Лабораторная работа № 7 по теме:
«Работа с OpenGL ES»
Теоретическая часть

Технология OpenGL была разработана компанией Silicon Graphics в 1992 году. Она обеспечивает унифицированный интерфейс для программистов, который позволяет пользоваться возможностями аппаратного обеспечения любого производителя при разработке приложений, использующих двумерную и трёхмерную компьютерную графику.

Так как технология OpenGL была разработана для настольных компьютеров, она слишком громоздка, чтобы подойти мобильным устройствам. Поэтому в Android применяется разновидность OpenGL, которая называется OpenGL for Embedded Systems.

OpenGL ES (Open Graphics Library for Embedded Systems — открытая графическая библиотека для встраиваемых систем) — подмножество графического интерфейса OpenGL, использующееся на мобильных устройствах.

OpenGL ES определяется и продвигается консорциумом Khronos Group, в который входят производители программного и аппаратного обеспечения, таких как Google, Intel Corporation, AMD, NVidia, Apple Inc., Samsung Electronics, Sony, Huawei, Valve Corporation, Epic Games и другие компании, заинтересованные в открытом API для графики и мультимедиа.

Каждый язык имеет собственные языковые привязки к OpenGL ES. Язык привязки Java определен в Java Specification Request (JSR) 239. Android следует этому стандарту настолько точно, насколько это возможно.

Некоторые дешевые устройства, которые работают на Android, могут не иметь аппаратного обеспечения, поддерживающего трехмерную графику. Однако программный интерфейс OpenGL у них есть. В таком случае все 3D-функции эмулируются программно. Программа,

использующая OpenGL ES, будет работать, но гораздо медленнее, чем на устройствах с аппаратным ускорением 3D. По этой причине желательно предоставлять пользователям возможность отключать определенные детали и спецэффекты, которые занимают ресурсы, но не являются необходимыми для программы.

Платформа Android поддерживает несколько версий OpenGL ES:

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

Сейчас версия 1.0 считается устаревшей и используются версии 2.0, 3.0 и 3.1. Эти версии по API существенно отличаются от 1.0 и несовместимы с ней. В OpenGL ES 3.x есть обратная совместимость с OpenGL ES 2.0. Для использования OpenGL ES 3.0 на устройстве должен быть реализован графический конвейер.

Основные концепции 3D-графики

Когда человек смотрит в окно, свет отражается от объектов, находящихся за окном, проходит через оконное стекло, и человек воспринимает изображение объектов, находящихся за окном. В понятиях 3D-графики, сцена из внешнего мира проецируется на *окно просмотра* (англ. viewport). Если заменить это окно на высококачественную фотографию, она будет выглядеть так же до тех пор, пока не изменится точка обзора. В компьютерной 3D-графике экран компьютера действует как окно просмотра.

В зависимости от того, как близко глаз расположен к окну и насколько оно велико, можно видеть лишь ограниченное пространство внешнего мира. Это пространство называется *полем зрения* (англ. field of view). Если провести воображаемые линии от глаза к четырём углам окна и дальше, получится *пирамида видимости* (англ. view frustum). С целью повышения

производительности *усеченная пирамида видимости* обычно ограничивается дальней и ближней плоскостями отсечения. Видно все, что находится внутри пирамиды, но ничего вне ее.

OpenGL ES на Android

Android поддерживает OpenGL ES как через API фреймворка, так и через Native Development Kit (NDK). Android NDK ― это набор инструментов, который позволяет реализовывать части приложения с использованием таких языков, как C и C++. В определенных приложениях это может быть полезно для повторного использования библиотек, написанные на этих языках.

Здесь будет рассматриваться использование OpenGL ES через API фреймворка Android.

В OpenGL ES на Android рисование выделено в класс рендеринга, который отвечает за инициализацию и рисование всего экрана. Для создания и управления графикой используются два основных класса: GLSurfaceView и GLSurfaceView.Renderer.

Объект GLSurfaceView ― это контейнер для просмотра графики, нарисованной с помощью OpenGL, а GLSurfaceView.Renderer управляет тем, что будет нарисовано в этом контейнере.

Класс GLSurfaceView является потомком класса View, то есть является представлением. На нём можно рисовать и управлять объектами с помощью вызовов API OpenGL. GLSurfaceView аналогичен по функциям классу SurfaceView. Использовать класс GLSurfaceView можно создав его экземпляр и присоединив к нему свой класс рендеринга. Однако, если необходимо обрабатывать события сенсорного экрана, следует расширить класс GLSurfaceView для реализации слушателей сенсорных событий.

Использование GLSurfaceView ― это лишь один из способов включения OpenGL ES в приложение. Для полноэкранного просмотра графики этот способ подходит лучше всего. Для включения графики

OpenGL ES в небольшую часть разметки может использоваться класс TextureView.

GLSurfaceView.Renderer — это интерфейс, который определяет методы, необходимые для рисования графики в GLSurfaceView. Необходимо реализовать этот интерфейс как отдельный класс и присоединить его к экземпляру GLSurfaceView с помощью GLSurfaceView.setRenderer ().

Если приложение использует функции OpenGL, которые доступны не на всех устройствах, следует включить требование наличия определённой версии OpenGL в файл AndroidManifest.xml (листинг 1).

Листинг 1. Требование наличия определённой версии OpenGL

```
<!-- Требование наличия OpenGL ES 2.0. -->
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
<!-- Требование наличия OpenGL ES 3.0. -->
<uses-feature android:glEsVersion="0x00030000" android:required="true" />
<!-- Требование наличия OpenGL ES 3.1. -->
<uses-feature android:glEsVersion="0x00030001" android:required="true" />
```

Если приложение использует форматы сжатия текстур, следует объявить поддерживаемые форматы в манифесте приложения (AndroidManifest.xml) с помощью элемента <supports-gl-texture>. Объявление требований к сжатию текстур в манифесте скрывает приложение в Google Play от пользователей с устройствами, которые не поддерживают хотя бы один из объявленных типов сжатия. Форматы сжатия текстур поддерживаются не на всех устройствах. Их поддержка зависит от производителя и устройства. В листинге 2 перечислены различные популярные форматы.

Листинг 2. Требования к сжатию текстур

```
<supports-gl-texture android:name="GL_AMD_compressed_ATC_texture" />
<supports-gl-texture android:name="GL_ATI_texture_compression_atitc" />
<supports-gl-texture android:name="GL_IMG_texture_compression_pvrtc" />
<supports-gl-texture android:name="GL_EXT_texture_compression_s3tc" />
<supports-gl-texture android:name="GL_EXT_texture_compression_dxt1" />
<supports-gl-texture android:name="GL_AMD_compressed_3DC_texture" />
<supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
<supports-gl-texture android:name="GL_OES_compressed_paletted_texture" />
```

Класс GLSurfaceView

Приложения, использующие OpenGL ES, содержат активити, подобно любым другим приложениям с пользовательским интерфейсом. Главное отличие от других приложений в том, что передаётся в метод setContentView (). В то время как в обычных приложениях в данный метод передаётся ссылка на ресурс компоновки, в приложениях с OpenGL ES туда передаётся объект GLSurfaceView.

В листинг 3 показана минимальная реализация активити, которое использует GLSurfaceView.

Листинг 3. Минимальная реализация активити, использующего GLSurfaceView.

```
package com.example.openglapp;
import androidx.appcompat.app.AppCompatActivity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {
    private GLSurfaceView gLView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        gLView = new GLSurfaceView(this); // MyGLSurfaceView(this);
        setContentView(gLView);
    }
}
```

Если в приложении потребуется обработка событий нажатия на экран, то следует создать свой класс, например, MyGLSurfaceView, унаследованный от GLSurfaceView. Если вы используете OpenGL ES 2.0, в классе MyGLSurfaceView необходимо добавить метод setEGLContextClientVersion (2), показывающий, что вы хотите использовать API версии 2.0. В последующих примерах используется OpenGL ES 2.0.

В листинге 4 приведён код класса MyGLSurfaceView.

Листинг 4. Класс MyGLSurfaceView.

```
package com.example.openglapp;
import android.content.Context;
import android.opengl.GLSurfaceView;

class MyGLSurfaceView extends GLSurfaceView {
    private final MyGLRenderer renderer;

    public MyGLSurfaceView(Context context){
        super(context);
        setEGLContextClientVersion(2);
        renderer = new MyGLRenderer();
        setRenderer(renderer);
        setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    }
}
```

При запуске рендерера методом setRenderer стартует отдельный поток, который вызывает у рендерера метод onDrawFrame.

После запуска рендерера нужно определить, как будет производиться отрисовка кадров: автоматически в бесконечном цикле или по команде. Режимом смены кадров управляет метод setRenderMode. Если в данный метод передаётся константа RENDERMODE_CONTINUOUSLY, то метод onDrawFrame вызывается циклически. Этот режим удобен для анимационных роликов. Если же передается константа RENDERMODE_WHEN_DIRTY, метод onDrawFrame вызывается только один раз при старте рендерера и может быть вызван повторно методом requestRender (). Такой режим можно использовать в играх, где требуется вручную управлять сменой кадров, запускать и останавливать анимацию.

Класс GLSurfaceView.Renderer

Как было сказано ранее, GLSurfaceView — это специальный контейнер, в котором можно рисовать графику с использование OpenGL ES. Но одного его недостаточно. Рисованием объектов управляет рендерер — объект GLSurfaceView.Renderer.

Есть три метода, которые вызываются Android, чтобы выяснить, что и как рисовать в GLSurfaceView. Их можно реализовать в потомке класса GLSurfaceView.Renderer:

- Метод onSurfaceCreated () вызывается один раз при создании GLSurfaceView. В нём задаются действия, которые должны выполняться только один раз, например, установка параметров среды OpenGL или инициализация графических объектов OpenGL.
- Метод onDrawFrame () вызывается при каждой перерисовке GLSurfaceView. Именно в этом методе выполняется рисование (и перерисовка) графических объектов.
- Метод onSurfaceChanged () вызывается при изменении геометрии GLSurfaceView, включая изменения размера GLSurfaceView или ориентации экрана устройства, когда, например, устройство меняет ориентацию экрана с книжной на альбомную и наоборот. Этот метод используется, чтобы реагировать на изменения в контейнере GLSurfaceView.

В листинге 5 приведён код класса рендерера, который рисует белый фон в GLSurfaceView.

Листинг 5. Класс MyGLRenderer.

```
package com.example.openglapp;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import android.opengl.GLES20;
import android.opengl.GLSurfaceView;

public class MyGLRenderer implements GLSurfaceView.Renderer {
    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        GLES20.glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    }
    public void onDrawFrame(GL10 unused) {
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    }
    public void onSurfaceChanged(GL10 unused, int width, int height) {
        GLES20.glViewport(0, 0, width, height);
    }
}
```

В onSurfaceCreated () мы вызываем метод glClearColor и передаем ему RGBA-компоненты в диапазоне от 0 до 1. Тем самым мы устанавливаем дефолтный цвет, который будет отображаться после полной очистки поверхности рисования.

В методе onDrawFrame () выполняется очистка. Метод glClear с параметром GL_COLOR_BUFFER_BIT очистит все цвета на экране, и установит цвет, заданный методом glClearColor.

В методе onSurfaceChanged методом glViewPort задаётся область, которая будет доступна для вывода изображения. Указывается левая нижняя точка ─ (0,0) и размеры области (width, height). В данном примере изображение будет выведено на всей поверхности рисования.

Внутри метода onSurfaceCreated () можно задать параметры OpenGL. OpenGL имеет десятки параметров, которые могут быть включены или выключены с помощью методов glEnable () и glDisable (). Наиболее часто используемых приведены в таблице 1.

Таблица 1. Примеры параметров OpenGL.

Параметр	**Описание**
GL_BLEND	Смешивает входящие цветовые значения с цветами, которые уже находятся в буфере цветов.
GL_CULL_FACE	Устанавливает режим отсечения граней фигуры (задней и/или лицевой).
GL_DEPTH_TEST	Производит сравнение глубин и обновляет буфер глубины. Пиксели, которые расположены дальше, чем уже нарисованные, игнорируются.
GL_DITHER	Включает дизеринг.

Все параметры по умолчанию выключены, за исключением GL_DITHER. Следует учитывать, что все включенные параметры требует некоторых затрат вычислительной мощности.

Определение фигур в OpenGL ES

OpenGL ES позволяет определять фигуры по тремя или более точками в трехмерном пространстве. Например, три такие точки задают

треугольник. Треугольники — это одни из основных примитивов, из которых строятся другие фигуры (четырёхугольники, параллелепипеды и т.д.) Точки в трехмерном пространстве на OpenGL ES называются вершинами или вертексами (англ. vertices).

Для отрисовки треугольника, нужно определить его координаты. Это можно сделать, определив массив вершин для координат — чисел с плавающей точкой. Для максимальной эффективности эти координаты записываются в ByteBuffer, который передается в графический конвейер OpenGL ES для обработки.

По умолчанию OpenGL предполагает, что начало координат (X;Y;Z) = (0;0;0) находится в центре контейнера GLSurfaceView, (1;1;0) — это верхний правый угол контейнера, а (-1;-1;0) — нижний левый угол.

Порядок, в котором указываются координаты вершин фигуры принципиален, потому что он определяет, какая сторона фигуры является передней гранью, которая обычно отрисовывается, а какая сторона — задней гранью, которую можно не рисовать, используя отсечение граней (англ. face culling).

Отсечение граней включается внутри метода обратного вызова onSurfaceCreated () методом glEnable(GL_CULL_FACE), а отсечение задней или лицевой граней устанавливается с помощью метода glCullFace(GL_FRONT | GL_BACK | GL_FRONT_AND_BACK) (листинг 6).

Листинг 6. Отсечение граней.

```
GLES20.glEnable(GLES20.GL_CULL_FACE);
GLES20.glCullFace(GLES20.GL_BACK);
```

Координаты передней (лицевой) грани фигуры задаются против часовой стрелки. На рисунке 1 приведена лицевая грань треугольника; порядок, в котором следует задавать его вершины указан стрелками.

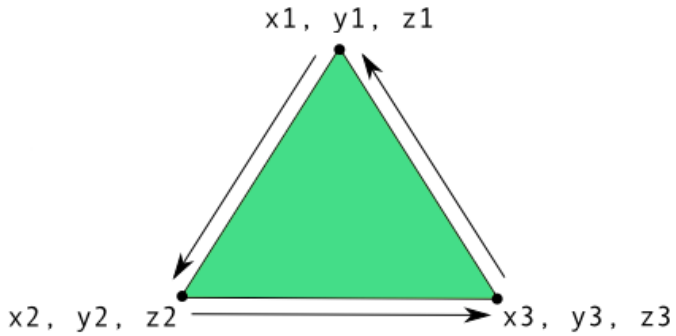

Рисунок 1. Порядок задания координат вершин треугольника.

В листинге 7 приведён код класса, в котором задаются координаты вершин для построения треугольника.

Листинг 7. Класс MyTriangle.

```
package com.example.openglapp;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

public class MyTriangle {
    private FloatBuffer vertexBuffer;

    static final int COORDS_PER_VERTEX = 3;
    static float triangleCoords[] = {
            0.0f,  0.5f, 0.0f, // верхняя вершина
            -0.5f, -0.25f, 0.0f, // нижняя левая вершина
            0.5f, -0.25f, 0.0f  // нижняя правая вершина
    };
    float color[] = { 0.0f, 0.5f, 0.5f, 1.0f };

    public MyTriangle() {
        //инициализируем буфер байтов вершин для координат фигуры; 4
байта на одно значение float
        ByteBuffer bb = ByteBuffer.allocateDirect(triangleCoords.length *
4);
        bb.order(ByteOrder.nativeOrder());
        // создаем буфер вершин из ByteBuffer
        vertexBuffer = bb.asFloatBuffer();
        vertexBuffer.put(triangleCoords);
        vertexBuffer.position(0);
    }
}
```

Для рисования других более сложных фигур в OpenGL в основном используется множество треугольников. Например, квадрат состоит из 2 треугольников (рис. 2).

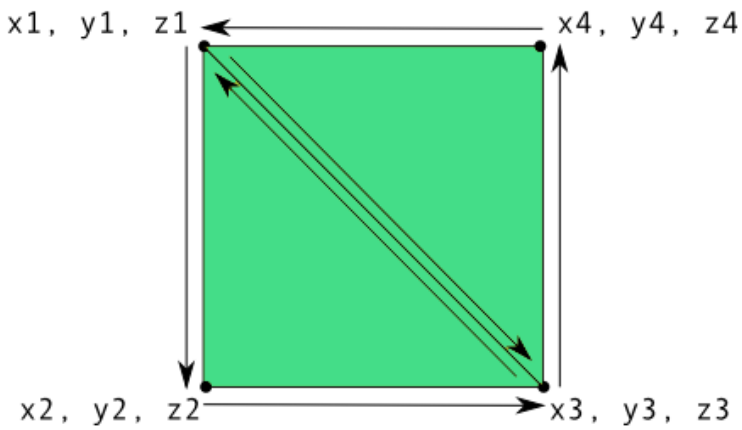

Рисунок 2. Порядок задания координат вершин прямоугольника.

В листинге 8 приведён код класса, в котором задаются координаты вершин для построения прямоугольника.

Листинг 8. Класс MySquare.

```
package com.example.openglapp;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.nio.ShortBuffer;

public class MySquare {
    private FloatBuffer vertexBuffer;
    private ShortBuffer drawListBuffer;

    static final int COORDS_PER_VERTEX = 3;
    static float squareCoords[] = {
            -0.5f,  0.5f, 0.0f,   // верхняя левая вершина
            -0.5f, -0.5f, 0.0f,   // нижняя левая вершина
            0.5f, -0.5f, 0.0f,   // нижняя правая вершина
            0.5f,  0.5f, 0.0f }; // верхняя правая вершина

    private short drawOrder[] = { 0, 1, 2, 0, 2, 3 }; //порядок рисования

    public Square() {
        ByteBuffer bb = ByteBuffer.allocateDirect(squareCoords.length *
4);
        bb.order(ByteOrder.nativeOrder());
        vertexBuffer = bb.asFloatBuffer();
        vertexBuffer.put(squareCoords);
        vertexBuffer.position(0);

        ByteBuffer dlb = ByteBuffer.allocateDirect(drawOrder.length * 2);
        dlb.order(ByteOrder.nativeOrder());
        drawListBuffer = dlb.asShortBuffer();
        drawListBuffer.put(drawOrder);
        drawListBuffer.position(0);
    }
}
```

Чтобы избежать двойного описания двух общих для треугольников координат, используется список порядка рисования, сообщающий OpenGL, как рисовать эти вершины.

Рисование фигур в OpenGL ES

Рисование фигур с помощью OpenGL ES 2.0 требует достаточно много кода (по сравнению с OpenGL ES 1.0), потому что обеспечивается больший контроль над конвейером рендеринга графики.

Прежде чем рисовать фигуры, необходимо их инициализировать и загрузить. Если координаты вершин фигур не изменятся в ходе выполнения программы, фигуры следует инициализировать в методе onSurfaceCreated () (листинг 9).

Листинг 9. Инициализация фигур.

```
public class MyGLRenderer implements GLSurfaceView.Renderer {
    private MyTriangle mTriangle;
    private MySquare   mSquare;

    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        ...
        mTriangle = new MyTriangle();
        mSquare = new MySquare();
    }
...
```

Для рисования определенной фигуры с использованием OpenGL ES 2.0 требуется предоставить множество деталей конвейеру рендеринга графики. Следует определить:

- Вершинный шейдер (англ. Vertex Shader) — графический код OpenGL ES для визуализации вершин фигуры.
- Фрагментный шейдер (англ. Fragment Shader) — код OpenGL ES для рендеринга лицевой грани фигуры с использованием цветов или текстур.
- Программа — объект OpenGL ES, содержащий шейдеры, которые будут использоваться для рисования одной или нескольких фигур.

Шейдеры — программы, предназначенные для выполнения на процессорах видеокарты. Для написания шейдеров в OpenGL используется язык программирования OpenGL Shading Language (GLSL).

Понадобится как минимум один вершинный шейдер, чтобы нарисовать фигуру, и один фрагментный шейдер, чтобы раскрасить эту фигуру. Эти шейдеры должны быть скомпилированы и добавлены в программу OpenGL ES, которая затем используется для рисования фигуры. В листинге 10 приведён пример того, как определить базовые шейдеры, которые можно использовать для рисования фигуры в классе MyTriangle.

Листинг 10. Шейдеры.

```
private final String vertexShaderCode =
        "attribute vec4 vPosition;" +
                "void main() {" +
                "  gl_Position = vPosition;" +
                "}";

private final String fragmentShaderCode =
        "precision mediump float;" +
                "uniform vec4 vColor;" +
                "void main() {" +
                "  gl_FragColor = vColor;" +
                "}";
```

Шейдеры содержат код GLSL, который необходимо скомпилировать перед использованием в среде OpenGL ES. Чтобы скомпилировать этот код, создаётся служебный метод в классе рендерера (листинг 11).

Листинг 11. Метод для компиляции шейдера.

```
public static int loadShader(int type, String shaderCode){
    int shader = GLES20.glCreateShader(type);
    GLES20.glShaderSource(shader, shaderCode);
    GLES20.glCompileShader(shader);
    return shader;
}
```

Чтобы нарисовать фигуру, нужно скомпилировать код шейдера, добавить его в программный объект OpenGL ES, а затем связать программу. Это следует делать в конструкторе фигуры, так чтобы это выполнялось только один раз (листинг 12).

Компиляция шейдеров OpenGL ES и связывание программ обходятся дорого с точки зрения циклов ЦП и времени обработки, поэтому не следует выполнять их более одного раза. Если вы не знаете содержимое своих шейдеров во время выполнения, следует построить код так, чтобы они создавались только один раз, а затем кэшировались для дальнейшего использования.

Листинг 12. Компиляция шейдеров и связывание программы.

```
public class MyTriangle {
    private final int mProgram;
    ...
public MyTriangle() {
    ...
    int vertexShader =
MyGLRenderer.loadShader(GLES20.GL_VERTEX_SHADER, vertexShaderCode);
    int fragmentShader =
MyGLRenderer.loadShader(GLES20.GL_FRAGMENT_SHADER, fragmentShaderCode);
    // создание пустой OpenGL ES программы
    mProgram = GLES20.glCreateProgram();
    // Добавление вершинного шейдера в программу
    GLES20.glAttachShader(mProgram, vertexShader);
    // Добавление фрагментного шейдера в программу
    GLES20.glAttachShader(mProgram, fragmentShader);
    // Связывание программы
    GLES20.glLinkProgram(mProgram);
}
}
```

Теперь можно добавить методы, которые будут рисовать фигуру. Рисование фигур с помощью OpenGL ES требует, чтобы были указаны несколько параметров, сообщающих конвейеру рендеринга, что и как нужно рисовать. Поскольку параметры рисования могут различаться в зависимости от фигуры, рекомендуется, чтобы классы фигур содержали собственную логику рисования — свой метод draw ().

В листинге 13 приведён пример метод draw () для рисования треугольника. Этот код устанавливает значения положения и цвета для вершинного шейдера фигуры и шейдера фрагмента, а затем выполняет функцию рисования.

Листинг 13. Метод draw ().

```
private int positionHandle;
private int colorHandle;

private final int vertexCount = triangleCoords.length /
COORDS_PER_VERTEX;
private final int vertexStride = COORDS_PER_VERTEX * 4;
// 4 байта на вершину

public void draw() {
    // Добавляем программу в среду OpenGL ES
    GLES20.glUseProgram(mProgram);
    // Получаем элемент vPosition вершинного шейдера
    positionHandle = GLES20.glGetAttribLocation(mProgram, "vPosition");
    // Подключаем массив вершин
    GLES20.glEnableVertexAttribArray(positionHandle);
    // Подготовка координат вершин треугольника
    GLES20.glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
            GLES20.GL_FLOAT, false,
            vertexStride, vertexBuffer);
    // Получаем элемент vColor фрагментного шейдера
    colorHandle = GLES20.glGetUniformLocation(mProgram, "vColor");
    // Устанавливаем цвет для рисования треугольника
    GLES20.glUniform4fv(colorHandle, 1, color, 0);
    // Рисуем треугольник
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);
    // Отключаем массив вершин
    GLES20.glDisableVertexAttribArray(positionHandle);
}
```

Для построения прямоугольника код аналогичный, только в методе draw () класса MySquare вместо метода GLES20.glDrawArrays (GLES20.GL_TRIANGLES, 0, vertexCount) вызывается метод GLES20.glDrawElements (GLES20.GL_TRIANGLES, drawOrder.length, GLES20.GL_UNSIGNED_SHORT, drawListBuffer).

Теперь для рисования фигуры требуется вызвать метода draw () из метода onDrawFrame () класса рендерера (листинг 14).

Листинг 14. Вызов метода draw ().

```
public void onDrawFrame(GL10 unused) {
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    mTriangle.draw();
}
```

На рисунке 3 представлен результат работы приложения, строящего треугольник.

Обратите внимание, что треугольник меняет форму при изменении ориентации экрана устройства. Это происходит из-за того, что вершины объекта не были скорректированы с учетом пропорций области экрана, в которой отображается GLSurfaceView. Эту проблему можно решить с помощью проекции и вида камеры.

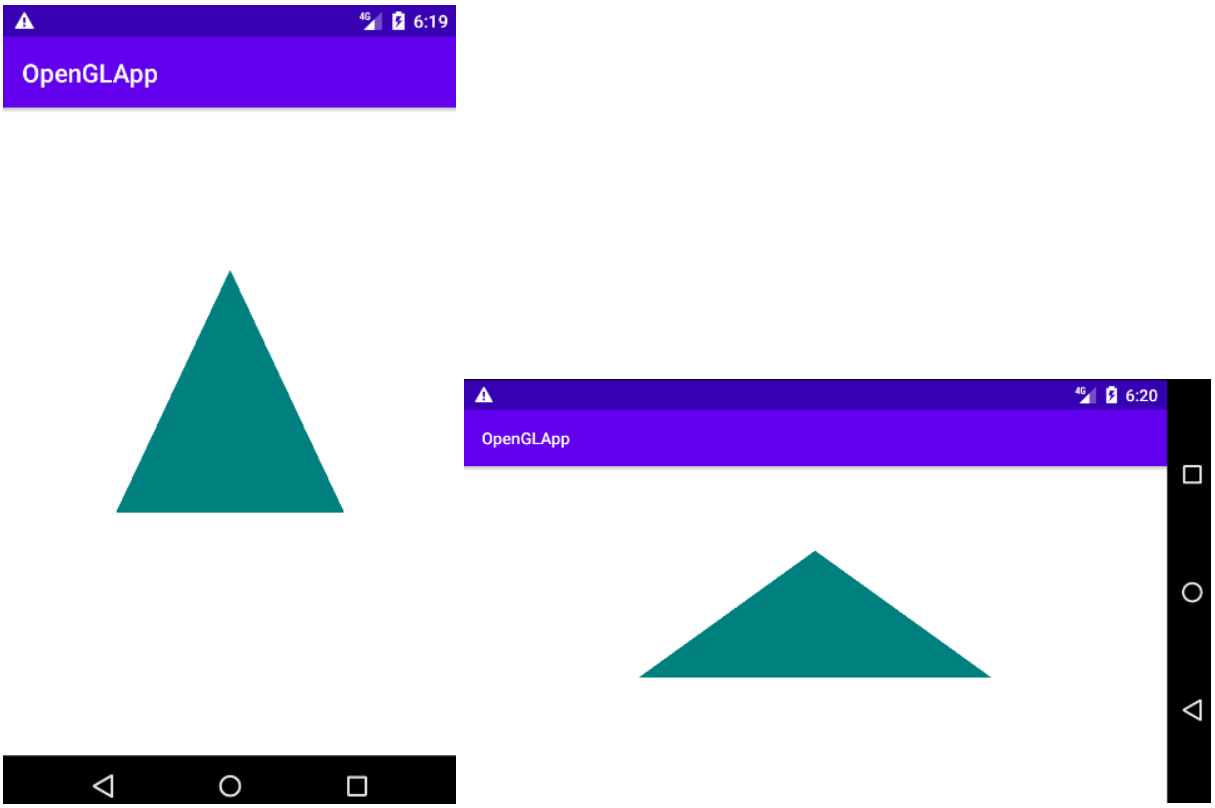

Рисунок 3. Результат работы приложения.

Применение проекции и вида камеры

Одна из основных проблем при отображении графики на устройствах Android заключается в том, что их экраны могут различаться по размеру и форме. OpenGL предполагает квадратную, однородную систему координат.

На рисунке 4 слева показана система координат OpenGL, а на рисунке справа, как это будет выглядеть на мобильном устройстве с альбомной ориентацией.

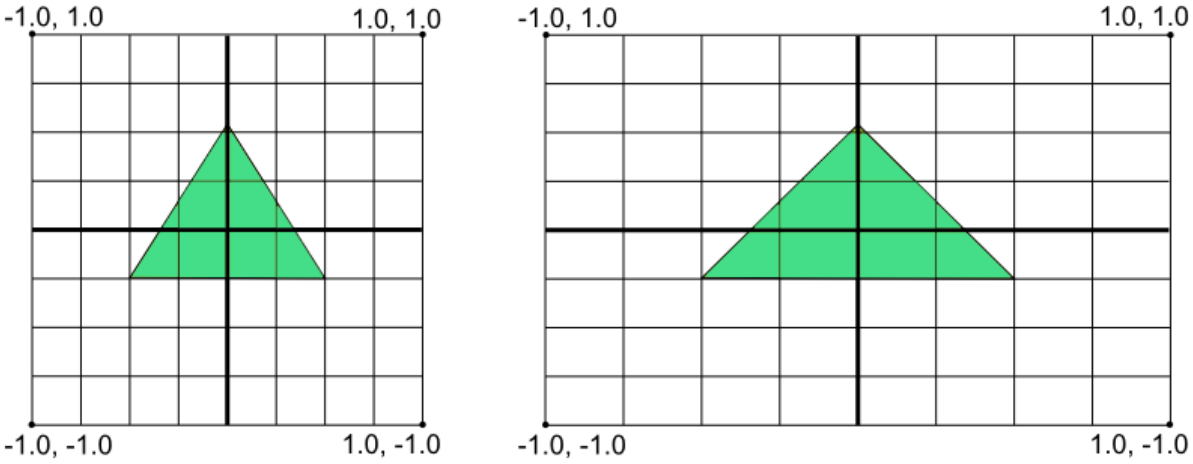

Рисунок 4. Система координат OpenGL.

Чтобы исправить эту ситуацию применяются проекции и виды камеры. При этом создаются матрица проекции и матрица вида камеры и применяются к конвейеру рендеринга OpenGL. С их помощью выполняется математическое преобразование координат нарисованного объекта.

Проекция — это преобразование, которое регулирует координаты нарисованных объектов в зависимости от ширины и высоты GLSurfaceView, в котором они отображаются. Преобразование проекции обычно рассчитывается тогда, когда пропорции представления установлены или изменены в методе onSurfaceChanged ().

Вид камеры — это преобразование регулирует координаты нарисованных объектов в зависимости от положения виртуальной камеры. OpenGL ES предоставляет служебные методы, которые имитируют камеру, изменяя отображение нарисованных объектов. Преобразование вида камеры может быть рассчитано только один раз при установке GLSurfaceView или может изменяться динамически в зависимости от действий пользователя или функций вашего приложения.

В листинге 15 приведён код, который принимает высоту и ширину GLSurfaceView и использует их для заполнения матрицы проекции с помощью метода Matrix.frustumM (), который определяет матрицу проекции в виде шести плоскостей отсечения.

Листинг 15. Преобразование проекции.

```
private final float[] vPMatrix = new float[16];
private final float[] projectionMatrix = new float[16];
private final float[] viewMatrix = new float[16];

public void onSurfaceChanged(GL10 unused, int width, int height) {
    GLES20.glViewport(0, 0, width, height);
    float ratio = (float) width / height;
// Матрица проекции применяется к координатам фигуры в onDrawFrame()
    Matrix.frustumM(projectionMatrix, 0, -ratio, ratio, -1, 1, 3, 7);
}
```

Этот код заполняет матрицу проекции projectionMatrix, которую затем следует комбинировать с преобразованием вида камеры в методе onDrawFrame ().

Для завершения процесса преобразования координат добавим преобразование вида камеры как часть процесса рисования в рендерере. В листинге 16 приведён код преобразования вида камеры с помощью метода Matrix.setLookAtM (), которое затем комбинируется с ранее рассчитанной матрицей проекции. Затем объединенные матрицы преобразования передаются в нарисованную фигуру.

Листинг 16. Преобразование вида.

```
public void onDrawFrame(GL10 unused) {
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    // Задаём позицию камеры (матрицу вида)
    Matrix.setLookAtM(viewMatrix, 0, 0, 0, 5, 0f, 0f, 0f, 0f, 1.0f,
0.0f);
    // Вычисляем преобразования с помощью матриц проекции и вида
    Matrix.multiplyMM(vPMatrix, 0, projectionMatrix, 0, viewMatrix, 0);
    // Рисуем фигуру
    mTriangle.draw(vPMatrix);
}
```

Чтобы использовать комбинированную матрицу преобразования проекции и вида камеры, следует добавить матричную переменную в вершинный шейдер, ранее определенный в классе Triangle. А затем модифицировать метод draw () так, чтобы он принимал комбинированную матрицу преобразования и применил ее к форме (листинг 17).

Листинг 17. Модификация класса Triangle.

```
private final String vertexShaderCode =
        "uniform mat4 uMVPMatrix;" +
         "attribute vec4 vPosition;" +
         "void main() {" +
                "  gl_Position = uMVPMatrix * vPosition;" +
                "}";

private int vPMatrixHandle;

...
public void draw(float[] mvpMatrix) {
    // Добавляем программу в среду OpenGL ES
    GLES20.glUseProgram(mProgram);
    ...

    vPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
    GLES20.glUniformMatrix4fv(vPMatrixHandle, 1, false, mvpMatrix, 0);

    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);
    GLES20.glDisableVertexAttribArray(positionHandle);
}
```

Теперь графические объекты в приложении будут нарисованы в правильных пропорциях и должны выглядеть следующим образом (рис. 5).

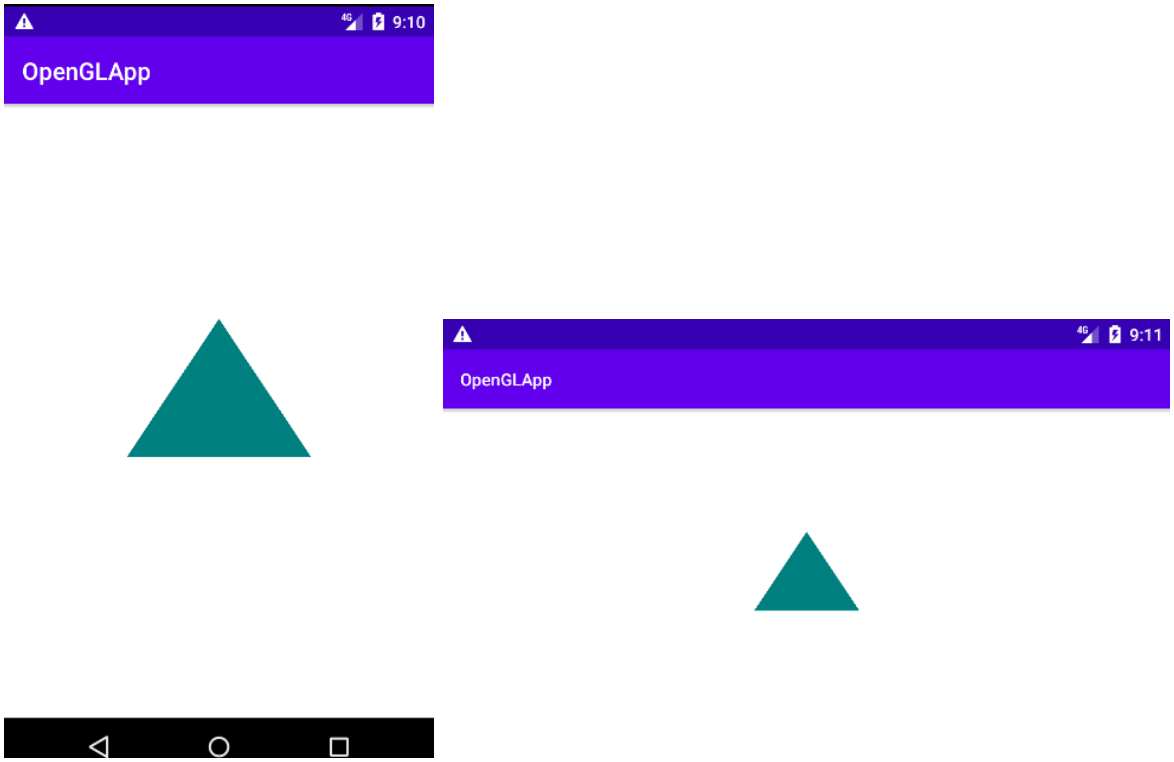

Рисунок 5. Треугольник, нарисованный с применением проекции и вида камеры.

Анимация движения объектов

OpenGL ES предоставляет возможности для перемещения и преобразования нарисованных объектов в трех измерениях. Рассмотрим пример вращения фигуры.

Для поворота фигуры необходимо создать в классе рендерера матрицу преобразования (матрицу вращения), а затем объединить её с матрицами преобразования проекции и вида камеры (листинг 18).

Листинг 18. Модификация метода onDrawFrame.

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

@Override
public void onDrawFrame(GL10 unused) {
   GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
   float[] scratch = new float[16];
   Matrix.setLookAtM(viewMatrix, 0, 0, 0, 5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
   Matrix.multiplyMM(vPMatrix, 0, projectionMatrix, 0, viewMatrix, 0);

   long time = SystemClock.uptimeMillis() % 4000L;
   float angle = 0.09f * ((int) time);
   Matrix.setRotateM(rotationMatrix, 0, angle, 0, 0, -1.0f);

   Matrix.multiplyMM(scratch, 0, vPMatrix, 0, rotationMatrix, 0);
   mTriangle.draw(scratch);
}
```

Также необходимо проверить какой режим смены кадров установлен в классе MyGLSurfaceView. Если в метод setRenderMode передаётся константа RENDERMODE_WHEN_DIRTY, то нужно поменять её на RENDERMODE_CONTINUOUSLY или удалить этот метод.

Задание

Разработайте с использование OpenGL ES 2.0 (или выше) приложение, в котором отображается вращающийся куб.

Контрольные вопросы:

1. Что такое OpenGL ES?
2. Какие версии OpenGL ES поддерживает платформа Android?
3. Для чего используется класс GLSurfaceView?
4. Для чего используется класс GLSurfaceView.Renderer?

5. Когда происходит вызов метода onSurfaceCreated?
6. Когда происходит вызов метода onSurfaceChanged?
7. Когда происходит вызов метода onDrawFrame?
8. Для чего используется метод setRenderer?
9. Для чего используется метод setRenderMode?
10. Для чего используется метод requestRender?
11. Для чего используется метод glViewPort?
12. Как включить отсечение задних граней?
13. Что такое вершинный шейдер?
14. Что такое фрагментный шейдер?
15. Для чего применяются проекция и вид камеры?