Курсовой проект

«Разработка интерактивной 3D-сцены с использованием OpenGL ES 2.0»

Цель: приобрести практические навыки построения 3D-объектов, применения материалов и текстур, настройки освещения и камеры, а также научиться использовать нативный код на C++, библиотеки Assimp и GLM.
Задание: Разработать мобильное приложение для Android, отображающее трёхмерную сцену в соответствии с индивидуальным вариантом задания. Использовать OpenGL ES 2.0 для рендеринга моделей и шейдеров. Необходимо использовать текстуры для реалистичного отображения объектов, модели в формате .obj для представления 3D-объектов и библиотеку Assimp для их загрузки и обработки. Камера должна вращаться вокруг сцены. Файлы моделей можно скачать с любого бесплатного источника, например, https://open3dmodel.com/3d-models/obj/page/3.
Номер варианта соответствует последней цифре Вашего пароля.
Среда разработки: Android Studio.
Язык: Java, C++.

Варианты задания:

Задание 1: "Холодильник".

Реализовать 3D-сцену, изображающую внутреннее пространство холодильника. На сцене должны быть:

- Статическое изображение холодильной камеры.
- Апельсин (сфера с текстурой).
- Пакет молока (прямоугольная призма с текстурой).
- Сосиски (цилиндр или овал).
- Бутылка (конус + цилиндр).
- Освещение внутри холодильника с легким мерцанием лампочки.
- Дверца как статическая модель.

Задание 2: "Железнодорожный вокзал"

Создать 3D-модель железнодорожного вокзала с поездом и окружением. На сцене должны быть:

- Статическое здание вокзала с информационным табло и скамейками.
- Поезд из 3 вагонов (платформа, цистерна, крытый вагон) — каждый вагон как отдельная модель.
- Простая анимация движения поезда (перемещение вдоль платформы).
- Текстуры для здания и вагонов.
- Равномерное освещение и мигание часов.

Задание 3: "Супермаркет"

Реализовать 3D-сцену торгового зала с полками и товарами. На сцене должны быть:

- Несколько стеллажей с разными товарами.
- Коробки (кубы), банки (цилиндры), бутылки (конус+цилиндр), консервные банки (цилиндры).
- Применение текстур для имитации этикеток и упаковок.
- Верхнее освещение с мерцанием вывески над кассами.

Задание 4: "Городской ландшафт с фонтаном"
Создать 3D-сцену городской площади с фонтаном и другими объектами. На сцене должны быть:

- Фонтан с анимацией с помощью простого движения частиц.
- Клумбы (плоские объекты с текстурой цветов).
- Декоративные ели (текстурированные пирамиды).
- Здание библиотеки и остановка общественного транспорта.
- Направленный дневной свет.

Задание 5: "Аэропорт"
Создать 3D-сцену аэропорта с терминалом и взлетно-посадочной полосой. На сцене должны быть:

- Терминал с информационным табло (без обновления).
- Самолёт на взлётной полосе.
- Башня управления и вращающиеся световые маячки на ней.
- Простые фигуры людей (цилиндры или кубы).
- Ночное освещение.

Задание 6: "Магазин бытовой техники"
Реализовать 3D-сцену магазина с электроприборами. На сцене должны быть:

- Полки с разной техникой.
- Телевизоры (прямоугольники с текстурой экрана), микроволновки (прямоугольники с текстурой), утюги, холодильники (кубы с текстурами).
- Вращение вентилятора на полке.
- Применение текстур для имитации металла и пластика.
- Освещение с подсветкой от техники.

Задание 7: "Автомобильная парковка"
Создать 3D-сцену многоуровневого паркинга с автомобилями. На сцене должны быть:

- Парковочные места и машины разных форм: кубы, цилиндры, эллипсоиды.
- Лестницы и указатели направления (без сложных моделей).
- Перемещение одной машины на свободное место (движение по прямой).
- Верхнее освещение.

Задание 8: "Игровая площадка"

Реализовать 3D-сцену детской игровой площадки. На сцене должны быть:

- Карусель, качели, горки (простые модели).
- Вращение карусели и колебание качелей.
- Минималистичные фигурки детей (цилиндры с головой).
- Деревья, песочница, скамейки (текстурированные модели).
- Мягкий дневной свет.

Задание 9: "Лаборатория научного оборудования"

Реализовать 3D-сцену интерьера научной лаборатории. На сцене должны быть:

- Столы с различным оборудованием.
- Микроскопы, колбы, пробирки (цилиндры, конусы, сферы с текстурами).
- Изменение цвета жидкости в колбе.
- Компьютеры (прямоугольники с текстурой экрана).
- Простое освещение.

Задание 0: "Кафе в стиле ретро"

Создать 3D-сцену интерьера кафе в ретро-стиле. На сцене должны быть:

- Стулья, столы, барная стойка (текстурированные модели).
- Посуда: чашки, тарелки, графины (цилиндры, конусы, кубы с текстурами).
- Элементы декора: виниловые пластинки, старинные часы, неоновая вывеска.
- Вращение проигрывателя.
- Мягкий свет от люстры.

Методические указания к выполнению курсового проекта

Прежде чем начать выполнение курсового проекта, необходимо получить положительную оценку за лабораторные работы, а также изучить лекцию №8.

Затем реализовать задание по варианту, следуя рекомендациям по шагам:

- Шаг 1: Создайте проект в Android Studio.

В процессе создания нового проекта убедитесь, что вы выбрали шаблон Empty Activity и включили поддержку C++ при помощи флажка Include C++ support. Это обеспечит автоматическую настройку NDK, CMake и компилятора C++. Также необходимо скачать и установить последнюю версию Android NDK через SDK Manager внутри Android Studio и установить CMake (доступен через SDK Tools в Android Studio).

- Шаг 2: Настройте структуру проекта.

Создайте структуру проекта, включающую Java-часть (для взаимодействия с Android API) и нативную часть на C++ (для работы с OpenGL). В Java-коде понадобятся классы MainActivity, MySurface, MyRenderer и LoadLibJNIWrapper.

В папке cpp будут находиться все нативные файлы: native-lib.cpp, Model.cpp/h, Mesh.cpp/h, Shader.cpp/h, model_loading.cpp/h, а также сторонние библиотеки Assimp и GLM. Правильно настройте CMakeLists.txt, чтобы он включал все необходимые модули и пути к библиотекам. В нём нужно указать все исходные файлы, пути к заголовочным файлам сторонних библиотек, а также зависимости, необходимые для работы OpenGL ES 2.0.

- Шаг 3: Подключите библиотеку GLM.

Библиотека GLM используется для работы с матрицами, векторами и преобразованиями в 3D-пространстве. Скачайте GLM с официального сайта (https://github.com/g-truc/glm/tags) и разархивируйте ее в каталог «dependencies», добавьте путь к заголовочным файлам в CMakeLists.txt. Эта библиотека не требует компиляции — она полностью состоит из заголовочных файлов и готова к использованию сразу после подключения.

- Шаг 4: Подключите библиотеку Assimp.

Assimp — это мощная библиотека для загрузки 3D-моделей различных форматов, в том числе .obj. Чтобы её использовать, Вам нужно скачать исходный код Assimp (https://github.com/assimp/assimp/releases), собрать его под Android с помощью CMake и скопировать в папку app/src/main/cpp/Assimp. Скопируйте папку assimp в корень вашего Android-проекта, создайте папку build для сборки (<YourProjectRoot>/app/src/main/cpp/assimp/build), чтобы CMake мог туда поместить результаты сборки и запустите CMake с указанием Android NDK и целевой архитектуры. Пример для ABI armeabi-v7a (ARMv7):

```
cmake -
DCMAKE_TOOLCHAIN_FILE=$ANDROID_NDK/build/cmake/android.toolchain.c
make \
-DANDROID_ABI="armeabi-v7a" \
-DANDROID_PLATFORM=android-21 \
-DBUILD_SHARED_LIBS=OFF \
-DASSIMP_BUILD_ASSIMP_TOOLS=OFF \
-DASSIMP_BUILD_TESTS=OFF \
-DASSIMP_BUILD_ZLIB=ON \
..
```

Здесь $ANDROID_NDK – это полный путь к вашему Android NDK (например, /Users/yourname/Library/Android/sdk/ndk/25.2.9539766).

Если вы хотите собрать под другую архитектуру, замените armeabi-v7a на: arm64-v8a, x86_64 или x86 (для эмуляторов).

После успешной конфигурации запустите сборку (для этого используется команда make в командной строке), и в результате будет создана статическая библиотека libassimp.a (после завершения сборки она должна появиться в одной из подпапок <YourProjectRoot>/assimp/build/code/libassimp.a), которую нужно скопировать в проект (<YourProjectRoot>/app/src/main/cpp/libs/<ABI>/libassimp.a, где <ABI> — это armeabi-v7a, arm64-v8a, x86_64 и т. д.). Заголовочные файлы библиотеки переместите также в свой проект (<YourProjectRoot>/app/src/main/cpp/include/assimp). Вы можете собрать Assimp для нескольких ABI и положить соответствующие .a файлы в нужные папки в

проекте. Добавьте путь к заголовкам и объектной библиотеке в CMakeLists.txt, чтобы ваш проект мог использовать функционал Assimp для парсинга 3D-моделей:

```
# Добавляем подкаталог Assimp
add_subdirectory(${CMAKE_SOURCE_DIR}/src/main/cpp/assimp)
# Указываем, где находятся заголовочные файлы
include_directories(${CMAKE_SOURCE_DIR}/src/main/cpp/assimp/include)
# Добавляем Assimp в список линковки
target_link_libraries(
    native-lib # имя вашей нативной библиотеки
    assimp     # добавляем Assimp
    ${log-lib}
    ${GL20}
    glm
)
```

Теперь вы можете использовать Assimp в вашем C++ коде:

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
```

Например, чтобы загрузить модель:

```
Assimp::Importer importer;
const aiScene* scene = importer.ReadFile("model.obj", aiProcess_Triangulate |
aiProcess_FlipUVs);
if (!scene) {
    // Обработка ошибки
}
```

Соберите проект и запустите его на эмуляторе или реальном устройстве. Если ошибок компиляции нет, Assimp успешно интегрирован в проект и готов к использованию.

- Шаг 5: Создайте Java-классы для OpenGL ES.

Создайте основные Java-классы: MainActivity, MySurface, MyRenderer и LoadLibJNIWrapper. MainActivity создаёт экземпляр MySurface, который использует MyRenderer для рендеринга. MyRenderer отвечает за инициализацию OpenGL, загрузку текстур и передачу путей к 3D-файлам в C++. LoadLibJNIWrapper содержит нативные методы, вызываемые из Java и связанные с C++.

- Шаг 6: Реализуйте шейдеры и программу шейдеров.

Создайте вершинный и фрагментный шейдеры на языке GLSL. Вершинный шейдер будет обрабатывать положение вершин и передавать текстурные координаты, а фрагментный — применять текстуру к поверхности модели. Объедините эти шейдеры в одну программу, которую вы будете использовать в OpenGL.

- Шаг 7: Загрузите текстуры для моделей.

Подготовьте текстуры в формате .png для каждого объекта сцены и поместите их в папку res/drawable. В методе onSurfaceChanged класса MyRenderer загружайте

текстуры с помощью GLUtils.texImage2D() и применяйте к ним стандартные параметры: фильтрация, повторение, генерация mipmaps. Текстуры привязываются к текстурным блокам, а затем передаются в шейдеры для использования при отрисовке моделей.

- Шаг 8: Загрузите 3D-модели в формате .obj.

Поместите ваши 3D-модели в формате .obj вместе с соответствующими .mtl файлами и текстурами в папку assets проекта. Файлы .mtl — это файлы материалов, которые работают вместе с .obj моделями, чтобы задать их внешний вид: цвета, текстуры, отражения и прочие поверхностные свойства. Через JNI передавайте пути к этим файлам в нативный код. Используйте библиотеку Assimp для парсинга файлов и извлечения данных о вершинах, нормалях и текстурных координатах. Сохраните каждую модель как набор мешей (Mesh) внутри класса Model. Меш – это минимальная единица модели, которая содержит вершины с координатами, текстурными данными и нормалями, индексы для построения полигонов, а также материал, определяющий её внешний вид.

- Шаг 9: Вычислите матрицы преобразования.

Используя библиотеку GLM, вычислите три основные матрицы: модель, вид и проекцию. Матрица модели определяет положение и ориентацию объекта, видовая матрица задаёт точку зрения камеры, а проекционная — перспективу. Перемножьте их, чтобы получить матрицу MVP (Model-View-Projection), которая передаётся в шейдеры. Каждый объект должен иметь свою матрицу модели, чтобы правильно размещаться в пространстве.

- Шаг 10: Организуйте рендеринг кадра.

В методе onDrawFrame класса MyRenderer очистите экран и глубинный буфер, активируйте текстурные блоки и передайте текущую матрицу MVP в шейдер. Для каждой модели вызывайте соответствующий нативный метод on_draw_frame, где будет происходить рисование с использованием VBO и EBO. Реализуйте анимацию камеры вокруг сцены, изменяя угол поворота камеры на каждом кадре и обновляя видовую матрицу. Таким образом, пользователь сможет рассмотреть сцену со всех сторон.

Далее рассмотрим пример реализации 3D-сцены, изображающей стол, на котором лежат различные фрукты и овощи, а также стоит стакан с напитком.

1. Точкой входа в приложение является файл MainActivity.java. Он создаёт главное окно и устанавливает OpenGL-поверхность для отрисовки 3D-сцены. Здесь загружается нативная библиотека opengl_kurs_modeld, которая содержит всю логику рендеринга на C++. В методе onCreate создаётся экземпляр MySurface, который является настраиваемой OpenGL-поверхностью, и устанавливается как контентное представление активности. Этот класс управляет жизненным циклом приложения и запускает OpenGL ES рендеринг.

```
package com.example.opengl_kurs_model;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
```

```
public class MainActivity extends AppCompatActivity {
    static {
        System.loadLibrary("opengl_kurs_modeld"); // Загрузка нативной библиотеки
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        MySurface surface = new MySurface(this); // Создание поверхности OpenGL
        setContentView(surface); // Установка как основного интерфейса
    }
}
```

2. Файл MySurface.java настраивает OpenGL ES 2.0 в Android и связывает Java-часть с нативным кодом через JNI. Здесь указывается версия OpenGL (2.0), создаётся пользовательский рендерер MyRenderer, а также устанавливается режим постоянного рендеринга (RENDERMODE_CONTINUOUSLY). Это позволяет приложению обновлять сцену на каждом кадре, создавая плавную анимацию камеры вокруг объектов.

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

public class MySurface extends GLSurfaceView {
    private MyRenderer renderer;

    public MySurface(Context c) {
        super(c);
        setEGLContextClientVersion(2); // Установка версии OpenGL ES
        renderer = new MyRenderer(c);
        setRenderer(renderer); // Привязка рендерера
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY); // Рендеринг на каждом
кадре
    }
}
```

3. Файл MyRenderer.java содержит класс MyRenderer, который реализует интерфейс GLSurfaceView.Renderer и отвечает за подготовку текстур и вызов нативных функций. Метод onSurfaceCreated вызывает нативный код для создания шейдера. В onSurfaceChanged загружаются текстуры из ресурсов (R.drawable.*) и копируются модели .obj из assets во временные файлы, чтобы их можно было передать в C++. Метод onDrawFrame очищает экран и вызывает отрисовку всех моделей, применяя соответствующие текстуры.

```
package com.example.opengl_kurs_model;
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.opengl.GLES20;
import android.opengl.GLUtils;
import android.util.Log;

import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
```

```
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

public class MyRenderer implements GLSurfaceView.Renderer {

    // Контекст приложения для доступа к ресурсам
    private Context context;

    // Дескриптор программы шейдера (не используется напрямую, но может быть полезен в будущем)
    private int program_Handle;

    // Общее количество объектов на сцене
    private int countObject = 6;

    // Массив идентификаторов текстур OpenGL
    private int[] textureName = new int[countObject];

    // ID ресурсов текстур в Android (R.drawable.*), соответствующих каждому объекту
    private int[] resourceID = {
            R.drawable.watermelon,
            R.drawable.apple,
            R.drawable.bannana,
            R.drawable.orange,
            R.drawable.kitchen_table__,
            R.drawable.coffee
    };

    // Имена файлов 3D-моделей формата .obj
    private String[] objName = {
            "SMK_JJ0KQAO2_Watermelon_2.91.obj",
            "Apple.obj", "Banana.obj",
            "Orange.obj", "Kitchen_Table_.obj",
            "MugL1.obj"
    };

    // Массив путей к локально скопированным .obj файлам
    private String[] objPath = new String[countObject];

    public MyRenderer(Context c) {
        this.context = c; // Сохраняем контекст для дальнейшего использования
    }

    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig eglConfig) {
        // Вызываем нативную функцию, которая создаёт и компилирует шейдеры
        String str = LoadLibJNIWrapper.on_surface_created();
        Log.i("TagShaderError", str); // Логируем возможные ошибки шейдера
    }

    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        // Цикл создания текстур для всех объектов
        for (int i = 0; i < countObject; ++i) {
            // Генерация нового текстурного объекта
            int[] names = new int[1];
            GLES20.glGenTextures(1, names, 0);
            this.textureName[i] = names[0]; // Сохраняем его дескриптор
```

```
            // Привязываем текстуру к типу GL_TEXTURE_2D
            GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureName[i]);

            // Устанавливаем параметры фильтрации текстуры
            GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MIN_FILTER,
GLES20.GL_LINEAR_MIPMAP_LINEAR);
            GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MAG_FILTER,
GLES20.GL_LINEAR);

            // Повтор текстуры по осям S и T (горизонталь и вертикаль)
            GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_S,
GLES20.GL_REPEAT);
            GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_T,
GLES20.GL_REPEAT);

            // Загрузка Bitmap из ресурса
            Bitmap bitmap = BitmapFactory.decodeResource(this.context.getResources(), this.resourceID[i]);

            // Передача текстуры в OpenGL
            GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0);

            // Освобождаем память после загрузки
            bitmap.recycle();

            // Создаём mipmap-уровни для улучшения качества текстурирования
            GLES20.glGenerateMipmap(GLES20.GL_TEXTURE_2D);
        }

        // Подготавливаем временные файлы для хранения .obj моделей
        for (int i = 0; i < countObject; ++i) {
            File file = new File(context.getCacheDir() + "/" + objName[i]);
            if (!file.exists()) try {
                // Открываем модель из assets
                InputStream is = context.getAssets().open(objName[i]);

                // Читаем содержимое файла в буфер
                int size = is.available();
                byte[] buffer = new byte[size];
                is.read(buffer);
                is.close();

                // Записываем буфер во временный файл
                FileOutputStream fos = new FileOutputStream(file);
                fos.write(buffer);
                fos.close();
            } catch (Exception e) {
                // Если произошла ошибка — выкидываем RuntimeException
                throw new RuntimeException(e);
            }
            // Сохраняем путь к созданному файлу
            objPath[i] = file.getPath();
        }

        // Передаём в C++ ширину, высоту экрана и пути к .obj файлам
        String str = LoadLibJNIWrapper.on_surface_changed(width, height, objPath);
```

```
        // Логируем результат выполнения on_surface_changed
        Log.i("TagError", str);
    }

    @Override
    public void onDrawFrame(GL10 gl) {
        // Устанавливаем цвет очистки экрана (черный)
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

        // Очищаем цветовой и глубинный буферы перед новым кадром
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT | GLES20.GL_DEPTH_BUFFER_BIT);

        // Цикл отрисовки каждого объекта сцены
        for (int i = 0; i < countObject; ++i) {
            // Активируем нужный текстурный блок
            if (i == 0) {
                GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
            } else {
                GLES20.glActiveTexture(GLES20.GL_TEXTURE0 + i);
            }

            // Привязываем ранее загруженную текстуру к текущему блоку
            GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, this.textureName[i]);

            // Вызываем нативную функцию для отрисовки текущего объекта
            LoadLibJNIWrapper.on_draw_frame(i);
        }
    }
}
```

4. Файл LoadLibJNIWrapper.java содержит класс LoadLibJNIWrapper с нативными методами, которые позволяют Java взаимодействовать с C++ кодом через JNI. Здесь объявляются методы on_surface_created, on_surface_changed, on_draw_frame, которые вызываются из MyRenderer. Они используются для инициализации OpenGL ES, передачи данных о моделях и выполнения рендеринга.

```
package com.example.opengl_kurs_model;

public class LoadLibJNIWrapper {
    static {
        System.loadLibrary("opengl_kurs_modeld");
    }

    public static native String on_surface_created(); // Нативный метод создания шейдера
    public static native String on_surface_changed(int width, int height, String[] paths); // Настройка
сцены
    public static native void on_draw_frame(int index); // Рендеринг кадра
}
```

5. Точкой входа в нативный код является файл native-lib.cpp, содержащий реализацию JNI-функций, через которые Java вызывает C++. Функции Java_com_example_opengl_1kurs_1model_LoadLibJNIWrapper_on_1surface_1created и другие используются для связи Java и C++. Эти функции вызывают соответствующие функции из model_loading.h/.cpp.

```
#include <jni.h>
#include <string>
```

```
#include "model_loading.h"

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_opengl_1kurs_1model_MainActivity_stringFromJNI(
        JNIEnv* env, jobject /* this */) {
    std::string hello = "Hello from C++";
    return env->NewStringUTF(hello.c_str());
}

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_opengl_1kurs_1model_LoadLibJNIWrapper_on_1surface_1created(
        JNIEnv* env, jclass clazz) {
    std::string str = on_surface_created(); // Вызов нативной функции создания шейдера
    return env->NewStringUTF(str.c_str());
}

std::vector<const char*> nativeMas;

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_opengl_1kurs_1model_LoadLibJNIWrapper_on_1surface_1changed(
        JNIEnv* env, jclass clazz, jint width, jint height, jobjectArray paths) {
    jint count = env->GetArrayLength(paths);
    for (int i = 0; i < count; ++i) {
        auto elem = (jstring)env->GetObjectArrayElement(paths, i);
        const char* str = env->GetStringUTFChars(elem, nullptr);
        nativeMas.push_back(str); // Сохранение путей к .obj файлам
    }
    std::string str = on_surface_changed(width, height, nativeMas); // Инициализация сцены
    return env->NewStringUTF(str.c_str());
}

extern "C" JNIEXPORT void JNICALL
Java_com_example_opengl_1kurs_1model_LoadLibJNIWrapper_on_1draw_1frame(
        JNIEnv* env, jclass clazz, jint index) {
    on_draw_frame(index); // Рендеринг кадра
}
```

6. В файле model_loading.h объявляются глобальные переменные, структуры и функции, используемые в model_loading.cpp. Определяется шейдер, матрица проекции, вектор моделей и т. д. Также объявляются внешние функции on_surface_created, on_surface_changed и on_draw_frame, которые вызываются из Java.

```
#pragma once
#include <GLES2/gl2.h>
#include <GLES2/gl2ext.h>
#include <GLES2/gl2platform.h>
#include <android/asset_manager.h>
#include <android/asset_manager_jni.h>
#include "Model.h"

std::string on_surface_created();
std::string on_surface_changed(int width, int height, std::vector<const char*> path);
void on_draw_frame(int index);
```

7. Файл model_loading.cpp содержит главные функции управления сценой: создание шейдера, инициализация матриц преобразования, загрузка моделей и анимация камеры. Здесь определён статический код вершинного и

фрагментного шейдера. Также вычисляются матрицы MVP, задаются начальные позиции и масштабы моделей. В on_draw_frame реализуется движение камеры по окружности вокруг сцены.

```
#include "model_loading.h"
static const char vertex_shader[] =
    "precision mediump float;"
    "uniform mat4 u_mvpMatrix;"
    "attribute vec3 a_Position;"
    "attribute vec2 a_TextureCoordinates;"
    "varying vec2 v_TextureCoordinates;"
    "void main(){"
    "v_TextureCoordinates = a_TextureCoordinates;"
    "gl_Position = u_mvpMatrix * vec4(a_Position, 1.0);"
    "}";

static const char fragment_shader[] =
    "precision mediump float;"
    "uniform sampler2D u_TextureUnit;"
    "varying vec2 v_TextureCoordinates;"
    "void main(){"
    "gl_FragColor = texture2D(u_TextureUnit, v_TextureCoordinates);"
    "}";

Shader mainShader;
int masN;
std::vector<Model> modelMas;
std::vector<glm::mat4> arrayMatrixModel;
glm::mat4 mainProjection;
double angle = 0;

std::string on_surface_created() {
    glEnable(GL_DEPTH_TEST); // Включение теста глубины
    Shader shader(vertex_shader, fragment_shader);
    mainShader = shader;
    mainShader.createProgram(); // Компиляция шейдера
    return mainShader.strError;
}

std::string on_surface_changed(int width, int height, std::vector<const char*> path) {
    std::string strSum;
    glm::mat4 view, model;

    std::vector<float> objPos = {
        -0.2, -0.23, -0.15, // Арбуз
        -0.35, -0.12, 0.2, // Яблоко
        0.3, -0.2, 0.0, // Банан
        0.0, -0.12, 0.2, // Апельсин
        0.0, -1.5, 0.0, // Стол
        0.6, -0.23, 0.1 // Чашка
    };

    mainProjection = glm::perspective(glm::radians((float)45.0f), (float)width / (float)height, (float)0.1f,
(float)100.0f); // Перспективная проекция

    view = glm::mat4(1.0f);
    view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f)); // Позиция камеры
```

```
    masN = (int)path.size();
    for (int i = 0, iPos = 0; i < masN; ++i) {
        std::string pStr = path.at(i);
        std::string subStr = pStr.substr(pStr.rfind('/') + 1);
        strSum += subStr;
        strSum += "\n";

        model = glm::mat4(1.0f);
        model = glm::translate(model, glm::vec3(objPos.at(iPos), objPos.at(iPos + 1), objPos.at(iPos + 2)));
        if (subStr == "SMK_JJ0KQAO2_Watermelon_2.91.obj" || subStr == "Kitchen_Table_.obj") {
            model = glm::scale(model, glm::vec3(1.5f, 1.5f, 1.5f));
        } else if (subStr == "Apple.obj" || subStr == "Banana.obj" || subStr == "Orange.obj") {
            if (subStr == "Banana.obj") {
                model = glm::rotate(model, glm::radians(-90.0f), glm::vec3(1.0f, 0.0f, 0.0f));
            }
            model = glm::scale(model, glm::vec3(0.003f, 0.003f, 0.003f));
        } else if (subStr == "MugL1.obj") {
            model = glm::rotate(model, glm::radians(-90.0f), glm::vec3(1.0f, 0.0f, 0.0f));
            model = glm::scale(model, glm::vec3(0.05f, 0.05f, 0.05f));
        } else {
            model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
        }
        arrayMatrixModel.push_back(model);
        glm::mat4 mvpMatrix = mainProjection * view * model;
        Model model1(path.at(i), mvpMatrix, width, height, mainShader);
        modelMas.push_back(model1);
        iPos += 3;
    }
    return strSum;
}

void on_draw_frame(int index) {
    glm::mat4 view = glm::mat4(1.0f);
    float radius = 4.0f;

    auto camX = (float)(sin(angle) * radius);
    auto camZ = (float)(cos(angle) * radius);
    view = glm::lookAt(glm::vec3(camX, 2.0f, camZ),
                       glm::vec3(0.0f, 0.0f, 0.0f),
                       glm::vec3(0.0f, 1.0f, 0.0f)); // Камера вращается вокруг сцены

    glm::mat4 mvpMatrix = mainProjection * view * arrayMatrixModel.at(index);
    modelMas.at(index).setMVP(mvpMatrix);
    modelMas.at(index).Draw(mainShader, index);

    if (angle >= 2 * M_PI) {
        angle = 0;
    } else if (index == masN - 1) {
        angle += 0.01; // Обновление угла поворота камеры
    } }
```

8. Объявление класса Model, который представляет 3D-модель содержится в файле Model.h. десь определены методы загрузки модели, её отрисовки, установки матрицы MVP. Также объявляются приватные поля: вектор мешей, директория модели, загруженные текстуры и матрица MVP. Метод loadModel использует Assimp для парсинга .obj файла.

```
#pragma once
// Подключаем стандартные библиотеки
#include <vector>       // Для хранения мешей и текстур
#include <string>       // Для работы с путями и именами
#include <assimp/Importer.hpp>  // Assimp: для загрузки моделей из .obj файлов
#include <assimp/scene.h>      // Сцена — содержит всю информацию о модели после загрузки
#include <assimp/postprocess.h> // Постобработка (например, триангуляция)
#include <stb_image.h>         // Для загрузки текстур из изображений
#include <android/log.h>       // Логирование на Android через __android_log_print
#include <sstream>             // Для формирования сообщений об ошибках

// Подключаем собственные заголовочные файлы
#include "Mesh.h"              // Класс Mesh — минимальная единица отрисовки

// Класс Model представляет собой трёхмерную модель, состоящую из одного или нескольких мешей.
// Он использует библиотеку Assimp для загрузки моделей из .obj файлов.
class Model {
public:
    // Конструктор по умолчанию
    Model() {}

    // Основной конструктор
    // Принимает путь к .obj файлу, матрицу MVP, размеры экрана и ссылку на шейдер
    Model(const char* path, glm::mat4 mvpMatrix, int width, int height, Shader& shader) {
        this->mvpMatrix = mvpMatrix;     // Сохраняем матрицу MVP
        loadModel(path, shader);         // Вызываем метод загрузки модели
    }

    // Метод Draw вызывает отрисовку всех мешей модели
    void Draw(Shader& shader, int textureUnit);

    // Обновляет матрицу MVP для всей модели
    void setMVP(glm::mat4 modelViewProjectionMatrix);

    std::string strError = "Correct work!"; // Хранит информацию об ошибках загрузки модели

private:
    glm::mat4 mvpMatrix;            // Матрица модели-вида-проекции
    std::vector<Mesh> meshes;       // Вектор мешей — основных частей модели
    std::string directory;          // Директория, где находится модель (для поиска текстур)
    std::vector<Texture> textures_loaded;  // Уже загруженные текстуры, чтобы не загружать дубликаты

    // Загружает модель из .obj файла и разбивает её на меши
    void loadModel(const std::string& path, Shader& shader);

    // Рекурсивно обрабатывает узлы модели. Assimp использует древовидную структуру.
    void processNode(aiNode* node, const aiScene* scene, Shader& shader);

    // Обрабатывает один меш модели: извлекает вершины, индексы, текстурные координаты
    Mesh processMesh(aiMesh* mesh, const aiScene* scene, Shader& shader);
};
```

9. Реализация класса Model находится в файле Model.cpp. Здесь происходит загрузка модели через Assimp, обработка узлов и мешей, применение координат вершин и текстур. Метод processMesh извлекает данные вершин и

текстурных координинат. Метод loadModel открывает .obj файл и начинает обработку сцены.

```
#include "Model.h"
// Метод загружает модель из .obj файла и обрабатывает её с помощью Assimp
void Model::loadModel(const std::string& path, Shader& shader) {
    // Создаем импортер Assimp
    Assimp::Importer importer;
    // Загружаем модель. Флаги:
    // aiProcess_Triangulate — превращаем все полигоны в треугольники
    // aiProcess_GenSmoothNormals — генерируем гладкие нормали для освещения
    // aiProcess_FlipUVs — переворачиваем текстурные координаты по Y (чтобы правильно отображалась текстура)
    // aiProcess_CalcTangentSpace — вычисляем тангентное пространство (для нормальных карт и других эффектов)
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate |
                                             aiProcess_GenSmoothNormals |
                                             aiProcess_FlipUVs |
                                             aiProcess_CalcTangentSpace);

    // Если модель не загрузилась — выводим ошибку и сохраняем её в strError
    if (!scene) {
        std::cout << "ERROR::ASSIMP::" << importer.GetErrorString() << std::endl;
        this->strError = "ERROR::ASSIMP::";
        this->strError += importer.GetErrorString();
        return;
    }
    // Сохраняем директорию модели, чтобы можно было находить связанные файлы, например, текстуры (.mtl)
    this->directory = path.substr(0, path.find_last_of('/'));

    // Обрабатываем корневой узел сцены рекурсивно
    processNode(scene->mRootNode, scene, shader);
}
// Рекурсивная обработка узлов сцены. Каждый узел может содержать несколько мешей.
void Model::processNode(aiNode* node, const aiScene* scene, Shader& shader) {
    // Перебираем все меши текущего узла
    for (unsigned int i = 0; i < node->mNumMeshes; ++i) {
        // Получаем сам меш из сцены по индексу
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];

        // Обрабатываем меш и добавляем его в вектор meshes
        this->meshes.push_back(processMesh(mesh, scene, shader));
    }

    // Рекурсивно обрабатываем дочерние узлы (если они есть)
    for (unsigned int i = 0; i < node->mNumChildren; ++i) {
        processNode(node->mChildren[i], scene, shader);
    }
}
// Обработка одного меша: извлекаем вершины и индексы
Mesh Model::processMesh(aiMesh* mesh, const aiScene* scene, Shader& shader) {
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;

    // Извлекаем данные о вершинах
    for (unsigned int i = 0; i < mesh->mNumVertices; ++i) {
        Vertex vertex;
```

```
        glm::vec3 vector;

        // Координаты вершин
        vector.x = mesh->mVertices[i].x;
        vector.y = mesh->mVertices[i].y;
        vector.z = mesh->mVertices[i].z;
        vertex.Position = vector;

        // Текстурные координаты
        if (mesh->mTextureCoords[0]) { // если доступны
            glm::vec2 vec;
            vec.x = mesh->mTextureCoords[0][i].x;
            vec.y = mesh->mTextureCoords[0][i].y;
            vertex.TexCoordinates = vec;
        } else {
            // Если текстурных координат нет — ставим нулевые значения
            vertex.TexCoordinates = glm::vec2(0.0f, 0.0f);
        }

        // Добавляем вершину в вектор
        vertices.push_back(vertex);
    }

    // Извлекаем индексы полигонов
    for (unsigned int i = 0; i < mesh->mNumFaces; ++i) {
        aiFace face = mesh->mFaces[i];

        // Каждая грань состоит из нескольких вершин — обычно это треугольник
        for (unsigned int j = 0; j < face.mNumIndices; ++j) {
            indices.push_back(face.mIndices[j]); // Добавляем индексы в список
        }
    }

    // Возвращаем созданный меш, передавая ему вершины, индексы и матрицу MVP
    return Mesh(vertices, indices, shader, this->mvpMatrix);
}

// Метод обновляет матрицу MVP для всей модели
void Model::setMVP(glm::mat4 modelViewProjectionMatrix) {
    this->mvpMatrix = modelViewProjectionMatrix;

    // Обновляем матрицу у каждого меша, чтобы он тоже использовал новую матрицу
    int meshLength = this->meshes.size();
    for (int i = 0; i < meshLength; ++i) {
        this->meshes.at(i).setMVP(this->mvpMatrix);
    }
}

// Метод Draw рисует всю модель, вызывая Draw для каждого меша
void Model::Draw(Shader& shader, int textureUnit) {
    for (unsigned int i = 0; i < this->meshes.size(); ++i) {
        meshes[i].Draw(shader, textureUnit); // Отрисовка каждого меша
    }
}
```

10. В файле Mesh.h находится определение структур Vertex и Texture, а также класса Mesh. Mesh содержит VBO, EBO, атрибуты шейдера и методы для отрисовки. Также здесь объявляются методы setMVP и Draw.

```
#pragma once
// Подключаем библиотеку GLM для работы с математикой 3D-графики
#include <glm/vec3.hpp>           // glm::vec3 — трёхмерные векторы (для позиций вершин)
#include <glm/vec2.hpp>           // glm::vec2 — двухмерные векторы (для текстурных координат)
#include <glm/glm.hpp>            // Основные функции GLM
#include <glm/gtc/matrix_transform.hpp>  // Функции преобразования матриц (rotate, translate, scale)
#include <glm/gtc/type_ptr.hpp>     // Для передачи матриц в шейдеры через указатель

// Стандартные библиотеки C++ для хранения данных
#include <vector>                 // std::vector — для хранения вершин и индексов
#include <string>                 // std::string — для работы с путями и типами текстур

// Библиотека OpenGL ES 2.0 для работы с графикой на Android
#include <GLES2/gl2.h>
#include <GLES2/gl2ext.h>
#include <GLES2/gl2platform.h>

// Заголовочный файл Shader.h — позволяет использовать шейдеры в Mesh
#include "Shader.h"

// Структура Vertex описывает одну вершину модели:
// - Position: координаты точки в пространстве
// - TexCoordinates: текстурные координаты (UV) для наложения изображения
struct Vertex {
    glm::vec3 Position;         // Позиция вершины
    glm::vec2 TexCoordinates;   // Текстурные координаты (U и V)
};

// Структура Texture содержит информацию о текстуре:
// - id — уникальный идентификатор текстуры в OpenGL
// - type — тип текстуры (например, диффузная или нормальная карта)
// - path — путь к текстуре, чтобы не загружать одинаковые текстуры дважды
struct Texture {
    unsigned int id;            // Уникальный номер текстуры в OpenGL
    std::string type;           // Тип текстуры (например, map_Kd — диффузная)
    std::string path;           // Путь к файлу текстуры
};

// Класс Mesh представляет минимальную единицу модели.
// Он содержит данные о вершинах, индексах и умеет себя отрисовать в OpenGL ES 2.0.
class Mesh {
public:
    // Вектор вершин и индексов — основные данные для рендеринга
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;

    // Конструктор принимает вершины, индексы, шейдер и начальную матрицу MVP
    Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices, Shader& shader, glm::mat4
mvpMatrix);

    // Метод Draw отрисовывает меш с использованием текущего шейдера и текстурного блока
    void Draw(Shader& shader, int textureUnit);
```

```
    // Обновляет матрицу MVP у меша
    void setMVP(glm::mat4 modelViewProjectionMatrix);

private:
    // Дескрипторы OpenGL объектов:
    GLuint VBO;     // Vertex Buffer Object — хранит вершины
    GLuint EBO;     // Element Buffer Object — хранит индексы
    GLuint programID; // ID программы шейдера

    // Текущая матрица модели-вида-проекции
    glm::mat4 mvpMatrix;

    // Локации атрибутов в шейдере:
    GLint posAttribute;       // Атрибут позиции вершины
    GLint posTexAttribute;    // Атрибут текстурных координат
    GLint mvpUniform;         // Uniform-переменная для матрицы MVP

    // Метод setupMesh() создаёт и заполняет VBO и EBO данными вершин и индексов
    void setupMesh();
};
```

11. В файле Mesh.cpp находится реализация класса Mesh. Здесь происходит инициализация буферов вершин и индексов, настройка атрибутов шейдера и отрисовка модели. Метод setupMesh инициализирует VBO и EBO, а Draw выполняет отрисовку модели с использованием текущего шейдера и матрицы MVP.

```
#include "Mesh.h"
// Конструктор Mesh
// Принимает:
// - вершины модели (vertices)
// - индексы для соединения вершин (indices)
// - шейдер и начальную матрицу MVP
Mesh::Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices, Shader& shader, glm::mat4
mvpMatrix) {
    // Получаем дескриптор программы шейдера, чтобы настроить атрибуты и uniform-переменные
    GLuint programHandler = shader.getProgram();

    // Сохраняем данные вершин и индексов локально
    this->vertices = vertices;
    this->indices = indices;

    // Сохраняем текущую матрицу MVP
    this->mvpMatrix = mvpMatrix;

    // Получаем локации атрибутов и uniform-переменной в шейдере
    this->posAttribute = glGetAttribLocation(programHandler, "a_Position");
    this->posTexAttribute = glGetAttribLocation(programHandler, "a_TextureCoordinates");
    this->mvpUniform = glGetUniformLocation(programHandler, "u_mvpMatrix");

    // Вызываем setupMesh(), чтобы подготовить VBO и EBO для рендеринга
    setupMesh();
}

// Метод setupMesh создаёт и заполняет буферы вершин и индексов (VBO и EBO)
void Mesh::setupMesh() {
```

```
    // Генерируем объекты буферов
    glGenBuffers(1, &this->VBO);
    glGenBuffers(1, &this->EBO);

    // Привязываем VBO как массив вершин и загружаем туда данные
    glBindBuffer(GL_ARRAY_BUFFER, this->VBO);
    glBufferData(GL_ARRAY_BUFFER,
            static_cast<int>(vertices.size() * sizeof(Vertex)),
            &vertices[0],
            GL_STATIC_DRAW);  // Данные не будут меняться

    // Привязываем EBO как массив индексов и передаём данные
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
            static_cast<int>(indices.size() * sizeof(unsigned int)),
            &indices[0],
            GL_STATIC_DRAW);

    // Отвязываем буферы после настройки — это хорошая практика
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}

// Обновляет матрицу MVP у меша
// Используется при движении камеры или объекта
void Mesh::setMVP(glm::mat4 modelViewProjectionMatrix) {
    this->mvpMatrix = modelViewProjectionMatrix;
}

// Основной метод отрисовки меша
void Mesh::Draw(Shader& shader, int textureUnit) {
    // Активируем программу шейдера
    shader.useProgram();

    // Передаём номер текстурного блока в униформу sampler2D в шейдере
    glUniform1i(glGetUniformLocation(shader.getProgram(), "u_TextureUnit"), textureUnit);

    // Передаём матрицу MVP в шейдер
    glUniformMatrix4fv(this->mvpUniform, 1, false, glm::value_ptr(this->mvpMatrix));

    // Привязываем буферы вершин и индексов
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->EBO);
    glBindBuffer(GL_ARRAY_BUFFER, this->VBO);

    // Включаем атрибуты вершин и текстурных координинат
    glEnableVertexAttribArray(this->posAttribute);
    glVertexAttribPointer(this->posAttribute,
                3,
                GL_FLOAT,
                GL_FALSE,
                sizeof(Vertex),
                nullptr);  // Смещение 0 для позиции

    glEnableVertexAttribArray(this->posTexAttribute);
    glVertexAttribPointer(this->posTexAttribute,
                2,
                GL_FLOAT,
```

```
                GL_FALSE,
                sizeof(Vertex),
                (void*)(sizeof(float)*3));  // Смещение 3 float'a — после Position

        // Рисуем модель через индексы: используем GL_TRIANGLES и EBO
        glDrawElements(GL_TRIANGLES,
                static_cast<int>(this->indices.size()),
                GL_UNSIGNED_INT,
                nullptr);

        // После отрисовки отвязываем буферы
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }
```

12. В файле Shader.h объявляется класс Shader, который отвечает за компиляцию вершинного и фрагментного шейдера, а также хранит дескриптор программы шейдера и ошибок.

```
#pragma once
// Подключаем библиотеку OpenGL ES 2.0
#include <GLES2/gl2.h>
// Стандартные библиотеки C++ для работы со строками и потоками
#include <iostream>
#include <sstream>   // Для формирования сообщений об ошибках
#include <string>   // Для хранения шейдерных строк и ошибок
// Класс Shader отвечает за создание и использование программ шейдеров в OpenGL ES 2.0.
// Шейдеры — это небольшие программы, выполняемые на GPU:
// - Вершинный шейдер: вычисляет позиции вершин
// - Фрагментный шейдер: определяет цвет каждого пикселя
class Shader {
public:
    // Конструктор по умолчанию
    Shader();

    // Основной конструктор: принимает код вершинного и фрагментного шейдера
    Shader(const std::string& vertexStr, const std::string& fragmentStr);

    // Создаёт и компилирует программу шейдера
    void createProgram();

    // Активирует программу шейдера перед рендерингом объекта
    void useProgram() const;

    // Возвращает дескриптор программы (GLuint) для дальнейшего использования
    unsigned int getProgram() const;

    // Сообщение о состоянии шейдера (успешно или нет)
    std::string strError = "Shader is work!";

private:
    // Строки с исходным кодом шейдеров
    std::string vertexProg, fragmentProg;

    // Дескрипторы шейдеров и программы
    unsigned int vertexID, fragmentID, program_Handle;
```

```
    // Проверяет, успешно ли скомпилировался шейдер или программа
    void checkCompileError(GLuint shader, const std::string& type);
};
```

13. В файле Shader.cpp содержится реализация класса Shader. Здесь создаются шейдеры, компилируются, проверяются на ошибки и объединяются в программу. Также реализованы методы активации программы и получения дескриптора программы.

```
#include "Shader.h"
// Конструктор по умолчанию — создаёт пустой шейдер
Shader::Shader() {}
// Основной конструктор — принимает код вершинного и фрагментного шейдера
Shader::Shader(const std::string& vertexStr, const std::string& fragmentStr) {
    // Преобразуем строки с кодом шейдеров в C-строки для OpenGL
    const char* vShaderCode = vertexStr.c_str();
    const char* fShaderCode = fragmentStr.c_str();

    // Создаем вершинный шейдер и загружаем в него исходный код
    this->vertexID = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(this->vertexID, 1, &vShaderCode, nullptr);
    glCompileShader(this->vertexID); // Компилируем вершинный шейдер
    checkCompileError(this->vertexID, "VERTEX"); // Проверяем ошибки компиляции

    // Создаем фрагментный шейдер и загружаем в него исходный код
    this->fragmentID = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(this->fragmentID, 1, &fShaderCode, nullptr);
    glCompileShader(this->fragmentID); // Компилируем фрагментный шейдер
    checkCompileError(this->fragmentID, "FRAGMENT"); // Проверяем ошибки компиляции
    this->program_Handle = 0; // Инициализируем дескриптор программы как 0
}
// Метод checkCompileError проверяет, успешно ли прошла компиляция шейдера или линковка
программы
void Shader::checkCompileError(GLuint shader, const std::string& type) {
    GLint state;
    GLchar infoLog[1024];
    std::stringstream strs;

    if (type != "PROGRAM") {
        // Если проверяем шейдер
        glGetShaderiv(shader, GL_COMPILE_STATUS, &state);
        if (!state) {
            // Если ошибка компиляции — получаем текст ошибки
            glGetShaderInfoLog(shader, 1024, nullptr, infoLog);
            strs << "ERROR::SHADER_COMPILATION_ERROR of type:" << type << "\n" << infoLog;
            this->strError = strs.str(); // Сохраняем сообщение об ошибке
        }
    } else {
        // Если проверяем программу
        glGetProgramiv(shader, GL_LINK_STATUS, &state);
        if (!state) {
            // Если ошибка линковки — получаем текст ошибки
            glGetProgramInfoLog(shader, 1024, nullptr, infoLog);
            // Добавляем информацию к строке ошибки
            this->strError += "ERROR::PROGRAM_LINK_ERROR:";
            this->strError += infoLog;
        }
    }
```

```
}
// Метод createProgram собирает программу из ранее созданных шейдеров
void Shader::createProgram() {
    this->program_Handle = glCreateProgram(); // Создаем объект программы

    // Прикрепляем скомпилированные шейдеры к программе
    glAttachShader(this->program_Handle, this->vertexID);
    glAttachShader(this->program_Handle, this->fragmentID);

    // Линкуем шейдеры в единую программу
    glLinkProgram(this->program_Handle);

    // Проверяем на ошибки линковки
    checkCompileError(this->program_Handle, "PROGRAM");

    // После линковки шейдеры больше не нужны — удаляем их
    glDeleteShader(this->vertexID);
    glDeleteShader(this->fragmentID);
}

// Активирует текущую шейдерную программу перед отрисовкой
void Shader::useProgram() const {
    glUseProgram(this->program_Handle);
}

// Возвращает дескриптор программы для использования в других частях кода
unsigned int Shader::getProgram() const {
    return this->program_Handle;
}
```

14. Файл CMakeLists.txt настраивает сборку проекта в CMake. Здесь подключаются сторонние библиотеки (Assimp, GLM, stb_image), указываются пути к исходникам, а также определяется, какие библиотеки будут использоваться при сборке. Также указывается, что используется OpenGL ES 2.0.

```
cmake_minimum_required(VERSION 3.18.1)
project("opengl_kurs_model")

add_subdirectory(glm)
add_subdirectory(${CMAKE_SOURCE_DIR}/Assimp)
include_directories(${CMAKE_SOURCE_DIR}/Assimp/include)

set(ASSIMP_LIB assimp)

add_library(opengl_kurs_model SHARED
    native-lib.cpp
    Mesh.cpp
    Shader.cpp
    Model.cpp
    stb_image.cpp
    model_loading.cpp)

find_library(log-lib log)
find_path(GLES2_DIR GLES2/gl2.h HINTS ${ANDROID_NDK})
find_library(GL20 libGLESv2.so HINTS ${GLES2_DIR}/../lib)
```

```
target_link_libraries(opengl_kurs_model ${log-lib} ${GL20} glm ${ASSIMP_LIB})
```

Требования к отчету

Результаты выполненной курсовой работы должны быть оформлены в формате текстового редактора Word, размером шрифта 14 пунктов. Отчет должен содержать:

- Титульный лист.
- Содержание. В содержание включают номера и наименования разделов и подразделов с указанием номеров страниц. В содержание также включают все приложения, вошедшие в данный документ, с указанием номера страницы.
- Введение. Обзор предметной области по заданной теме, а также описание среды разработки и инструментов в общем объеме 1 страница.
- Текст задания, соответствующий Вашему варианту.
- Реализация проекта. Сценарии взаимодействия пользователя со сценой; описание элементов сцены и их геометрии; настройка освещения и камеры; использование шейдеров и управления потоком рендеринга.
- Результаты. Описание используемых методов, скриншоты работы приложения, демонстрирующие реализованный функционал. Ссылка на исходные коды приложения.
- Заключение. Сделать выводы о проделанной работе, описать возникшие проблемы в процессе выполнения работы и пути их решения.
- Приложение. Код программы с комментариями.
- Список используемых источников. Перечислить используемые источники литературы.

Отчет о курсовой работе должен соответствовать плану задания и содержать не менее 20 страниц поясняющего текста (не считая исходного кода программ), подготовленного в формате текстового редактора Word, размером шрифта 14 пунктов. Код программы должен быть представлен в разделе отчета «приложение 1» и должен содержать комментарии.

Список дополнительных источников

1. Баяковский Ю. М., Игнатенко А. В., Фролов А. И. *Графическая библиотека OpenGL* : учебно-методическое пособие. — Москва : Издательский отдел факультета ВМиК МГУ, 2003. — 132 с. — ISBN 5-89407-153-4. *(https://disk.yandex.ru/i/JQez-Sm-oQI83g)*
2. Официальная документация: Android NDK. https://developer.android.com/ndk?spm=a2ty_o01.29997173.0.0.56d0c921u8Sxy8
3. JNI (Java Native Interface) — документация Oracle: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/?spm=a2ty_o01.29997173.0.0.56d0c921u8Sxy8
4. Руководство по использованию Assimp: https://sourceforge.net/projects/assimp/