбор.
5. Выберите имя и папку для M3G - файла, который вы хотите сохранить и нажмите кнопку **Save M3G J2ME**.

Итак все. Вы экспортировали вашу модель. Давайте используем ее в приложении(мидлете). Я буду использовать структуру подобную предыдущему примеру. Листинг 4 показывает изменения в init() и в run().

Листинг 4. Чтение M3G двоичного файла

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

    try
    {
      // Загрузка Мира из M3G двоичного файла.
      Object3D[] objects = Loader.load(M3G_FILE_NAME);
      _world = (World) objects[0];

      // Изменение свойств камеры на соответствие текущему устройству.
      Camera camera = _world.getActiveCamera();
      float aspect = (float) getWidth() / (float) getHeight();
```

```
    camera.setPerspective(60.0f, aspect, 1.0f, 1000.0f);
   }
   catch (Exception e)
   {
    e.printStackTrace();
   }
  }

  /**
   * Ведение анимации.
   */
  public void run()
  {
   while(_isRunning)
   {
    // Найти Каркас и повернуть его.
    Mesh text = (Mesh) _world.find(USER_ID_TEXT);
    text.postRotate(-2.0f, 0.0f, 0.0f, 1.0f);
    repaint();

    try
    {
     Thread.sleep(50);
    }
    catch (Exception e){}
   }
  }
```

Приложение стало действительно простым. В init(), я читаю M3G файл, экспортированный из Blender-а и помещаю содержимое в объект World (Мир). Я это могу сделать потому, что я знаю структуру которую скрипт произвел. Только одно ручное изменение в установке камеры. Отношение аспекта зависит от размера экрана конкретного телефона и должен вычисляться во время выполнения. В методе run(), я нахожу Mesh(Каркас) текста по его пользовательскому ID; По этой причине я так назвал текстовый Объект в Blender-е. Вращение применено к объекту Каркас, потому что сам Мир не может быть преобразован.

Вы должны быть осторожны с осями координат. Если Вы берете вашу правую руку и указываете направо ось X (ваш большой палец), ось Y Blender's (указательный палец) указывает от вас и ось Z (средний палец) - вверх. M3G's X ось та же самая а, ось У указывает вверх и ось Z указывает на Вас. (Смотри в части 1 описание координатной системы M3G's coordinate system.) Обе системы "правой руки", но повернуты на 90 градусов относительно оси X. Поскольку Камера и Каркас текста экспортировались, используя систему координат Blender-а – текст прекрасно показывается. Однако, если Вы хотите применить ваши собственные преобразования, Вы должны изменить оси. Именно поэтому в Листинг 4, вращение - вокруг оси Z.

Рисунок 5 показывает скриншот из законченного приложения; BinaryWorldSample.java показывает полный исходный код.

Рисунок 5. Модель Blender-а импортированная в M3G приложение

Большое преимущество файлов, экспортируемых из 3D-инструментов состоит в том, что Вы можете разделить работу между разработчиками и художниками. Когда Вы создаете более сложную модель в Blender-е, скоро поймете, что к навыку на разработку программного

обеспечения нужен дополнительный художественный навык. В коммерческих играх для консолей и PC, большинство трудовых ресурсов и денег потрачено на художественные работы. Для мобильных телефонов, модель не только должна выглядеть хорошей, но также и должна быть маленькой. Файл, экспортируемый из Blender-а - 48 КБ; большинство места используется для вершин, нормалей, и определений полосы треугольника. Если Вы проанализируете файл, Вы можете видеть, что модель имеет 3 392 вершины с 1 364 треугольниками, определенными в 1 014 полосах треугольника.
Вместе с представленным освещением, это - тяжелая работа по обработке для вашего среднего мобильного телефона. Если Вы выключите освещение, Вы не нуждаетесь в нормалях и можете сэкономить на размере файла и требуемой мощности обработки. M3G's файл поддерживает сжатие, но без этого вы уже более обеспечены сжатием, потому, что M3G файл будет сжат так или иначе, когда он добавлен через ресурсы к JAR файлу Java-приложения. В примере приложения, M3G – файл сжат до 16 КБ, как часть архива. Blender также содержит инструменты, которые позволяют Вам уменьшать число вершин вашего Каркаса с минимальным воздействием на форму (используйте инструмент **Decimator** в панели **Mesh** контекста **Editing**).

Чтобы использовать Blender для коммерческого проекта, Вы должны были бы затратить некоторое время в улучшение скрипта экспорта, который не поддерживает все особенности M3G's и разборчив в способе, которым Вы моделируете ваши 3D- сцены. Это помогло бы сделать инвестиции в несвободный заслуживающий внимания пакет. С другой стороны, скрипт доступен под GPL(прим. Лицензией), так, что любой может улучшить его. Автор скрипта недавно добавил поддержку текстур. К счастью, кто - то уже инвестировал время в обеспечение свободного инструмента.

Выравнивание и выбор
После того, как модель импортирована в ваше приложение, Вы можете применить все возможности M3G. Я покажу Вам, два метода общего использования возможностей M3G в соединении с сохраненным режимом - выравнивание Узлов и Выбор.
Когда Вы кладете мяч для первого удара в игре гольф, Вы хотели бы, чтобы камера всегда показывал шар в середине экрана, и его путь полета к земле. С Node.setAlignment(), узел, типа камера, может автоматически ориентироваться на другой объект, например мяч для гольфа. В следующем примере, я буду использовать это, чтобы преобразовать текст, так что он непосредственно стоит перед камерой.
Другая общая проблема в играх выбор объекта, с вызовом выбора(*picking)*. Когда Вы проектируете стрелка и запускаете пулю, Вы должны проверить, поражает ли она цель. Group.pick() бросает луч, который Вы снабжаете точкой отсчета и направлением. Если луч пересекается с Каркасом, метод возвращает истину и сообщает о поражении Каркаса - объекта. Я буду использовать этот метод для индикации, когда текст - в перекрестии(прицеле), которое я буду рисовать в середине экрана. Листинг 5 показывает как использовать выравнивание и выбор.
Листинг 5. Выравнивание и выбор

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

  try
  {
    // Загрузка Мира из M3G двоичного файла.
    Object3D[] objects = Loader.load(M3G_FILE_NAME);
    _world = (World) objects[0];

    // Изменение свойств камеры на соответствие текущему устройству.
    Camera camera = _world.getActiveCamera();
    float aspect = (float) getWidth() / (float) getHeight();
    camera.setPerspective(60.0f, aspect, 1.0f, 1000.0f);

    // Выравнивание текста к камере.
    Mesh text = (Mesh) _world.find(USER_ID_TEXT);
```

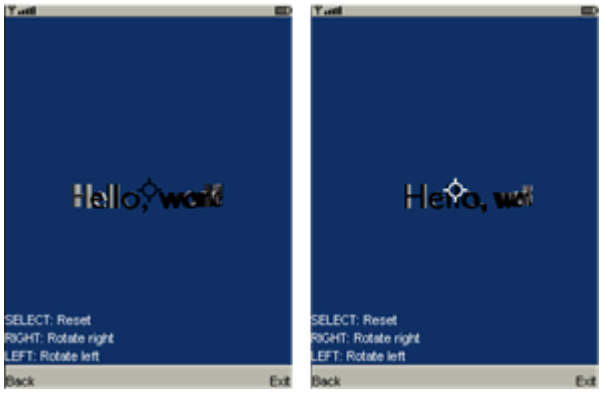
```
      text.setAlignment(camera, Node.Y_AXIS, null, Node.NONE);
      text.align(null);

      // Загрузка изображения перекрестия.
      try
      {
        _crossHairImageOn = Image.createImage(CROSS_HAIR_ON);
        _crossHairImageOff = Image.createImage(CROSS_HAIR_OFF);
      }
      catch (Exception e) {}
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  /**
   * Рендеринг примера на экран.
   *
   * @ параметр graphics – графический объект для вывода.
   */
  protected void paint(Graphics graphics)
  {
    _graphics3d.bindTarget(graphics);
    _graphics3d.render(_world);
    _graphics3d.releaseTarget();

     // Нарисовать перекрестие.
    if ((_crossHairImageOff != null) && (_crossHairImageOn != null))
    {
      graphics.drawImage(_crossHairOn ? _crossHairImageOn : _crossHairImageOff,
            getWidth()/2, getHeight()/2, Graphics.VCENTER | Graphics.HCENTER);
    }

    drawMenu(graphics);
  }

  /**
   * Управление нажатием клавиш.
   *
   * @параметр keyCode код клавиши.
   */
  protected void keyPressed(int keyCode)
  {
    Mesh text = (Mesh) _world.find(USER_ID_TEXT);

    switch (getGameAction(keyCode))
    {
      case LEFT:
        text.postRotate(2.0f, 0.0f, 0.0f, 1.0f);
        break;

      case RIGHT:
        text.postRotate(-2.0f, 0.0f, 0.0f, 1.0f);
        break;

      case FIRE:
        init();
        break;

// no default
    }
```

```
  //  Проверить, пересекается ли луч, брошенный перекрестием с текстом.
  _crossHairOn = isHit();

  repaint();
 }

 /**
  * Проверяют как луч, который начинается  в середине окна просмотра(viewport) и имеет
  * тоже направление что и  активная камера, пересекается ли с текстом.
  *
  * @возвращается true, если есть пересечение.
  */
 private boolean isHit()
 {
  boolean isHit = false;
  RayIntersection rayIntersection = new RayIntersection();
  Mesh mesh = (Mesh) _world.find(USER_ID_TEXT);

  if (_world.pick(-1, 0.5f, 0.5f, _world.getActiveCamera(), rayIntersection))
  {
   if (rayIntersection.getIntersected() == mesh)
   {
    isHit = true;
   }
  }

  return isHit;
 }
```

После загрузки M3G файла, вызов setAlignment () в init () устанавливает информацию выравнивания для Каркаса текста. Первая пара параметров определяет выравнивание оси Z; вторая пара определяет это оси Y. Я ориентирую ось Z текста к оси Y камеры и не изменяю ось Y текста. Выбор осей обусловлен различием в Blender's и M3G's системе координат. Фактическое преобразование на Каркасе текста выполнено с вызовом align().Если Вы хотите непрерывно отследить путь объекта, Вы должны вызывать align() на каждом шаге перемещения объекта. init() также создает два изображения, которые я использую, чтобы начертить перекрестие в paint().
В зависимости от значения булевой переменной _crossHairOn, перекрестие -черное(_crossHairImageOff) или белое(_crossHairImageOn). Вы можете легко сКаркасать запросы к Graphics и Graphics3D. Только удостоверьтесь, что запросы к Graphics3D сделаны между bindTarget () и releaseTarget (), а запросы к Graphics снаружи *Target ().
Состояние перекрестия определено в keyPressed().
Когда нажата соответствующая клавиша, keyPressed() вращает текст или налево или направо. В то же самое время, isHit () проверяет, наведено ли перекрестие на текст. В этом методе, я использую Group.pick (), который входит в два варианта – первый, который берет произвольное координаты луча и цели, второй - определяет луч в координатах окна просмотра(viewport). Его начало координат находится в верхне-левом углу и диапазоне координат от 0 до 1.
Поскольку я интересуюсь выбором Каркаса, основанного на текущей камере, последний вариант наиболее удобен.
Координаты, (0.5, 0.5), обозначают середину экрана, потому что окно просмотра по умолчанию - того же самого размера, что и экран.
Если есть сообщение о пересечении, я изменяю значение _crossHairOn, и показываю соответствующее изображение перекрестия как изображено на Рисунке 6.
AlignmentPickingSample.java - полный исходный пример кода.

Рисунок 6. Текст, выровненный к камере:
a) Перекрестие не пересекается с текстом, b) Перекрестие пересекается

a) b)



Резюме
В двух статьях этой серии, я изложил основы разработки 3D – приложений для мобильных телефонов. Непосредственный режим рендерит 3D- объекты непосредственно на экран. Сохраненный режим, с другой стороны, позволяет Вам строить граф сцены, которым Вы можете манипулировать и рендерить в более позднее время. Для непосредственного режима, важно понять, как вручную создать 3D- сцену. Альтернативно, Вы можете импортировать модель из 3D- инструмента, используя M3G's двойной формат файла. Это позволяет Вам создать сложные 3D- миры и помогает разделению работы между разработчиками и художниками.
Однако, я не упомянул о последовательностях мультипликации(animation sequences), тумане(fog), слоях(layers), 3D-спрайтах, морфинге Каркасов(morphing meshes), или Каркасах, которые представляют скелет(skeletally), анимированный многоугольник. Я оставлю это Вам для исследования.

Загрузи

Описание	Имя	Рамер	Метод загрузки
Пример кода	wi-mobile2source.zip	136KB	HTTP

Ресурсы
Обучение

- The first part of this series вводит в непосредственный режим и обучает как Вы можете работать с огнями, камерами, и материалами.
- 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).
- Получи много идей и экспертных советов по беспроводной технологии на developerWorks Wireless technology zone.

Получение продуктов и технологий

- Загрузи open source инструмент Blender для создания Ваших 3D моделей. Обучающие программы и документация могут быть найдены на BlenderWiki.
- Я нашел два скрипта экспорта для Blender которые пишут M3G файлы: один у MIKlabs и другой у Gerhard Völkl. Я использовал последнего для примеров в этой статье.
- 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.
- Стройте Вашу последующую разработку с IBM trial software, доступной для загрузки с developerWorks.
- Посетите IBM Press Bookstore для получения всеобъемлющего списка технических книг.

Дискуссии

- 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.