Октябрь 2005

Подарок от Redikod:
Учебное пособие по 3D- программированию для мобильных устройств, часть четыре: встроенное столкновение M3G, физика света и перспективы камеры

Мы теперь достигли четвертой части в серии учебных пособий " 3D- программирование для мобильных устройств, используя M3G (JSR 184) ", созданных Mikael Baros, старшим программистом из Redikod.Основываясь на первых трех частях, он теперь берет Вас в мир перспектив камеры и столкновений. После объяснения некоторой теории, он проведет Вас через создание 3D- Pong игры.

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

Часть четыре: встроенное столкновение M3G, физика света и перспективы камеры

Часть четыре путеводителя от Mikael-a.

Введение

Добро пожаловать в четвертую часть обучающей программы по M3G! На сей раз вокруг я преподам Вам некоторые очень важные вещи относительно динамики любой 3D-игры, а именно столкновение и физику. Я также покажу Вам, как работает матрица перспективы камеры и что Вы можете сделать, управляя ей.

Как всегда, проверьте здесь, если когда-либо Вы растерялись:

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

Есть много целей в четвертой части этой обучающей программы и я определю их для Вас. Это - то, чего Вы с надеждой достигнете:

- Узнаете о перспективной матрице и манипуляции камерой
- Узнаете, как использовать встроенный в M3G механизм для быстрого обнаружения столкновения
- Получите основное понимание того, как физика может изменить чувство игры в том, что она становится более реалистической
- Осуществите очень простую pong-подобную 3D-игру

Возможно слишком много? Нет. Вы увидите, что большинство вещей, которые я покажу Вам сегодня, действительно легки, как только Вы станете использовать их.Эта обучающая программа будет сначала немного отяжелена теорией, но облегчится, когда мы доберемся до реальной игры, которую мы собираемся делать.

Так как код предназначен для образовательных целей, он не оптимален, и при этом не закрывает все ошибки, которые могут произойти. Это в более продвинутых темах, к которым обратимся позже.

Что Вы должны знать

Прежде, чем Вы начнете читать это, Вы должны иметь элементарные знания 3D- математики, в частности - векторные и матричные операции.

Вы должны были также конечно прочитать первые три части обучающей программы, познакомиться с API M3G и его линией рендеринга.

Камера и Перспектива

Теперь, Вы могли бы задаваться вопросом, что является предметом разговора о перспективе? Хорошо, перспективная матрица – техническо-математическое свойство в 3D- движке определяет несколько параметров, которые изменяют обзор(viewing) нашей сцены. Но, вместо того, чтобы объяснять Вам чистую математику, я объясню принцип.

Когда мы рассматриваем 3D- объекты в M3G, мы делаем это помещением камеры в определенную позицию пространства, и рассматриваем мир вокруг нас оттуда с определенной областью обзора.

Например; люди не имеют такой же области обзора как птицы. Некоторые птицы имеют область обзора до 360 градусов, в то время как люди обычно имеют приблизительно 150-180 градусов. Теперь, область обзора или FOV (field of view; я буду далее называть так, и Вы помните об этом!) в 3D- игре - в основном описание того, как камера (думайте; игрок) видит объекты в нашем мире. Есть четыре параметра о которых я буду говорить сегодня. Эти параметры - те же самые параметры, которые необходимы в вызове метода Camera.setPerspective в M3G. Это выглядит так:

setPerspective(float fovy, float aspectRatio, float near, float far)

Ось Y области обзора, или коротко fovy
Большинство API обычно нуждается в области обзора, в градусах, чтобы быть способным построить перспективную матрицу. M3G не отличается. Первый параметр метода setPerspective - float fovy. fovy устанавливает область обзора относительно оси Y. Обычно, люди связывают угол с областью обзора относительно оси X, или как широко Вы можете видеть. Однако, M3G интересуется тем, как высоко Вы можете видеть. Нормальные значения для этого из постоянного диапазона от 45-90 градусов, в зависимости от того, что Вы хотите видеть. Если Вы имеете трудности с выбором - высота с fovy 60 градусов.

Так, что fovy делает? Хорошо, это просто говорит камере о том, какие угловые объекты должны быть исключены из рендеринга. Например, если объект - под y-углом 110 градусов от камеры, а наш fovy - только 60 градусов, мы не способны видеть его, пока мы фактически не увеличим угол камеры достаточно (свыше 50 градусов). Это - очень критический компонент в создании визуального реалистического вида 3D-игры. Слишком маленький fovy дает эффект неспособности видеть что - нибудь за исключением мертвой дали, в то время как слишком большое значение позволяет Вам нереалистично намного больше видеть, чем приучен нормальный человеческий глаз. В этой обучающей программе, Вы увидите, как можно управлять fovy, чтобы изменить обзор вашей игры в зависимости от того, что Вы хотите показать.

Аспект отношение
Это - довольно простой параметр, это - фракционное число, которое говорит движку M3G о отношении ширины к высоте текущего экрана. Большинство компьютерных экранов имеет отношение аспект отношения 4:3 (то есть высота - ¾ ширины. 600/800 = 0.75 = ¾), в то время как нормальные мобильные телефоны имеют широкое разнообразие аспектов отношений. Чтобы получить эту переменную, все, что Вы должны сделать, должны разделить текущую ширину экрана на высоту.

Я не буду говорить больше об этом, поскольку все довольно очевидно. Вы должны знать аспект отношения вашего телефонного экрана, чтобы показать геометрию, которая не искажена неестественными способами.

Ближний и дальний планы обрезания
Другой очень легкий параметр. Ближний и дальний планы обрезания просто определяют, какие близкие и далекие объекты не могут рендериться. Так например, установив ближний план обрезания 0.1 и дальнего - на 50 единиц, мы установили, что все объекты, которые ближе чем 0.1 единицы к камере, не будут рендериться. То же самое для объектов, которые расположены дальше от камеры на более 50 единиц. Фактически, 0.1 и 50 - довольно нормальные значения, но конечно Вы можете изменить их на те, в которых Вы нуждаетесь в вашей игре. Это, все сводится к тому, как Вы измеряете вашу игру и используете floating единицы. В этой обучающей программе, мы будем использовать 0.1 для ближнего плана обрезания, и 50 -для дальнего плана обрезания.

Почему, о почему, я должен это знать?
Хорошо, есть много ответов на вышеупомянутый вопрос, но я начну с самого очевидного ответа. Поскольку по умолчанию камера в M3G (является камера, которую Вы не извлекаете от M3G файла), не имеет определенной реалистической перспективной матрицы. Если Вы попробуете отрендерить геометрию на экран, используя только по умолчанию камеру (Camera cam = new Camera()), Вы будете удивлены. Если Вы хотите использовать вашу собственную камеру (которую Вы хотите использовать 90 % времени, особенно при использовании режима непосредственного рендеринга) Вы захотите определить вашу собственную перспективную матрицу. Также, другая хорошая вещь о знании перспективной матрицы - то, что Вы можете достигнуть некоторых довольно “кульных” эффектов, только управляя перспективными параметрами.

Так, как мы создаем чистую, новую, камеру, то назначим ей перспективную матрицу? Подобно этому:

```
float ar = (float)getWidth() / getHeight();
Camera cam = new Camera();  // Это - наша новая камера
cam.setPerspective( 60.0f, // fovy
    ar,  // аспект отношения
    0.1f,  // ближний план обрезания
    50.0f ); // дальний план обрезания
```

Это было не трудно? Фактически, подобная часть кода может быть найдена в исходном коде этой обучающей программы, в методе loadCamera. Вы можете проверить это позже.

Перемещение объектов в трехмерном пространстве

Чтобы перемещать объекты, Вы должны сделать немногое, и Вы вероятно имеете хорошую идею как это сделать. Я пройду некоторые очень легкие и простые методы физики для перемещения объектов в 3D-пространстве.
Обычно, когда объект перемещается в 3D-пространстве, Вы нуждаетесь в трех различных скоростях: x, y и z скоростях. Однако, большинство игр может комбинировать x и z скорости в одну, называемую скоростью "вперед". (Например, если ваша скорость"вперед" - 1.0, то все, что Вы должны сделать – трансформацияывать ее с синусом и косинусом и добавить к x и z) В этой обучающей программе, мы не будем делать такой дискриминации осей, вместо этого мы будем использовать все три отличительных скорости, чтобы описать двигающееся тело в 3D-пространстве.

Самый легкий способ сделать это, является использование трех векторов.
1. Вектор координаты
Первый вектор - вектор координаты объекта. Это - критический вектор, который содержит текущее положение объекта в 3D-пространстве, содержит след его x, y и z координат. Как Вы будете еще знать, где прорендеренный объект?

2. Скоростной вектор
Второй вектор - скоростной вектор. Он определяет, на сколько единиц массив координат объекта будет передвигаться в следующее время, когда движение вычислено. Так что также требуются компоненты x, y и z. В основном, каждый раз Вы только добавляете x-скорость к координате x, y-скорость к координате y и z-скорость к координате z. Так, если бы Вы имеете космический корабль, перемещающийся на одну единицу прямо по оси Y в каждом фрейме(прим. шаге игры), Вы имели бы скоростной вектор (0, 1, 0).

3. Вектор ускорения
Последний вектор - вектор ускорения. Хотя мы не будем использовать его в этом примере, это все же важно. Вектор ускорения работает подобно скоростному вектору, но он оперирует скоростным вектором, а не координатами. Это означает, что вектор ускорения напрямую не манипулирует координатами, а вместо этого он увеличивает скорость в каждом фрейме(прим. шаге игры), давая игроку чувство ускорения. Обычно, объект не может немедленно разогнаться от состояния покоя до максимальной скорости. Требуется время и для модели реалистичной 3D-игры необходимо ускорение. Скажите, что космический корабль из вышеупомянутого примера хотел начать двигаться со скоростью одна единица в шаг игры и достичь скорости пяти единиц в шаг игры. Это могло быть сделано при наличии скоростного вектора (0, 1, 0) и вектора ускорения (0, 0.1, 0). Это означает, что после 40 шагов, наш космический корабль будет иметь скорость 5 единиц.

Пример движения
Теперь, давайте поместим наш пример космического корабля в код! Это - то, что выглядит так:

```
float[] spaceShipCoords = {0, 0, 0};
float[] spaceShipVel = {0, 1, 0};
float[] spaceShipAcc = {0, 0.1, 0};
while(gameLoopIsRunning)
{
 for(int i = 0; i < 3; i++)
 {
  spaceShipCoords[i] = (spaceShipVel[i] += spaceShipAcc[i]);
 }
}
```

Смотрите, как легко это было? Каждая шаг игры мы ускоряем и перемещаем наш космический корабль.

Столкновение
Объекты в трехмерном пространстве обычно непрозрачны и сталкиваются друг с другом. Комета может врезаться в планету и (довольно бедный), автомобиль может врезаться в здание. Однако, в наших 3D-играх это не столь же очевидно, как это происходит в реальном мире. Это потому что пока мы только рендерили объекты и эти объекты - только математические представления 3D- моделей, которые трансформацияаны в пикселы экрана. Мы действительно не рассматривали, как различные объекты касаются друг друга. Так, если бы мы перемещали наш космический корабль к некоторой планете, которая могла бы существовать, космический корабль продолжит проходить сквозь планету, и появится с другой стороны. Все потому, что мы не имеем никакого столкновения. Теперь мы изменим это.

Столкновение пересечением лучом
Пересечение лучом - очень интуитивная и легкая форма обнаружения столкновения. По крайней мере для нормальных и простых случаев столкновения. Это работает подобно этому; объект, от которого Вы желаете проверить столкновение, запускает луч от его центра (или откуда-нибудь из его тела) в данном направлении (обычно скоростной вектор объекта, но скоро Вы увидите, что это не всегда так). Этот луч путешествует через ваш 3D- мир, пока он фактически не поражает что-нибудь(подобно лазеру). Когда он поражает что-нибудь, он сообщает о столкновении и говорит Вам, как далеко можно было путешествовать, пока не поразили объект. В зависимости от этого расстояния, Вы определите, был ли факт столкновением или нет. Это - главным образом, потому что, если Вы знаете, что впереди Вас есть здание на расстоянии одна тысяча футов, это не подразумевает, что Вы столкнетесь с ним. (Хорошо, Вы бы в конечном счете могли, но Вы уверен, все же не сделали этого)

Подход M3G
M3G дает нам исключительно простой способ вычислить столкновение. Каждая Группа в нашем мире (помните, Группа - только набор Узлов, которые могут быть чем - нибудь существенным в нашем мире), имеют метод pick(выбор). Этот метод - тот, который определяет для нас все столкновения. Это работает подобно тому, скажете Вы, как работает следующий мегахит FPS для мобильных телефонов, и Вы хотите начать, проверяя столкновение вашего главного персонажа со стенами в комнате, или столкновение его с монстрами в комнате. Так, что Вы делаете - Вы, создаете две Группы. Одна Группа для всех стен, и один для всех монстров. Тогда Вы только просто бросаете луч к каждой группе, от персонажа и к его направлению, и видите, сталкиваетесь ли Вы с чем - нибудь.

M3G управляет частью игры с столкновениями двумя компонентами. **Pick** и RayIntersection класс.

Класс RayIntersection
Это - довольно простой класс, который только содержит критическую информацию к броску луча в нашем 3D-мире. Метод, который мы пройдем сегодня (и вероятно единственный, для легкого столкновения Вы будете в нем нуждаться) - getDistance метод. getDistance метод имеет вид:

public float getDistance()

Столь же прост, как и выглядит.Он возвращает расстояние до объекта, с которым этот специфический луч столкнулся, масштабированное с вектором направления, которым Вы снабдили метод **pick** (больше об этом позже). Так, только делая простую проверку, Вы можете увидеть на сколько Вы близки к объекту в Группе, чтобы определить, имеете ли Вы столкновение. Только посмотрите на эту часть кода:

```
// Дистанция до столкновения(по умолчанию) в нашей игре
float collisionDistance = 0.1f;
// Этот метод возвращает луч, который имеет или нет столкновение со стеной
RayIntersection ray = checkCollisionWithWalls();
// Проверить расстояние до стены (если мы не столкнулись, расстояние будет большое)
if(ray.getDistance() < collisionDistance)
{
 // Получить стену, с которой мы столкнулись
 Node wall = ray.getIntersected();
 // Сделать кое-что...
}
```

Смотрите, вообще не трудно! Это - все волшебство в RayIntersection классе. Теперь давайте посмотрим, как мы в него можем фактически заполнить информацию столкновения.

Метод Group.pick
Для заполнения необходимой информацией RayIntersection класса, Вы должны будете использовать метод **pick** (если Вы не хотите отменить стандартное определение столкновения и хотите заполнять информацию самостоятельно. Возможно Вы хотите осуществить ваш собственный алгоритм пересечения луча?). Pick метод имеет два варианта, и я покажу Вам оба перед процессом:
public boolean pick(int scope, float ox, float oy, float oz, float dx, float dy, float dz, RayIntersection ri)
public boolean pick(int scope, float x, float y, Camera camera, RayIntersection ri)

Сначала давайте разговор о подобиях. Хорошо, оба хотят область действия(scope). Область действия просто определяет, какие объекты луч будет считать как сталкивающимися. Каждый Узел в API M3G имеет область действия, так если Вы передаете 1 методу **pick** как параметр область действия, **все узлы** с областью действия эквивалентной 1. Поставляя-1 Вы проверяете (прим. столкновение) против всех узлов в той группе. Иногда поставка-1 плоха, поскольку Вы не хотите вашим алгоритмом вычислять столкновение с чем либо, зная, что ваш объект далеко. Всегда должна быть сделана попытка минимизировать работу алгоритма.

Вторая и последняя вещь, что оба в общем имеют - объект RayIntersection. Это снабженный класс RayIntersection, который они заполняют в случае столкновения.

Теперь, давайте пройдем первый метод. Этот метод - наиболее используемый, поскольку он позволяет Вам определять отправную точку вашего луча. ox, oy и oz - три компонента вектора координат(three components of the origin vector). То есть те три определяют точку в пространстве, с которой луч начнется. Обычно, Вы установите его на центр столкновения для вашего объекта. Следующие три компонента, dx, dy и dz составляют вектор направления. Этот вектор решает, в каком направлении луч будет путешествовать. Вы будете всегда желать снабдить им как вектором единицы (unit vector; вектор с длиной 1), поскольку метод pick масштабирует расстояние, о котором сообщает getDistance в зависимости от длины вашего вектора направления. Иногда, это- нужно, но в большинстве времени – нет необходимости, так только удостоверьтесь, что ваш вектор направления имеет длину 1.

Итак, как первый метод выглядит в коде?

```
// Получить нашу группу со стенами
Group walls = getWallGroup();
// Создать объект RayIntersection для метода заполнения
RayIntersection ri = new RayIntersection();
// Получить координаты нашего героя
float[] coords = getPlayerCoords();
// Получить направление нашего героя(куда указывает его нос?)
float[] dir = getPlayerDirection();
// Пробовать столкнуться (метод возвращает, true, если столкновение имело местопространствео)
Try colliding (method returns true if collision has taken place)
if(walls.pick(-1, coords[0], coords[1], coords[2], dir[0], dir[1], dir[2], ri))
{
 // Мы пересекли кое-что, проверяем для расстояния
 if(ri.getDistance() < acceptableDistance)
 {
  // Игрок столкнулся со стеной. Заставьте его взорваться...
  // Или кое-что.
   }
}
```

Смотрите, не столь страшно, как Вы ожидали! Это фактически довольно легко, только определять от куда Вы стреляете лучом и его направление. Все остальное для Вас делает M3G. Мы - не испортили ли?
Теперь Вы могли бы задаваться вопросом, для чего другой метод. Он тоже используется, но он работает немного по-другому. Я не буду пробегать через него здесь, так как он не используется нами в этой обучающей программе. Однако я очень рекомендую почитать об этом в документации по API M3G.

Спуск и грязь
Теперь, когда я преподал Вам несколько уловок, не поговорить ли о размещении их в коде для использования? Я имею в виду игру, которая удовлетворит нас только прекрасным в сложности. Давайте делать 3D- pong. Мы не будем иметь двух игроков, а скорее Вы будете подпрыгивать за шаром от вашей ракетки и глубоко от экрана. Область игры будет состоять из четырех стен от которых ваш шар отпрыгнет(АГА! Столкновение!). Что мы должны сделать теперь – разбить(прим. сделать декомпозицию) простую идею на несколько ключевых компонентов.

Область игры и ракетка
Наша область игры, как я уже сказал Вам, будет состоять из ракетки и четырех стен. Мы будем использовать точно тот же самый класс для генерации планов, как и делали в части три из обучающей программы, так теперь самое время освежить вашу память.

Ракетка и стены будут простыми текстурированными планами. Чтобы сделать их различимыми, ракетка и стены будут иметь различные текстуры. Все это будет создано методом MeshFactory.createPlane.

Вот - отрывок кода, который создает ракетку, используя класс MeshFactory, который мы построили в части три из этой обучающей программы:

```
// Создать план, используя нашу остроту- класс MeshFactory
paddle = MeshFactory.createPlane("/res/paddle.png", PolygonMode.CULL_BACK);
```

Ничего трудного там, это сейчас уже должно быть очень знакомо. Ракетка будет конечно нуждаться трансформации, с которой мы будем способны перемещать ракетку. Мы назовем это trPaddle, и вот как мы инициализируем ее.

```
// Установить ракетку в ее начальное положение
trPaddle = new Transform();
trPaddle.postTranslate(paddleCoords[0], paddleCoords[1], paddleCoords[2]);
paddle.setTransform(trPaddle);
```

Переменная paddleCoords - float вектор, который определяет, где ракетка находится сейчас. Его содержание помещается в trPaddle каждый раз, когда ракетка перемещается.

Теперь, чтобы убедиться, что наша ракетка будет сталкивающейся, мы должны будем установить несколько переменных. Сначала, мы нуждаемся в родовом способе узнать нашу ракетку, когда мы используем метод RayIntersection.getIntersected. Он возвращает класс Узла, помните? Так, это - то время, чтобы использовать переменную userID, которую имеет каждый Object3D. Я создал несколько переменных, которые определяют стены и ракетку:

```
// Константы стен
    public static final int TOP_WALL = 0;
    public static final int LEFT_WALL = 1;
    public static final int RIGHT_WALL = 2;
    public static final int BOTTOM_WALL = 3;
    public static final int PADDLE_WALL = 4;
    public static final int PLAYING_FIELD = 5;
```

Имея этие переменные, мы можем просто вызвать setUserID у ракетки и снабдить его константой PADDLE_WALL. Последняя вещь, которую мы должны сделать, должна удостовериться, что у нашей ракетка установлен выбор столкновений. Помните, что метод столкновения называют pick(выбор)? Хорошо, в этом методе все объекты проверяются на две вещи. Сначала на область действия возможности (который мы установим -1, так что мы не должны волноваться об этом) и вторая вещь флаг разрешения выбора столкновений. Если у объекта флаг разрешения выбора столкновений не установлен, то он просто не будет включен в вычисление столкновения. Флаг - легко переключается, вызывая метод setPickingEnabled у объекта ракетка. Установка userID и флага разрешения выбора столкновений выглядит так:

```
// Сделать это – сталкивающейся(collidable)
paddle.setPickingEnable(true);
paddle.setUserID(PADDLE_WALL);
```

Там, все сделано. Наша ракетка теперь создана и она сталкивающаяся(collidable). Теперь мы только добавим ее к группе (которую я называю playingField), и позже мы можем проверить столкновение против всех объектов в playingField.

```
// Добавить к области игры
playingField = new Group();
playingField.setUserID(PLAYING_FIELD);
playingField.addChild(paddle);
```

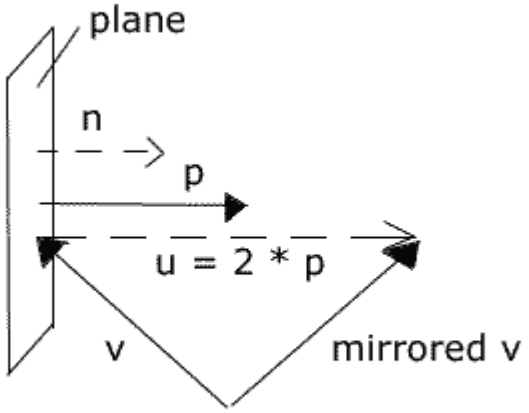
Есть также еще некоторый код в конце метода createPaddle, но я пробегусь через эту часть немного позже, когда мы поговорим о подпрыгивании шара.

Наши четыре стены созданы подобным способом, так что я не буду повторять код. Только проверьте метод createWalls в исходном коде, если Вы хотите видеть, как это сделано.

Шар
Некоторые говорят, что шар - самая критическая часть pong, а некоторые говорят -ракетка. Я сказал бы, что критически - оба. Теперь, каким наш шар должен быть? Прежде всего, это должен быть Мешь, который может быть рендерен на экран(onscreen; это помогает, если этот Мешь, в свою очередь, является фактически моделью шара). Вовторых для всех, он должен содержать достаточно физики, чтобы наш шар мог двигаться в 3D- пространстве и ускоряться. Наконец, должна иметься способность вычислять его собственное продвижение через 3D- мир.

Подпрыгивание
В нашем очень простом примере, мы не нуждаемся ни в какой трудной теории, чтобы отпрыгивать шар от стены. Однако, так или иначе я собираюсь дать Вам некоторую теорию, вдруг Вы решите создать в игре вращающиеся стены или попробуете сделать кое-что фактически реальное и получить отпрыгивающие вещи от нерегулярный поверхности. Теперь, как Вы подпрыгиваете (зеркало) -вектор? В чем мы нуждаемся, это просто зеркальный вектор к направлению шара, когда он сталкивается со стеной. Мы также имеем зеркало, это в правильном плане (зеркалирование в xz-плане не очень хорошо, если мы столкнулись с правой стеной). Это весь управляемо простой векторной математикой. Я не буду проходить трудную математическую теорию, так как я надеюсь, что Вы уже имеете базу в 3D- математике, так как Вы пробуете делать 3D- игру. В контексте этого я только скажу Вам о подходе.

Прежде всего, я создал класс VectorOps, который должен помочь Вам во время создания и эксперимента с векторами и подпрыгиванием. Он содержит несколько из основных операций, которые Вы хотели бы сделать с вектором, таких как проектирование его на план, точечные и пересекающиеся изделия, нормализацию, вычисление длины, и т.д. Я также добавил метод зеркала. Это - метод, что зеркалирует вектор относительно данного плана (только в этом случае у плана должен быть определен его вектор нормали).
Как тогда зеркало работает? Хорошо это фактически довольно просто. Сначала, Вы проектируете ваш исходный вектор (далее- v) на нормаль из поверхности (далее - n). Тогда, делая некоторую простую векторную математику Вы понимаете, что, используя проецируемый вектор (далее- p) Вы можете построить новый вектор с удвоенной длиной (далее- u). Теперь этот новый вектор u в пространствее с первоначальным вектором v может быть объединен, чтобы формировать зеркальный вектор. Как? Хорошо только посмотрите на эту картину.



Вы можете легко видеть, что зеркальный вектор v, что нам требуется, может быть вычислен от этой картины. Если Вы хотите видеть как вычисляется, проверьте метод зеркалирования VectorOps.

Это означает, что в игре мы должны хранить векторы нормалей каждой стены (и ракетки). Это вообще не трудно, так как мы имеем только четыре стены и ракетку, но для более сложных проектов это могло быть проблемой. Однако, мы будем строить вектор нормалей плана, используя простое взаимное изделие двух векторов, которые составляют план. (Помните 3D-математику? Каждый план может быть описан как натягиваемый двумя векторами. Беря взаимное изделие этих двух векторов Вы получаете вектор нормали плана)

В нашем приложении мы будем использовать векторы нормалей для только зеркальных вычислений, и мы не будем волноваться об ориентации и других вещах, которые Вы хотели бы учесть при вычислении нормалей. Это также означает, что левые и правые стены будут иметь те же самые нормали. Так будет и у верхней и нижней стен. Нормали рассчитаны и помещены в floats массив, так, чтобы наш шар используя их, способен был бы вычислить сильный удар. Массив будем называть wallVec и вот - отрывок кода, который показывает, как рассчитан вектор нормали левой стены и помещен в wallVec.

```
float[] v = VectorOps.vector(0.0f, 1.0f, 0.0f);
float[] u = VectorOps.vector(0.0f, 0.0f, 1.0f);
float[] normVec = VectorOps.calcNormal(v, u);
wallVec[LEFT_WALL][0] = v;
wallVec[LEFT_WALL][1] = u;
wallVec[LEFT_WALL][2] = normVec;
```

Здесь действительно ничего сверхъестественного. Поскольку Вы все знаете, что левая стена - фактически yz план, перемещение на несколько единиц вниз по оси X, так именно поэтому он составлен из чистых y и z векторов. После того, как векторы созданы, рассчитан вектор нормали вызовом метода calcNormal. Он просто делает crossProduct двух снабженных векторов и затем нормализует результат, делающий вектор вектором единиц.

Вышеупомянутое сделано для каждой стены в области игры, если бы не стена камеры (нос игрока, если Вам нравится). То столкновение сделано было бы по-другому (и намного более простым способом).

Бдительный читатель и программист непосредственно поймут, что вышеупомянутое - для слишком простого приложения. Так как наши стены - xz, xy и zy планы, в которых проверку мы могли сделать в if-операторах, с каким планом мы столкнулись - зеркальная координата той стены представляется. Однако, я хотел дать Вам более изящное и общее решение. Вышеупомянутое решение работает для любых стен, независимо от того как они перемещены. Поскольку я уже сказал однажды, это - не обучающая программа оптимизации, так что если Вы хотите, чтобы приложение работало немного быстрее, удалите вычисление зеркалирования и сделайте вложенным если хотите.

Реализация

Так как я показал Вам руководящие принципы, теперь я покажу Вам реализацию метода рендеринга шара, который не только рендерит, но также и перемещает шар и проверяет столкновения. Давайте начнем с рассмотрения первых нескольких строк:

```
// Очистка трансформации
  trBall.setIdentity();

  // Проверить, двигаемся ли мы
  if(moveVec != null)
  {
   // Сначала вращайте шар немного
    trBall.postRotate(rotated += 1.0f, 1.0f, 1.0f, 1.0f);
   // Нормализовать вектор движения
       ri = new RayIntersection();
       float[] nMove = VectorOps.normalize(moveVec);
```

Как Вы можете видеть, что первые строки являются довольно прямыми. Что мы делаем, во-первых мы сначала очищаем все трансформации в Transform классе шара, названном trBall. Затем, мы просто делаем проверку на движение (игра начинается с покоющимся шаром). Шар также имеет переменную rotated, которая содержит след вращения шара, давать впечатление от вращения и

полета шара. Эта переменная используется в методе postRotate, который Вы должны теперь знать наизусть. После этого стартует интересная часть. Мы создаем объект RayIntersection, который мы будем использовать для столкновения и нормализуем вектор скорости шара (называемый moveVec). Помните, почему мы нормализуем? Потому что метод pick всегда масштабирует расстояние-до-пересечения с длиной вашего вектора направления. Если длина вектора руководства равна 1, расстояние-до-пересечения не масштабировано. Это - так, как мы этого хотим. Давайте продолжим с методом, теперь прибыло столкновение:

```
// Смотри, если есть любое столкновение
 if(walls.pick(-1, coords[0], coords[1], coords[2], nMove[0], nMove[1], nMove[2], ri) && ri.getDistance()
<= 0.5f)
{
   // Мы столкнулись, получаем поверхность, которую мы пересекли
   Node n = ri.getIntersected();

   // Исправить наш вектор движения зеркалированием
   moveVec = VectorOps.mirror(moveVec, wallVectors[n.getUserID()][2]);
```

Так что есть легендарный запрос к Group.pick. Как Вы видите,он выполнен в Группе названной стенами, которая фактически является playingField группой, прошел к методу рендеринга шара. Так что группа стен содержит и все стены и ракетку. Мы проверяем, возвращает ли метод pick истину и если расстояние меньше 0.5, что является довольно разумным расстоянием в этом приложении.

Если мы имеем столкновение, мы сначала получаем узел, который мы пересекли. Это - одна из стен из нашей Группе стен. Тогда мы получаем ID стены. Помните? Мы использовали их как индексы в массиве wallVec, чтобы хранить векторы нормалей стен. Здесь мы получаем их, вызывая метод getUserID у Узла. Когда мы имеем вектор нормали, мы вызываем метод mirror для зеркалирования текущего вектора направления шара вокруг пересеченной стены. Это - все, что есть по этому вопросу.

Вышеупомянутое действительно - не только забавы. Так как все стены рассматривают одинаково, шар только категорически подпрыгнет прочь от ракетки и даст нам некоторый действительно скучный и статический ход игры. Именно поэтому мы будем делать столкновения с ракеткой немного различными. Проверьте эту часть кода:

```
// Позволить пользователю иметь управление  движением для перемещения ракетки
if(n.getUserID() == M3GCanvas.PADDLE_WALL)
{

     // Добавить дополнительную скорость к
    //максимальному количеству в зависимости от положения шар/ракетка
   float distX = (paddleCoords[0] - coords[0]) / 10.0f;
   float distY = (paddleCoords[1] - coords[1]) / 10.0f;
   moveVec[0] = Math.max(-0.3f, Math.min(moveVec[0] - distX, 0.3f));
   moveVec[1] = Math.max(-0.3f, Math.min(moveVec[1] - distY, 0.3f));

   // После 30 сильных ударов это должно быть невозможно быстро (ХА! Вызов!)
   moveVec[2] = moveVec[2] + 0.01f;

   // Увеличение числа сильных ударов
   bounces++;
}
```

Хорошо, так, что мы здесь делаем? Хорошо сначала мы проверяем, отпрыгиваем ли мы фактически от ракетки, сравнивая пользовательский ID Узла и константу PADDLE_WALL(ракетка). Тогда мы используем некоторую простую математику, чтобы вычислить расстояние от центра шара до центра ракетки. Мы используем этот результат, чтобы исказить результирующий вектор направления, чтобы создать более динамичный ход игры.

Другая забавная вещь должна заставить двигаться шар быстрее с каждым сильным ударом. Как Вы знаете, скорость в этой игре может быть столь же проста как скорость по оси Z, так для каждого сильного удара ракетки, мы ускоряем скорость по оси Z.

Последняя вещь, переменная приращения сильного удара, только используется как "выигрыш". Лучший игрок имеет большинство сильных ударов. Довольно простое понятие, но это работает.

Мы увидели теперь в методе Ball.render почти все, если бы не отскок от стены камеры (или от носа игрока). Это сделано в следующим отрывке кода. Я не буду комментировать это, так как Вы только глядя на это, должны быть способны объяснить это.

```
// Проверить для того, чтобы подпрыгнуть против экрана
if(coords[2] >= -0.7f)
{
   // Сальто от экрана (то же самое для ракетки)
   moveVec = VectorOps.mirror(moveVec, wallVectors[M3GCanvas.PADDLE_WALL][2]);
}
```

Там! Это - все, что есть по этому вопросу. Не очень реально, чтобы говорить об этом, не так ли?

Перспектива и обзор трубы

Я обещал Вам, что в этой обучающей программе мы будем использовать перспективные вычисления немного по-другому и делать внешний вид игры немного специальным. Это - то, что мы сделаем, только настроя единственную переменную FOV. Мы изменим fovy на очень большое число, таким образом создавая странный эффект стрижки, который выглядит весьма хорошим в этом контексте. Я здесь использовал 130 градусов (максимум значения, позволенный M3G - 180), так что это дает чувство, что Вы фактически смотрите вниз очень длинной и расширенной трубы. Вы можете играть с этим значением в исходном коде, чтобы видеть, какую стрижку Вы будете способны получить.

Цикл игры

Только одна последняя вещь, которую теперь сделаем, это - реализация цикла игры. Вы уже должны быть способны сделать это во сне вашими руками, привязанными к вашей спине, но я пройду семантику еще раз. Давайте сначала посмотрим на целый цикл игры:

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

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

       // Очистка фона
       g3d.clear(back);

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

       // Рендерить область игры и шар
       g3d.render(playingField, identity);
       ball.render(g3d, playingField, wallVec, paddleCoords);

       // Проверка клавиш для движения ракетки
       if(key[UP])
       {
          paddleCoords[1] += 0.2f;
          if(paddleCoords[1] > 3.0f)
             paddleCoords[1] = 3.0f;
       }
       if(key[DOWN])
       {
          paddleCoords[1] -= 0.2f;
          if(paddleCoords[1] < -3.0f)
             paddleCoords[1] = -3.0f;
       }
       if(key[LEFT])
```

```
        {
            paddleCoords[0] -= 0.2f;
            if(paddleCoords[0] < -3.0f)
                paddleCoords[0] = -3.0f;
        }
        if(key[RIGHT])
        {
            paddleCoords[0] += 0.2f;
            if(paddleCoords[0] > 3.0f)
                paddleCoords[0] = 3.0f;
        }

        // Установка координат ракетки
        trPaddle.setIdentity();
        trPaddle.postTranslate(paddleCoords[0], paddleCoords[1], paddleCoords[2]);
        paddle.setTransform(trPaddle);

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

      // Сделать некоторый старомодный 2D- рисунок
      if(!ball.isMoving())
      {
          g.setColor(0);
          g.drawString("Press fire to start!", 2, 2, Graphics.TOP | Graphics.LEFT);
      }
      else
      {
          int red = Math.min(255, ball.getBounces() * 12);
          g.setColor(red, 0, 0);
          g.drawString("Score: " + ball.getBounces(), 2, 2, Graphics.TOP | Graphics.LEFT);
      }
  }
```

Так сначала – стандартные вещи для непосредственного режима рендеринга. (Если Вы не можете вспомнить, проверьте часть три из обучающей программы).

- Получаем инстанцию Graphics3D
- Связываем с целевым Graphics
- Очищаем фон
- Установка и модернизация камеры

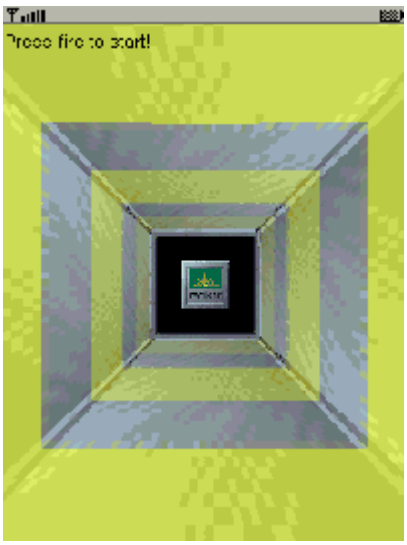
Что мы делаем затем, также довольно просто. Мы только рендерим группу playingField, которая содержит все стенные Меши и их трансформацию. Этим путем мы сжимаем рендеринг пяти объектов в единственную группу. Графы сцены делают очень чистый код.

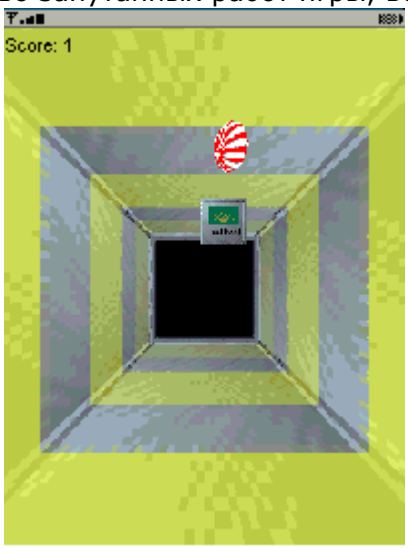
После рендеринга области игры мы вызываем Ball.render метод, и передаем ему необходимые значения, и он делает остальное. Мы уже проходили этот метод, так что я здесь о нем не буду говорить.

Затем мы проверяем контроль. Здесь также ничего нового. Мы перемещаем ракетку джойстиком и повторно устанавливаем первоначальное положение шара клавишей FIRE. После того, как мы сделали всю 3D- графику, мы делаем некоторый старомодный 2D- рисунок, так как мы хотим, чтобы

пользователь прямо сейчас знал его счет и выход шара из границ (и он должен повторно установить шар).

Это - целая энчилада, Так вероятно сказали бы люди из промышленности быстрого питания. Теперь, когда Вы увидели большинство запутанных работ игры, вот - несколько выстрелов этого в действии.





Как Вы можете видеть выбор fovy - 130 дает нам довольно интересный эффект трубы/туннеля. Также, не забудьте, эта игра - фактически симпатичная забава для того и должна быть легкой. С маленьким развитием Вы могли делать эту игру - действительно захватывающей игрой. Как упражнение, пробуйте сделать стены области игры вращающимися! Помните проблемы; Вы должны будете вращать, все векторы нормалей, сохраненные в wallVec, когда Вы вращаете стены, чтобы зеркалирование работало должным образом.

Имеете забаву, и я надеюсь, что Вы узнали кое-что.

TutorialMidlet

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

 // Холст
 private M3GCanvas canvas = null;

 // Ссылка в MIDlet-е на себя самого
 private static MIDlet self = null;

 / ** Вызывается, когда приложение стартует, и когда возобновлено.
  * Мы игнорируем резюме(resume) здесь и помещаем данные для нашего
  *(прим. лучше бы в классе холста)
  * в startApp методе. Это - вообще то очень плохая практика.
  */
 protected void startApp() throws MIDletStateChangeException
 {
    // Выделение
  display = Display.getDisplay(this);
  canvas = new M3GCanvas(30);

  // Добавить к холсту команду выхода
  // Эта команда не будет видна так  как мы
  // работаем в полноэкранном режиме,
```

```
 // но всегда хорошо иметь команду выхода
 canvas.addCommand(new Command("Quit", Command.EXIT, 1));

 // Установка листенера мидлета
 canvas.setCommandListener(this);

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

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

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

 }
 /** Вызывается когда приложение должно быть закрыто */
 protected void destroyApp(boolean unconditional) throws MIDletStateChangeException
 {
    // Метод закрывает MIDlet
  notifyDestroyed();
 }

  /** Слушает команды и обрабатывает */
    public void commandAction(Command c, Displayable d) {
        // Если мы получаем команду  EXIT(ВЫХОД), мы уничтожаем приложение
        if(c.getCommandType() == Command.EXIT)
            notifyDestroyed();
    }

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

M3GCanvas

```
import java.io.IOException;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.game.GameCanvas;
import javax.microedition.m3g.Background;
import javax.microedition.m3g.Camera;
import javax.microedition.m3g.Graphics3D;
import javax.microedition.m3g.Group;
import javax.microedition.m3g.Light;
import javax.microedition.m3g.Loader;
import javax.microedition.m3g.Mesh;
import javax.microedition.m3g.Object3D;
import javax.microedition.m3g.PolygonMode;
import javax.microedition.m3g.Transform;
import javax.microedition.m3g.World;
public class M3GCanvas  extends GameCanvas implements Runnable {

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

    // Если игра закончилась
```

```
    public static boolean gameOver = false;

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

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

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

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

    // Глобальный объект Graphics3D
    Graphics3D g3d = null;

    // Фон
    Background back = null;

    // Глобальный объект камеры
    Camera cam = null;

   // Игровые поля
    Mesh paddle;
    Group playingField;
    Ball ball;

    // Трансформации
    Transform trLeftWall, trRightWall, trTopWall, trBottomWall;
    Transform trPaddle;
    Transform trCam = new Transform();

    // Координаты ракетки
    float[] paddleCoords = {0.0f, 0.0f, -5.0f};

    // Константы стен
    public static final int TOP_WALL = 0;
    public static final int LEFT_WALL = 1;
    public static final int RIGHT_WALL = 2;
    public static final int BOTTOM_WALL = 3;
    public static final int PADDLE_WALL = 4;
    public static final int PLAYING_FIELD = 5;

    // Векторы для наших стен
    // Объяснение: Каждая стена содержит два вектора,
    // которые определяют план (См. линейную алгебру) и
    // вектор нормали стены.
      float[][][] wallVec = new float[PLAYING_FIELD][3][3];

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

```
      // Мы желаем полноэкранный холст
      setFullScreenMode(true);

      //Создаем игроые поля
      createField();

      // Загружаем камеру
      loadCamera();

      // Загружаем фон
      loadBackground();

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

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

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


/** Создает простой окружающий(ambient) свет */
private Light createAmbientLight()
{
Light l = new Light();
l.setMode(Light.AMBIENT);
l.setIntensity(1.0f);
return l;
}

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

   }


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

            // Установить перспективу нашей камеры (выберите довольно
            // широкий FoV для изящного эффекта трубы)
      cam.setPerspective(130.0f, (float)getWidth() / (float)getHeight(), 0.1f, 50.0f);
   }

  /** Загрузка камеры */
    private void loadCamera()
   {
      // Создание новой камеры
```

```
   cam = new Camera();
}

/** Загрузка фона */
private void loadBackground()
{
   // Создание нового фона и установка его цвет черным
   back = new Background();
   back.setColor(0);
}

 / ** Создание нашей области игры. Это будут инстанции шара и трех
 * стен (четвертая стена - "экран"
 */
private void createField()
{
   try
   {
      loadBall();

      createPaddle();

      createWalls();
   }
   catch(IOException e)
   {
      System.out.println("Loading error: " + e);
   }
}
/**
 *
 */
private void createWalls() {
        // Создать все планы  нашим острым классом MeshFactory
        // (мы нуждаемся в нескольких для столкновения)
   Mesh wall1 = MeshFactory.createPlane("/res/wall.png", PolygonMode.CULL_BACK);
   Mesh wall2 = MeshFactory.createPlane("/res/wall.png", PolygonMode.CULL_BACK);
   Mesh wall3 = MeshFactory.createPlane("/res/wall.png", PolygonMode.CULL_BACK);
   Mesh wall4 = MeshFactory.createPlane("/res/wall.png", PolygonMode.CULL_BACK);

   // Здесь Мы хотим хорошую перспективную коррекцию
   MeshOperator.setPerspectiveCorrection(wall1, true);
   MeshOperator.setPerspectiveCorrection(wall2, true);
   MeshOperator.setPerspectiveCorrection(wall3, true);
   MeshOperator.setPerspectiveCorrection(wall4, true);

   // Установить левую стену в ее истинном положении
   trLeftWall = new Transform();
   trLeftWall.postTranslate(-4.0f, 0.0f, -5.0f);
   trLeftWall.postRotate(90, 0.0f, 1.0f, 0.0f);
   trLeftWall.postScale(5.0f, 5.0f, 5.0f);
   wall1.setTransform(trLeftWall);

   // Сделать ее векторы
   float[] v = VectorOps.vector(0.0f, 1.0f, 0.0f);
   float[] u = VectorOps.vector(0.0f, 0.0f, 1.0f);
   float[] normVec = VectorOps.calcNormal(v, u);
   wallVec[LEFT_WALL][0] = v;
   wallVec[LEFT_WALL][1] = u;
   wallVec[LEFT_WALL][2] = normVec;

   // Установить правую стену в ее истинном положении
   trRightWall = new Transform();
   trRightWall.postTranslate(4.0f, 0.0f, -5.0f);
```

```
    trRightWall.postRotate(-90, 0.0f, 1.0f, 0.0f);
    trRightWall.postRotate(180, 0.0f, 0.0f, 1.0f);
    trRightWall.postScale(5.0f, 5.0f, 5.0f);
    wall2.setTransform(trRightWall);

    // Те же векторы что и у левой стены
    wallVec[RIGHT_WALL][0] = v;
    wallVec[RIGHT_WALL][1] = u;
    wallVec[RIGHT_WALL][2] = normVec;

   // Установить верхнюю стену в ее истинном положении
    trTopWall = new Transform();
    trTopWall.postTranslate(0.0f, 4.0f, -5.0f);
    trTopWall.postRotate(90, 1.0f, 0.0f, 0.0f);
    trTopWall.postRotate(-90, 0.0f, 0.0f, 1.0f);
    trTopWall.postScale(5.0f, 5.0f, 5.0f);
    wall3.setTransform(trTopWall);

    // Сделать ее векторы
    v = VectorOps.vector(1.0f, 0.0f, 0.0f);
    u = VectorOps.vector(0.0f, 0.0f, 1.0f);
    normVec = VectorOps.calcNormal(v, u);
    wallVec[TOP_WALL][0] = v;
    wallVec[TOP_WALL][1] = u;
    wallVec[TOP_WALL][2] = normVec;

    // Установить нижнюю стену в ее истинном положении
    trBottomWall = new Transform();
    trBottomWall.postTranslate(0.0f, -4.0f, -5.0f);
    trBottomWall.postRotate(-90, 1.0f, 0.0f, 0.0f);
    trBottomWall.postRotate(90, 0.0f, 0.0f, 1.0f);
    trBottomWall.postScale(5.0f, 5.0f, 5.0f);
    wall4.setTransform(trBottomWall);

    // Те же векторы что и у верхней стены
    wallVec[BOTTOM_WALL][0] = v;
    wallVec[BOTTOM_WALL][1] = u;
    wallVec[BOTTOM_WALL][2] = normVec;

    // Так что можем узнать их позже
    wall1.setUserID(LEFT_WALL);
    wall2.setUserID(RIGHT_WALL);
    wall3.setUserID(TOP_WALL);
    wall4.setUserID(BOTTOM_WALL);

    // Сделать их сталкивающимися
    wall1.setPickingEnable(true);
    wall2.setPickingEnable(true);
    wall3.setPickingEnable(true);
    wall4.setPickingEnable(true);

    // Добавить стены к группе игрового поля
    playingField.addChild(wall1);
    playingField.addChild(wall2);
    playingField.addChild(wall3);
    playingField.addChild(wall4);
}
/**
 * Создание ракетки
 */
private void createPaddle()
{
    // Создать план, используя нашу острый класс MeshFactory
```

```
    paddle = MeshFactory.createPlane("/res/paddle.png", PolygonMode.CULL_BACK);

    // Установить ракетку в ее начальном положении
    trPaddle = new Transform();
    trPaddle.postTranslate(paddleCoords[0], paddleCoords[1], paddleCoords[2]);
    paddle.setTransform(trPaddle);

    // Сделать ее сталкивающейся
    paddle.setPickingEnable(true);
    paddle.setUserID(PADDLE_WALL);

    // Добавить к группе игрового поля
    playingField = new Group();
    playingField.setUserID(PLAYING_FIELD);
    playingField.addChild(paddle);

    // Сделать ее векторы
    float[] v = {0.0f, 1.0f, 0.0f};
    float[] u = {1.0f, 0.0f, 0.0f};
    float[] normVec = VectorOps.calcNormal(v, u);
    wallVec[PADDLE_WALL][0] = v;
    wallVec[PADDLE_WALL][1] = u;
    wallVec[PADDLE_WALL][2] = normVec;
  }
  /**
   * Загружаем наш шар
   */
  private void loadBall() throws IOException
  {
     // Просто создаем инстанцию класса Ball
     ball = new Ball();
  }

  /** Чертим на экране
   */
    private void draw(Graphics g)
  {
    // Окутать все пробующим/ловящим блоком на всякий случай
try
{
 // Получить Graphics3D контекст
g3d = Graphics3D.getInstance();

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

// Очистка фона
g3d.clear(back);

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

      // Рендерить поле игры и шар
      g3d.render(playingField, identity);
      ball.render(g3d, playingField, wallVec, paddleCoords);

      // Проверить клавиши для движения ракетки
      if(key[UP])
      {
         paddleCoords[1] += 0.2f;
         if(paddleCoords[1] > 3.0f)
            paddleCoords[1] = 3.0f;
      }
```

```
     if(key[DOWN])
     {
        paddleCoords[1] -= 0.2f;
        if(paddleCoords[1] < -3.0f)
           paddleCoords[1] = -3.0f;
     }
     if(key[LEFT])
     {
        paddleCoords[0] -= 0.2f;
        if(paddleCoords[0] < -3.0f)
           paddleCoords[0] = -3.0f;
     }
     if(key[RIGHT])
     {
        paddleCoords[0] += 0.2f;
        if(paddleCoords[0] > 3.0f)
           paddleCoords[0] = 3.0f;
     }

     // Установить координаты ракетки
     trPaddle.setIdentity();
     trPaddle.postTranslate(paddleCoords[0], paddleCoords[1], paddleCoords[2]);
     paddle.setTransform(trPaddle);

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

    // Сделать некоторый старомодный 2D- рисунок
    if(!ball.isMoving())
    {
       g.setColor(0);
       g.drawString("Press fire to start!", 2, 2, Graphics.TOP | Graphics.LEFT);
    }
    else
    {
       int red = Math.min(255, ball.getBounces() * 12);
       g.setColor(red, 0, 0);
       g.drawString("Score: " + ball.getBounces(), 2, 2, Graphics.TOP | Graphics.LEFT);
    }
 }
/** Стартует холст, разжигая нить(thread)
  */
 public void start() {
    Thread myThread = new Thread(this);

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

    // Старт
    myThread.start();
 }
```

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

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

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

   // Уведомление о завершении
   done = true;
}
/**
 * @param e
 */
private void reportException(Exception e) {
   System.out.println(e.getMessage());
   System.out.println(e);
   e.printStackTrace();
}
/** Пауза в игре
 */
public void pause() {}

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

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

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

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

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

   if((keys & GameCanvas.LEFT_PRESSED) != 0)
```

```
            key[LEFT] = true;
        else
            key[LEFT] = false;

        if((keys & GameCanvas.RIGHT_PRESSED) != 0)
            key[RIGHT] = true;
        else
            key[RIGHT] = false;
    }

      /** Проверка запуска Нити
     */
    public boolean isRunning() { return running; }

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

VectorOps

```
/**
* Простой класс, который упрощает некоторую векторную математику
*/
public class VectorOps
{
  / ** Вычисляет продукт – точку двух векторов.
     * Считается, что гарантировано существование векторов v и u
     * в 3-мерной комнате.
     */
    public static float dotProduct(float[] v, float[] u)
    {
        return v[0] * u[0] + v[1] * u[1] + v[2] * u[2];
    }

  / ** Вычисляет продукт – пересечение двух векторов.
     * Считается, что гарантировано существование векторов v и u
     * в 3-мерной комнате.
     */
    public static float[] crossProduct(float[] v, float[] u)
    {
        float[] newVec = {v[1] * u[2] - v[2] * u[1], v[2] * u[0] - v[0] * u[2], v[0] * u[1] - v[1] * u[0]};
        return newVec;
    }

    /** Вычисляет длину двух векторов */
    public static float length(float[] v)
    {
        return (float)Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
    }

    /** Проецирует вектор v на вектор u */
    public static float[] project(float[] v, float[] u)
    {
        float lenU = length(u);
        float scalar = dotProduct(v, u) / (lenU * lenU);
        return scalarMul(scalar, u);
    }

    /** ** Вычисляет нормали плана, определенного вектором v и вектором u */
    public static float[] calcNormal(float[] v, float[] u)
    {
        float[] cross = crossProduct(v, u);
        return normalize(cross);
```

```
    }

    /** Нормализует вектор (например делает его длину = 1)*/
    public static float[] normalize(float[] v)
    {
        float len = length(v);
        float[] newVec = {v[0] / len, v[1] / len, v[2] / len};
        return newVec;
    }

    /** Умножает вектор на скаляр */
    public static float[] scalarMul(float scalar, float[] v)
    {
        float[] newVec = {v[0] * scalar, v[1] * scalar, v[2] * scalar};
        return newVec;
    }

    /** Вычитает два вектора. v - u */
    public static float[] sub(float[] v, float[] u)
    {
        float[] newVec = {v[0] - u[0], v[1] - u[1], v[2] - u[2]};
        return newVec;
    }

    /** Добавляет два вектора. v - u */
    public static float[] add(float[] v, float[] u)
    {
        float[] newVec = {v[0] + u[0], v[1] + u[1], v[2] + u[2]};
        return newVec;
    }

    /** Зеркалирует вектор v в плане, который имеет вектор нормали n */
    public static float[] mirror(float[] v, float[] n)
    {
        float[] u = VectorOps.project(v, n);
        return VectorOps.sub(v, VectorOps.scalarMul(2.0f, u));
    }

    /** Сделать простой конструктор для вектора */
    public static float[] vector(float x, float y, float z)
    {
        float[] v = {x, y, z};
        return v;
    }

    /** Сделано для отладки. Трансформирует вектор в String. */
    public static String toString(float[] v)
    {
        return "(" + v[0] + ", " + v[1] + ", " + v[2] + ")";
    }
}
```

MeshFactory

```
import javax.microedition.lcdui.Image;
import javax.microedition.m3g.Appearance;
import javax.microedition.m3g.Image2D;
import javax.microedition.m3g.IndexBuffer;
import javax.microedition.m3g.Mesh;
import javax.microedition.m3g.PolygonMode;
import javax.microedition.m3g.Texture2D;
import javax.microedition.m3g.TriangleStripArray;
import javax.microedition.m3g.VertexArray;
import javax.microedition.m3g.VertexBuffer;
```

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

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

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

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

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

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

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

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

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

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

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

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

Ball

```
import java.io.IOException;
import java.util.Random;
import javax.microedition.m3g.Graphics3D;
import javax.microedition.m3g.Group;
import javax.microedition.m3g.Loader;
import javax.microedition.m3g.Node;
import javax.microedition.m3g.Object3D;
import javax.microedition.m3g.RayIntersection;
import javax.microedition.m3g.Transform;
import javax.microedition.m3g.World;
/ **
 * Инкапсулирует наш шар. Держит только вектор движения и текущие координаты.
 * Также, вращает шар.
 */
public class Ball
{
    // Мешь шара
    Group ball = null;

    // Вектор движения шара
    float[] moveVec = null;

    // Трансформация шара
    Transform trBall = new Transform();

    // Вектор, используемый в вычислениях
    float[] coords;

    // Насколько шар вращался
```

```
   float rotated = 0.0f;

   // Число сильных ударов
   int bounces = 0;

   // RayIntersection для столкновения
   RayIntersection ri = new RayIntersection();

   /** Конструктор нашего шара и его трансформация */
   public Ball() throws IOException
   {
      // Использовать загрузчик для загрузки как Object3D
      Object3D[] ballObj = Loader.load("/res/rediBall.m3g");

      // Найти узел Мира (Группу)
      int i = 0;
      while(!(ballObj[i] instanceof World))
      {
         i++;
      }
      ball = (World)ballObj[i];

      // Позволим шару стартовать на ракетке
      trBall = new Transform();
      coords = VectorOps.vector(0.0f, 0.0f, 10.0f);
   }


     / **
    * Рендеринг шара. Также исполняет все необходимые вычисления
    * типа зеркалирование, вращение, и т.д...
    * Возвращает TRUE, если шар перелетел ракетка и игру закончена
    */
   public boolean render(Graphics3D g3d, Group walls, float[][][] wallVectors, float[] paddleCoords)
   {
      //Очистка трансформации
      trBall.setIdentity();

      // Проверить, двигаемся ли мы
      if(moveVec != null)
      {
       // Сначала немного вращайте шар
       trBall.postRotate(rotated += 1.0f, 1.0f, 1.0f, 1.0f);
       // Нормализовать вектор движения
          ri = new RayIntersection();
          float[] nMove = VectorOps.normalize(moveVec);

       // Смотри, если  есть любое столкновение
       if(walls.pick(-1, coords[0], coords[1], coords[2], nMove[0], nMove[1], nMove[2], ri) &&
ri.getDistance() <= 0.5f)
       {
          // Мы столкнулись, получаем поверхность, которую мы пересекли
          Node n = ri.getIntersected();

          // Исправить наш вектор движения зеркалированием
          moveVec = VectorOps.mirror(moveVec, wallVectors[n.getUserID()][2]);

          // Позволим пользователю иметь управление движением перемещением ракетки
          if(n.getUserID() == M3GCanvas.PADDLE_WALL)
          {
             // Добавить сверхскорость в зависимости от позиции мяч/ракетка
             float distX = (paddleCoords[0] - coords[0]) / 10.0f;
             float distY = (paddleCoords[1] - coords[1]) / 10.0f;
             moveVec[0] = Math.max(-0.3f, Math.min(moveVec[0] - distX, 0.3f));
```

```
            moveVec[1] = Math.max(-0.3f, Math.min(moveVec[1] - distY, 0.3f));

            // После 30 сильных ударов это должно быть невозможно быстро (ХА! Вызов!)
   moveVec[2] = moveVec[2] + 0.01f;

   // Увеличение числа сильных ударов
            bounces++;
         }
      }

      // Проверить для того, чтобы отскочить от экрана
      if(coords[2] >= -0.7f)
      {
         // Щелчок по экрану (то же самое что был у ракетки)
         moveVec = VectorOps.mirror(moveVec, wallVectors[M3GCanvas.PADDLE_WALL][2]);
      }

      // Перемещение шара
      coords[0] += moveVec[0];
      coords[1] += moveVec[1];
      coords[2] += moveVec[2];
     }

     // Если мы ушли (позволим шару пройти через ракетку)
     boolean quit = coords[2] <= -7.5f;
     rotated += 15.0f;

     // Устанавливаем трансформацию
     trBall.postTranslate(coords[0], coords[1], coords[2]);
     trBall.postRotate(rotated, 1.0f, 1.0f, 1.0f);

     // Наконец: рендерим шар
     g3d.render(ball, trBall);

     if(quit)
     {
        moveVec = null;
        coords = VectorOps.vector(0.0f, 0.0f, 10.0f);
     }

     return quit;
   }

   /** Стартуем шар в положении покоя в случайной позиции */
   public void start()
   {
      Random r = new Random();
      bounces = 0;
      moveVec = VectorOps.vector(-0.1f + r.nextFloat() * 0.2f, -0.1f + r.nextFloat() * 0.2f, 0.1f);
      coords = VectorOps.vector(-3.0f + r.nextFloat() * 6.0f, -3.0f + r.nextFloat() * 6.0f, -5.0f);
   }

   /** Возвращает true если шар перемещается */
   public boolean isMoving() { return moveVec != null; }

   /** Получаем число сильных ударов (для изящной печати счета) */
   public int getBounces() { return bounces; }
}
```

О Redikod

Redikod, из Мальмо(Malmo) в Швеции, - разработчик с 1997 сетевых и мобильных игр, и эта маленькая компания - теперь один из лидеров в скандинавской промышленности игр. Его непропорциональное влияние происходит от стратегических инициатив типа Скандинавского Потенциала Игр, ежегодной конференции, и скандинавского участия в E3 2006. Redikod уполномочен проектировать скандинавскую общественную систему поддержки и

финансирования разработок для развития игр, включая мобильный телефон, что, ожидаемое заключительное принятие этой системы произойдет этой осенью и войдет в силу в 2006. Но разработка 3D- и мульти-плеерных для мобильного телефона - их ежедневная работа. Более подробно на WEB-сайте Redikod>>.

P.S.