Модуль 3. Основы программирования Android приложений

*Тема 3.9. Введение в OpenGL

Оглавление

3.9. Введение в OpenGL

OpenGL (**Open** **G**raphics **L**ibrary) — спецификация, определяющая независимый от языка программирования платформонезависимый программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику.

OpenGL ориентируется на следующие две задачи:

- Скрыть сложности адаптации различных 3D-ускорителей, предоставляя разработчику единый API (от англ. **A**pplication **P**rogramming **I**nterface - интерфейс программирования приложений).
- Скрыть различия в возможностях аппаратных платформ, требуя реализации недостающей функциональности с помощью программной эмуляции.

Основным принципом работы OpenGL является получение наборов векторных графических примитивов в виде точек, линий и треугольников с последующей математической обработкой полученных данных и построением растровой картинки на экране и/или в памяти. Векторные трансформации и растеризация выполняются графическим конвейером (graphics pipeline). Абсолютное большинство команд OpenGL попадают в одну из двух групп: либо они добавляют графические примитивы на вход в конвейер, либо конфигурируют конвейер на различное исполнение трансформаций.

Подмножество графического интерфейса OpenGL, разработанное специально для встраиваемых систем — мобильных телефонов, карманных компьютеров, игровых консолей, — получило название OpenGL ES (OpenGL for Embedded Systems — OpenGL для встраиваемых систем). OpenGL ES определяется и продвигается консорциумом Khronos Group, в который входят производители программного и аппаратного обеспечения, заинтересованные в открытом API для графики и мультимедиа.

В операционной системе Android поддерживается несколько версий OpenGL ES API:

- OpenGL ES 1.0 и 1.1 - поддерживается в Android версии 1.0 и выше.
- OpenGL ES 2.0 - поддерживается в Android 2.2 (API 8) и выше.
- OpenGL ES 3.0 - поддерживается в Android 4.3 (API 18) и выше.
- OpenGL ES 3.1 - поддерживается в Android 5.0 (API 21) и выше.

Разработка приложений с использованием OpenGL ES версий 2.0 и 3.0 имеет много общего, т.к. версия 3.0 является расширением версии 2.0 за счёт раз□дополнительных возможностей. Использование версии OpenGL ES 1.x значительно отличается от более поздних версий OpenGl ES API, поэтому разработчикам нужно учитывать несколько факторов при выборе используемого в своём приложении API:

1. Быстродействие - как правило, OpenGL ES 2.0 и старше обеспечивают более высокое быстродействие графики, чем версия 1.x. Однако на быстродействие графики также оказывает влияние само устройство, т.к. графический конвейер может быть реализован по-раному на аппаратном уровне производителем устройства.
2. Совместимость устройств - разработчики должны учитывать типы устройств, версии системы Android и, соответственно, версии OpenGL ES API потенциальных пользователей приложения.

3. Удобство программирования - OpenGL ES 1.x предоставляет ограниченный набор функций, которые недоступны в OpenGL ES 2.0 и выше. Начинающие разработчики могут найти разработку с использованием версий OpenGL ES 1.x более удобной и быстрой.
4. Управление графическим конвейером - версии OpenGL ES 2.0 И 3.x предоставляют разработчику больше возможностей для программирования графического конвейера (например, шейдеры), благодаря чему появляется возможность создания таких эффектов, которые было бы весьма трудно реализовать с использованием OpenGL EX 1.x API.
5. Поддержка текстур - OpenGL ES 3.0 поддерживает сжатие текстур ETC2, в то время как версии 1.x и 2.0 поддерживают только ETC1. Использование ETC2 выгодно отличается от ETC1 поддержкой прозрачности, для реализации которой в версиях 1.x и 2.0 нужно выбирать другие форматы сжатия, поддерживаемые целевым устройством.

Версии OpenGL ES 1.x считаются морально устаревшими. Версии OpenGL ES 3.x API обратно совместимы с OpenGL ES 2.0 API, то есть приложение, в котором реализована поддержка OpenGL ES 2.0, будет без проблем выполняться на устройствах с версией Android 4.3 и выше. По этой причине в рамках данной темы будет рассматриваться разработка приложений с использованием OpenGL ES версии 2.0.

Если приложение использует возможности OpenGL, доступные не на всех устройствах, необходимо включить требование к поддерживаемой на устройстве версии OpenGL ES в файл AndroidManifest.xml для версии OpenGL ES 2.0:

```
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

Указание данного требования приведёт к тому, что Google Play не позволит установить приложение на устройствах, которые не поддерживают OpenGL ES 2.0. Для OpenGL ES 3.0 аналогичеое указание на требуемую версию поддерживаемого API выглядит следующим образом:

```
<uses-feature android:glEsVersion="0x00030000" android:required="true" />
```

Для версии OpenGL ES 3.1:

```
<uses-feature android:glEsVersion="0x00030001" android:required="true" />
```

3.9.1. Подготовка Activity к созданию графических объектов с помощью OpenGL

Для того, чтобы можно было в Activity создавать графические объекты с помощью OpenGL, необходимо создать специальное представление (View):

```
private GLSurfaceView mGLSurfaceView;
```

GLSurfaceView обеспечивает работу в отдельном потоке (thread) для визуализации OpenGL. Таким образом, UI поток у нас не занят. GLSurfaceView поддерживает непрерывный или по требованию рендеринг (обновление, перерисовка содержимого на экране), поддерживает настройки экрана для EGL — интерфейса между OpenGL и оконной системой Android.

Прежде чем использовать методы OpenGL ES 2.0, необходимо выяснить, поддерживается ли данным устройством этот API. Для этого в метод onCreate добавляем строки:

```
final ActivityManager activityManager = (ActivityManager)
     getSystemService(Context.ACTIVITY_SERVICE);
final ConfigurationInfo configurationInfo =
     activityManager.getDeviceConfigurationInfo();
final boolean supportsEs2 = configurationInfo.reqGlEsVersion >= 0x20000;
```

Мы обращаемся к Диспетчеру Активностей и с его помощью получаем данные о конфигурации устройства. Среди этих данных интересующая нас информация - поддерживает ли устройство OpenGL ES 2.0 - содержится в поле reqGlEsVersion, значение которого мы и проверяем.

Если OpenGL ES 2.0 поддерживается, то константе supportsEs2 будет присвоено значение True. В этом случае можно создать экземпляр класса GLSurfaceView:

```
if (supportsEs2) {
    mGLSurfaceView = new GLSurfaceView(this);
```

указать версию OpenGL ES, с которой мы будет работать в данном приложении (версия 2.0):

```
mGLSurfaceView.setEGLContextClientVersion(2);
```

Далее необходимо установить визуализатор (Renderer) - интерфейс, который отвечает за обращение к OpenGL для прорисовки изображения:

```
     mGLSurfaceView.setRenderer(new NewRenderer());
}
else {
     return;
}
```

Таким образом метод setRenderer устанавливает визуализатор для отображения в представлении mGLSurfaceView. В качестве параметра при вызове это метода указан класс NewRenderer, который мы создадим позже и который будет реализовывать визуализатор.

Ещё одна строка, которую необходимо указать в методе onCreate:

```
setContentView(mGLSurfaceView);
```

устанавливает mGLSurfaceView в качестве представления текущей активности.

Когда у активности в соответствии с её жизненным циклом будут выполняться методы onResume и onPause, необходимо, чтобы соответствующие методы вызывались и у GL-поверхности:

```
@Override
protected void onResume() {
        super.onResume();
        mGLSurfaceView.onResume();
}
```

```
@Override
protected void onPause() {
        super.onPause();
        mGLSurfaceView.onPause();
}
```

Добавление этих методов и обращение из них к методам GL-поверхности необходимо для того, чтобы запускать и приостанавливать поток визуализации на различных этапах жизненного цикла активности.

Следующим этапом в разработке приложения для рисования с использованием OpenGL является определение визуализатора. Визуализатор - это специальный класс, у которого есть три важных метода:

1. **public void onSurfaceCreated(GL10 glUnused, EGLConfig config)** — этот метод вызывается, когда поверхность создается. Это происходит при запуске приложения, при переключении пользователя обратно в данную активити. То есть данный меод может вызываться несколько раз во время работы приложения.
2. **public void onSurfaceChanged(GL10 glUnused, int width, int height)** - этот метод вызывается после создания поверхности и каждый раз, когда происходит изменение размера поверхности, а также при переключении с портретной ориентации на альбомную и наоборот.
3. **public void onDrawFrame(GL10 glUnused)** - этот метод вызывается каждый раз, когда необходимо прорисовать новый кадр.

Параметры glUnused относятся к OpenGL ES 1.0 API и являются рудиментом. Если бы создавали визуализатор в версии OpenGL ES 1.0 API, мы бы их использовали.

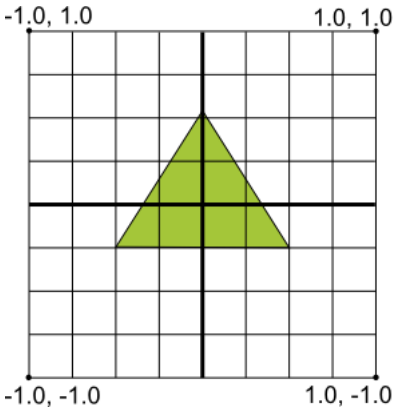
Создадим класс NewRenderer, к которому мы обращаемся при создании визулизатора:

```
class NewRenderer implements GLSurfaceView.Renderer
```

В OpenGL ES 2 объект для отображения описывается в виде массива чисел. В нашем примере мы отобразим три треугольника.

```
public NewRenderer() {
   final float[] triangle1VerticesData = {
           // X, Y, Z,
           // R, G, B, A
           -0.5f, -0.25f, 0.0f,
           1.0f, 0.0f, 0.0f, 1.0f,


           0.5f, -0.25f, 0.0f,
           0.0f, 0.0f, 1.0f, 1.0f,


           0.0f, 0.559016994f, 0.0f,
           0.0f, 1.0f, 0.0f, 1.0f};
```

Каждая группа, состоящая из семи чисел типа float, задаёт одну точку треугольника: первые три цифры соответствуют координатам X, Y, Z, последние четыре цифры опредеют цвет: красный R, зеленый G, синий B и альфа A (прозрачность). Двухмерная координатная модель OpenGL имеет следующий вид:



Аналогичным образом описываем второй и третий треугольники:

```
final float[] triangle2VerticesData = {
      // X, Y, Z,
      // R, G, B, A
      -0.5f, -0.25f, 0.0f,
      1.0f, 1.0f, 0.0f, 1.0f,

      0.5f, -0.25f, 0.0f,
      0.0f, 1.0f, 1.0f, 1.0f,

      0.0f, 0.559016994f, 0.0f,
      1.0f, 0.0f, 1.0f, 1.0f};

final float[] triangle3VerticesData = {
      // X, Y, Z,
      // R, G, B, A
      -0.5f, -0.25f, 0.0f,
      1.0f, 1.0f, 1.0f, 1.0f,

      0.5f, -0.25f, 0.0f,
      0.5f, 0.5f, 0.5f, 1.0f,

      0.0f, 0.559016994f, 0.0f,
      0.0f, 0.0f, 0.0f, 1.0f};
```

В Android Studio и Eclipse используется язык Java для разработки программ под Android, но родная реализация OpenGL ES2 написана и выполняется на языке Си. Поэтому прежде чем передать данные об объектах на прорисовку, необходимо преобразовать их в понятный для OpenGL ES2 вид. Для этого перед конструтором класса добавим несколько срок, создающих буферы для хранения данных о треугольниках:

```
private final FloatBuffer mTriangle1Vertices;
private final FloatBuffer mTriangle2Vertices;
private final FloatBuffer mTriangle3Vertices;
```

Здесь же укажем количество байт, занимаемых одним числом:

```
private final int mBytesPerFloat = 4;
```

Java и нативная система (native system) не могут хранить байты в одинаковом порядке, поэтому используется специальный набор классов буфера и создаётся ByteBuffer, достаточно большой для хранения всех данных. Далее мы используем его для хранения наших байтов в собственном порядке. После этого мы преобразовываем его в FloatBuffer, и теперь мы можем использовать его для хранения данных в виде чисел с плавающей запятой. Всё это существляется в следующих строках кода, инициализирующих буферы (их необходимо добавить в конце конструктора):

```
mTriangle1Vertices = ByteBuffer.allocateDirect(triangle1VerticesData.length
     * mBytesPerFloat).order(ByteOrder.nativeOrder()).asFloatBuffer();
mTriangle2Vertices = ByteBuffer.allocateDirect(triangle2VerticesData.length
     * mBytesPerFloat).order(ByteOrder.nativeOrder()).asFloatBuffer();
mTriangle3Vertices = ByteBuffer.allocateDirect(triangle3VerticesData.length
     * mBytesPerFloat).order(ByteOrder.nativeOrder()).asFloatBuffer();

mTriangle1Vertices = ByteBuffer.allocateDirect(triangle1VerticesData.length
      * mBytesPerFloat).order(ByteOrder.nativeOrder()).asFloatBuffer();
mTriangle2Vertices = ByteBuffer.allocateDirect(triangle2VerticesData.length
      * mBytesPerFloat).order(ByteOrder.nativeOrder()).asFloatBuffer();
mTriangle3Vertices = ByteBuffer.allocateDirect(triangle3VerticesData.length
      * mBytesPerFloat).order(ByteOrder.nativeOrder()).asFloatBuffer();
```

Наконец, мы копируем наш массив в буфер и перемещаемся на позицию 0 в этом массиве.

```
mTriangle1Vertices.put(triangle1VerticesData).position(0);
mTriangle2Vertices.put(triangle2VerticesData).position(0);
mTriangle3Vertices.put(triangle3VerticesData).position(0);
```

Возможно эта последовательность может показаться слегка запутанной. Но помните, что этот шаг очень важен перед тем, как передать наши данные в OpenGL.

Наш буфер данных теперь готов к использованию и передаче своих значений в OpenGL.

3.9.2. Создание матрицы вида

Добавим в наш класс метод onSurfaceCreated, который будет выполнять действия сразу после создания GL-поверхности.

```
@Override
public void onSurfaceCreated(GL10 glUnused, EGLConfig config){
```

Установим светло-серый цвет фона:

```
GLES20.glClearColor(0.5f, 0.5f, 0.5f, 0.5f);
```

и определим полжение точки наблюдения в пространстве. Используем для этого матрицу вида.

Матрица - это двумерный массив элемнтов. В OpenGL существует несколько видов матриц:

1. Матрица модели. Эта матрица используется для размещения объекта где-то в пространстве. Например, если у вас есть модель автомобиля, и вы хотите расположить ее на 1000 метров на восток, вы будете использовать для этого матрицу модели.
2. Матрица вида. Эта матрица представляет собой камеру, точку наблюдения за объектами. Если мы хотим посмотреть на наш автомобиль, который расположен на 1000 метров на

восток, нам придется самим переместиться на 1000 метров на восток. Мы используем для этого матрицу вида.

3. Матрица проекции. Так как наши экраны плоские, нам необходимо проделать некую трансформацию трёхмерного виртуального пространства, построенного с использованием OpenGL, чтобы получить видимое изображение на плоской поверхности экрана наших объемных моделей с учетом 3D перспективы. Для этого и используется матрица проекции.

Для работы с матрицами в Android есть класс Matrix, которым мы воспользуемся, чтобы создать матрицу вида.

Определим положение камеры в пространстве:

```
final float eyeX = 0.0f;
final float eyeY = 0.0f;
final float eyeZ = 1.5f;
```

Определим расстояние, на которое камера может смотреть:

```
final float lookX = 0.0f;
final float lookY = 0.0f;
final float lookZ = -5.0f;
```

Определим вектор. Это направление, в котором “смотрит” камера:

```
final float upX = 0.0f;
final float upY = 1.0f;
final float upZ = 0.0f;
```

Создаём матрицу. Для этого в метод onSurfaceCreated добавим обращение к методу setLookAtM, :

```
Matrix.setLookAtM(mViewMatrix, 0, eyeX, eyeY, eyeZ, lookX, lookY, lookZ,
        upX, upY, upZ);
```

а в поля данного класса добавим описание массива:

```
private float[] mViewMatrix = new float[16];
```

Таким образом, OpenGL существенно облегчает нам задачу прорисовки всех объектов в зависимости от того, с какого ракурса мы на эти объекты наблюдаем. С помощью матрицы вида мы просто указываем, откуда мы смотрим, в каком направлении и как далеко мы видим объекты виртуального пространства, а всю прорисовку объектов в правильном ракурсе и взаимном расположении относительно друг друга OpenGl сделает сам.

3.9.3. Шейдеры

В OpenGL ES2 всё, что мы хотим отобразить на экране, сначала нужно пропустить через вершинный и фрагментный (пиксельный) шейдеры. Шейдеры - это программы, в которых мы осуществляем почти всю логику по обработке наших моделей. Вершинные шейдеры занимаются обработкой каждой вершины и нахождение её координат с учетом матрицы модели и прочих, зависящих от задумки программиста, условий. Для этого вершинный шейдер получает все параметры вершины (координаты, цвет, данные по текстуре т.д.). Также основная программа может передать шейдеру любые другие, определяемые программистом, параметры. Далее

результат этих операций передаются и используются в фрагментном шейдере, который осуществляет дополнительные расчеты по каждому пикселю.

Пиксельные шейдеры имеют своей целью расчет цвета каждого пикселя и значения глубины или вынесение решения о том, что пиксель выводить не надо. Пиксельный шейдер получает на входе текстурные координаты, соответствующие обрабатываемой точке, примешиваемый цвет и, возможно, некоторые другие параметры. На основании этих данных он расчитывает цвет точки и завершает работу.

Определим вершинный шейдер следующим образом:

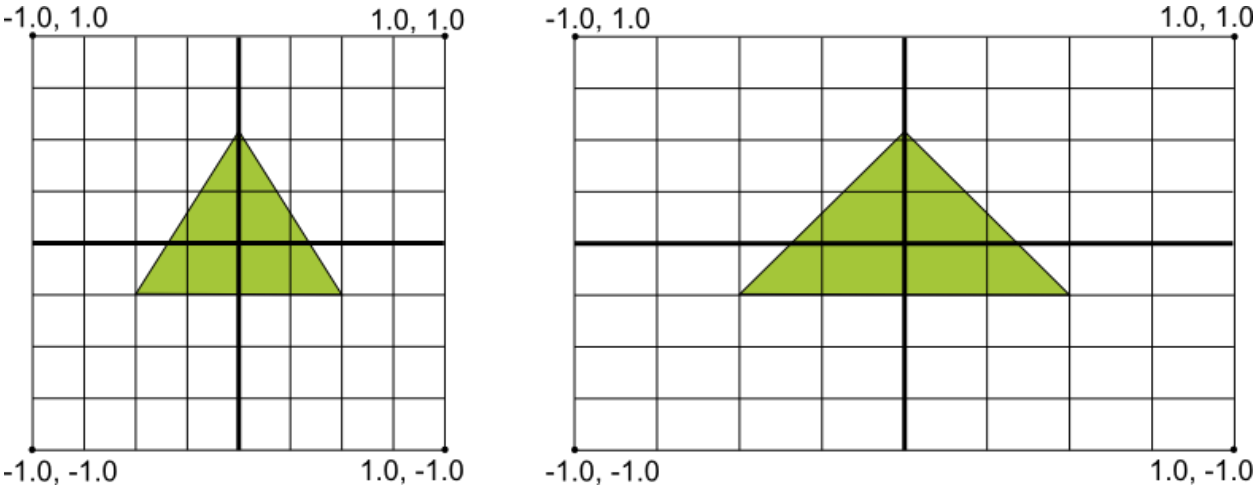
```
// Константа отвечающая за комбинацию матриц МОДЕЛЬ/ВИД/ПРОЕКЦИЯ
final String vertexShader = "uniform mat4 u_MVPMatrix;      \n"

       + "attribute vec4 a_Position;   \n"  //Информация о положении вершин.
       + "attribute vec4 a_Color;      \n"  //Информация о цвете вершин.
       //Это будет передано в фрагментный шейдер
       + "varying vec4 v_Color;        \n"
       //Начало программы вершинного шейдера
       + "void main()                  \n"
       + "{                            \n"
       //Передаем цвет для фрагментного шейдера. Он будет интерполирован
       //для всего треугольника.
       + "   v_Color = a_Color;        \n"
   //gl_Position специальная переменная, используемая для хранения конечного
положения
       + "   gl_Position = u_MVPMatrix \n"
       //Умножаем вершины на матрицу для получения конечного положения
       //в нормированных координатах экрана
       + "               * a_Position;   \n"
       + "}                            \n";
```

Каждый шейдер в основном состоит из точки входа (начала программы), выхода, и самой логики программы. Сначала определим константу (uniform) u_MVPMatrix, которая представляет собой комбинированную матрицу, содержащую значения всех преобразований. Эта матрица используется для всех вершин при проецировании их на экран. Потом мы определяем две константы-атрибуты — положение вершин и цвет. Значения в них будут переданы из числового байт-буфера, который описан ранее. В нём уже содержатся данные о положении и цвете для каждой вершины. Далее определяем переменные, значения которых описывают наши треугольники, затем передаем их значения в фрагментный шейдер, в котором над ними проводятся необходимые операции, которые касаются каждого пикселя.

Допустим, мы определили, что у треугольника вершины должны быть красного, зеленого и синего цвета. А размером он должен быть таким, чтобы занять 10 пикселей на экране. Когда программа фрагментного шейдера будет запущена, в нём уже будут значения переменных, отвечающих за цвет для каждого пикселя. В одной вершине значение будет соответствовать красному цвету, но посредине между красной и синей вершиной значение будет интерполироваться (усредняться), что отобразится в виде пурпурного цвета.

Последнее, что мы сделали в программе вершинного шейдера - сообщили OpenGL окончательное положение вершин, расположенных на экране, в виде нормированных координат. Это необходимо сделать по следующей причине. Одной из проблем в отображении графики на устройствах с ОС Android является то, что их экраны могут сильно различаться в размерах и форме.

Система координат OpenGL - квадратная, а объекты, расположенные в этой системе координат, должны отображаться на неквадратном экране так, как будто экран квадратный.



На рисунке показана однородная система координат, используемая в OpenGL (слева) и как эти координаты в действительности располагаются на экране Android-устройства в альбомной ориентации (справа). Чтобы решить эту проблему, для преобразования координат таким образом, чтобы графические объекты выглядели правильно на любом экране, используется матрица проекции, которая применяется к графическому конвейеру.

Далее приведен пример программы фрагментного шейдера:

```
final String fragmentShader =
       "precision mediump float;     \n"  //Устанавливаем по умолчанию
среднюю
                //точность для переменных.
                //Максимальная точность в фрагментном
                //шейдере не нужна.
                + "varying vec4 v_Color;        \n"  //Цвет вершинного
шейдера,
                //интерполированный для фрагмента.
                + "void main()                  \n"  //Точка входа для
фрагментного
                //шейдера.
                + "{                            \n"
                + "   gl_FragColor = v_Color;   \n"  //Передаем значения
цветов напрямую
                + "}                            \n"; //через конвейер.
```

В этом фрагментном шейдере мы получаем значения цветов из вершинного шейдера, а потом передаем их прямо в OpenGL. Каждая точка уже преобразована в определенное количество пикселей, с учетом маштаба экрана, так как в фрагментном шейдере обрабатывается каждый пиксель, который в последствии будет нарисован.

Загрузим программы наших шейдеров в OpenGL.

```
// Загрузка вершинного шейдера.
int vertexShaderHandle = GLES20.glCreateShader(GLES20.GL_VERTEX_SHADER);

if (vertexShaderHandle != 0) {
   // Передаем в шейдер данные.
   GLES20.glShaderSource(vertexShaderHandle, vertexShader);

   // Компиляция шейреда.
   GLES20.glCompileShader(vertexShaderHandle);

   // Получаем результат компиляции.
   final int[] compileStatus = new int[1];
   GLES20.glGetShaderiv(vertexShaderHandle, GLES20.GL_COMPILE_STATUS,
           compileStatus, 0);

   // Если компиляция не удалась, удаляем шейдер.
   if (compileStatus[0] == 0) {
       GLES20.glDeleteShader(vertexShaderHandle);
       vertexShaderHandle = 0;
   }
}

if (vertexShaderHandle == 0) {
   throw new RuntimeException("Error creating vertex shader.");
}

// Аналогичные действия выполняем с фрагментным шейдером.
int fragmentShaderHandle = GLES20.glCreateShader(GLES20.GL_FRAGMENT_SHADER);
if (fragmentShaderHandle != 0) {
   // Передаем в шейдер данные.
   GLES20.glShaderSource(fragmentShaderHandle, fragmentShader);

   // Компиляция шейреда.
   GLES20.glCompileShader(fragmentShaderHandle);

   // Получаем результат компиляции.
   final int[] compileStatus = new int[1];
   GLES20.glGetShaderiv(fragmentShaderHandle, GLES20.GL_COMPILE_STATUS,
           compileStatus, 0);

   // Если компиляция не удалась, удаляем шейдер.
   if (compileStatus[0] == 0) {
       GLES20.glDeleteShader(fragmentShaderHandle);
       fragmentShaderHandle = 0;
   }
}

if (fragmentShaderHandle == 0) {
   throw new RuntimeException("Error creating fragment shader.");
}
```

Далее необходимо объединить вершинный и фрагментый шейдеры в одну программу, прежде чем мы сможем их использовать. Мы создаём новый объект программы OpenGL и если это нам удаётся, мы подключаем к программе наши шейдеры, а также такие атрибуты, как положение и цвет, после чего объединяем шейдеры.

```
// Создаем программу OpenGL и получаем ссылку на него.
int programHandle = GLES20.glCreateProgram();

if (programHandle != 0) {
   // Подключаем вершинный шейдер к программе.
   GLES20.glAttachShader(programHandle, vertexShaderHandle);

   // Подключаем фрагментный шейдер к программе.
   GLES20.glAttachShader(programHandle, fragmentShaderHandle);

   // Подключаем атрибуты цвета и положения.
   GLES20.glBindAttribLocation(programHandle, 0, "a_Position");
   GLES20.glBindAttribLocation(programHandle, 1, "a_Color");

   // Объединяем оба шейдера в программе.
   GLES20.glLinkProgram(programHandle);

   // Проверяем статус ссылки на программу.
   final int[] linkStatus = new int[1];
   GLES20.glGetProgramiv(programHandle, GLES20.GL_LINK_STATUS,
           linkStatus, 0);

   // Если ссылку не удалось получить, удаляем программу.
   if (linkStatus[0] == 0) {
       GLES20.glDeleteProgram(programHandle);
       programHandle = 0;
   }
}

if (programHandle == 0) {
   throw new RuntimeException("Error creating program.");
}
```

Далее в методе onSurfaceCreated необходимо определить указатели на атрибуты шейдера, которыми мы воспользуемся позже:

```
mMVPMatrixHandle = GLES20.glGetUniformLocation(programHandle,
       "u_MVPMatrix");
mPositionHandle = GLES20.glGetAttribLocation(programHandle, "a_Position");
mColorHandle = GLES20.glGetAttribLocation(programHandle, "a_Color");
```

и сообщить OpenGL, чтобы при визуализации он использовал созданную ранее программу. Закрывающая фигурная скобка завершает метод onSurfaceCreated:

```
   GLES20.glUseProgram(programHandle);
}
```

В полях класса необходимо описать соответствующие переменные:

```
// Используется для передачи матрицы преобразований.
private int mMVPMatrixHandle;
// Используется для передачи информации о позиции модели.
private int mPositionHandle;
// Используется для передачи информации о цвете модели.
private int mColorHandle;
```

3.9.4. Матрица проекции и матрица моделей

Когда происходят изменения GL-поверхности, вызывается метод onSurfaceChanged. В этом методе необходимо определить матрицу проекции, которая используется для преобразования трёхмерной сцены OpenGL в плоское изображение, которое выводится на экран.

```
@Override
public void onSurfaceChanged(GL10 glUnused, int width, int height) {
   // Устанавливаем окно OpenGL того же размера, что и поверхность экрана.
   GLES20.glViewport(0, 0, width, height);


   // Создаем новую матрицу проекции. Высота остается та же,
   // а ширина будет изменяться в соответствии с соотношением сторон.
   final float ratio = (float) width / height;
   final float left = -ratio;
   final float right = ratio;
   final float bottom = -1.0f;
   final float top = 1.0f;
   final float near = 1.0f;
   final float far = 10.0f;


   Matrix.frustumM(mProjectionMatrix, 0, left, right, bottom, top,
           near, far);
}
```

В полях класса описываем матрицу проекции:

```
private float[] mProjectionMatrix = new float[16];
```

и матрицу моделей, которая понадобится нам далее. Эта матрица используется для перемещения моделей из объектного пространства (где каждая модель находится в центре своей системы координат) в пространство мира.

```
private float[] mModelMatrix = new float[16];
```

Добавляем метод onDrawFrame, в котором модели будут прорисовываться на экране. Перед началом рисования очистим экран, а затем заставим модели плавно вращаться, используя для задания угла поворота текущее время, с периодом вращения 10 секунд. После этого прорисовываем треугольники в разном положении.

```
@Override
public void onDrawFrame(GL10 glUnused) {
   GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);


   // Делаем полный оборот при вращении за 10 секунд.
```

```
    long time = SystemClock.uptimeMillis() % 10000L;
    float angleInDegrees = (360.0f / 10000.0f) * ((int) time);


    // Рисуем первый треугольник плоскостью к нам.
    Matrix.setIdentityM(mModelMatrix, 0);
    Matrix.rotateM(mModelMatrix, 0, angleInDegrees, 0.0f, 0.0f, 1.0f);
    drawTriangle(mTriangle1Vertices);


    // Рисуем второй треугольник повёрнутый параллельно земле.
    Matrix.setIdentityM(mModelMatrix, 0);
    Matrix.translateM(mModelMatrix, 0, 0.0f, -1.0f, 0.0f);
    Matrix.rotateM(mModelMatrix, 0, 90.0f, 1.0f, 0.0f, 0.0f);
    Matrix.rotateM(mModelMatrix, 0, angleInDegrees, 0.0f, 0.0f, 1.0f);
    drawTriangle(mTriangle2Vertices);


    // Рисуем третий треугольник повёрнутый своей плоскостью влево.
    Matrix.setIdentityM(mModelMatrix, 0);
    Matrix.translateM(mModelMatrix, 0, 1.0f, 0.0f, 0.0f);
    Matrix.rotateM(mModelMatrix, 0, 90.0f, 0.0f, 1.0f, 0.0f);
    Matrix.rotateM(mModelMatrix, 0, angleInDegrees, 0.0f, 0.0f, 1.0f);
    drawTriangle(mTriangle3Vertices);
}
```

Каждый раз, когда вы хотите что-то анимировать на экране, лучше всего использовать время для задачи движения вместо фиксированной частоты кадров, т.к. на различных устройствах может отличаться скорость анимации при задании фиксированной частоты кадров в зависимости от производительности процессора.

Для прорисовки треугольников мы вызываем метод drawTriangle - фактически именно в нём происходит вывод изображения на экран. Добавим новые поля и метод drawTriangle в класс NewRenderer.

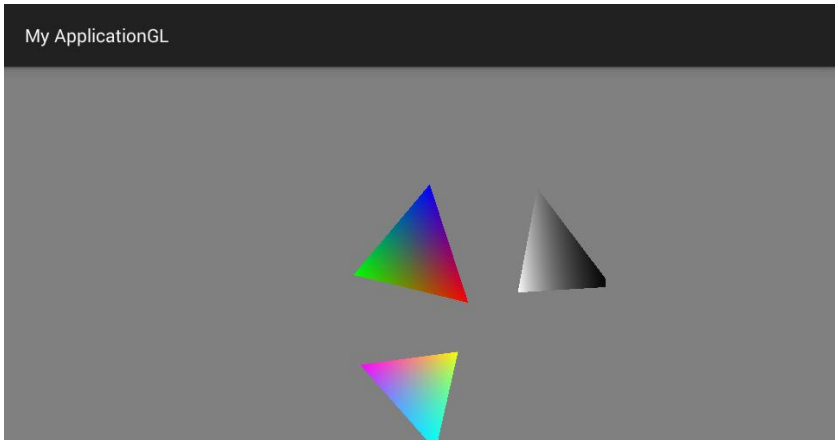
```
/* Выделяем массив для хранения объединеной матрицы. Она будет передана в
программу шейдера. */
private float[] mMVPMatrix = new float[16];


// Количество элементов в вершине.
private final int mStrideBytes = 7 * mBytesPerFloat;


// Смещение для данных о позиции.
private final int mPositionOffset = 0;


// Размер данных о позиции в элементах.
private final int mPositionDataSize = 3;


// Смещение для данных о цвете.
private final int mColorOffset = 3;


// Размер данных о цвете в элементах.
private final int mColorDataSize = 4;
```

```
private void drawTriangle(final FloatBuffer aTriangleBuffer) {
    // Передаем информацию о расположении.
    aTriangleBuffer.position(mPositionOffset);
    GLES20.glVertexAttribPointer(mPositionHandle, mPositionDataSize,
            GLES20.GL_FLOAT, false, mStrideBytes, aTriangleBuffer);
    GLES20.glEnableVertexAttribArray(mPositionHandle);


    // Передаем информацию о цвете.
    aTriangleBuffer.position(mColorOffset);
    GLES20.glVertexAttribPointer(mColorHandle, mColorDataSize,
            GLES20.GL_FLOAT, false, mStrideBytes, aTriangleBuffer);
    GLES20.glEnableVertexAttribArray(mColorHandle);


    // Перемножаем матрицу ВИДА на матрицу МОДЕЛИ и сохраняем результат
    // в матрицу MVP (которая теперь содержит модель*вид).
    Matrix.multiplyMM(mMVPMatrix, 0, mViewMatrix, 0, mModelMatrix, 0);


    // Перемножаем матрицу модели-вида на матрицу проекции, сохраняем
    // в MVP матрицу (которая теперь содержит модель*вид*проекцию).
    Matrix.multiplyMM(mMVPMatrix, 0, mProjectionMatrix, 0, mMVPMatrix, 0);


    GLES20.glUniformMatrix4fv(mMVPMatrixHandle, 1, false, mMVPMatrix, 0);
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, 3);
}
```

В самом начале мы определили числовые буферы, когда первоначально создавали визуализацию. Теперь нужно сообщить OpenGL, чтобы он обрабатывал данные с помощью GLES20.glVertexAttribPointer() - массива, содержащего данные об атрибутах вершин. Устанавливаем в числовом буфере смещение в элементах относительно начального элемента буфера. Сообщаем OpenGL, чтобы он использовал эти данные, передал их в вершинный шейдер и применил всё к атрибутам. Мы также должны сообщить OpenGL количество элементов (шагов) между значениями, описывающими каждую вершину.

Шаг должен быть определен в байтах. Пусть у нас есть 7 элементов (3 описывают положение вершины и 4 - её цвет) между вершинами, у нас фактически занято 28 байт, так как каждое число с плавающей точкой занимает 4 байта. Если мы забудем про этот шаг, у нас в процессе выполнения программы не будет никаких ошибок, но на экране будет пусто.

Заканчивается метод drawTriangle тем, что конечная матрица передаётся в вершинный шейдер, после чего треугольники прорисовываются на экране.



Задание 3.9.1

В разработанном приложении измените скорость анимации, положение вершин и их цвет.

Список источников

1. https://ru.wikipedia.org/wiki/OpenGL
2. https://ru.wikipedia.org/wiki/OpenGL_ES
3. http://www.learnopengles.com/android-lesson-one-getting-started
4. http://dedfox.com/izuchaem-opengl-es2-pod-android-urok-1-samoe-nachalo
5. http://nappy.it-forge.net/downloads/shaders.pdf
6. Brothaler K. OpenGL ES 2 for Android. A Quick-Start Guide

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Ребро Вадиму Владимировичу.