Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

Лабораторная работа 6

«OpenGL ES с использованием JNI»

Выполнил: студент 4 курса ИВТ,

гр. ИП-713

Михеев Н. А.

Проверил: ассистент кафедры ПМиК,

Павлова У.В.

Новосибирск, 2020 г.

Задание

Написать программу, рисующую куб с текстурой. Вся прорисовка должна быть реализована в JNI.

Решение и обоснование решения

За основу были взят код из официального репозитория ARM “OpenGL ES SDK”.

JNI (NDK – Native Development Kit) – набор инструментов, который помогает нам использовать C/C++ код с Android и обеспечивает платформенные библиотеки для управления и доступа к физическим компонентам устройства.

Используется в двух случаях:

1. получение дополнительной производительности от устройства для достижения меньших задержек, запуск вычислительно-интенсивных программ, таких как игры или симуляция физики.
2. Использование своих или сторонних библиотек C/C++.

Для использования доступны как стандартные библиотеки языков (Core C11/C++17), так и различные графические библиотеки (OpenGL ES, EGL, Vulkan) и т. д.

Программа с использованием NDK делится на две части: стандартную Android часть (Java) и native часть. В стандартной настраивается поверхность (surface), которая необходима для рисования графики, а также выполняем какие-то другие задачи. В native части мы описываем рендеринг сцены, она написана на C/C++.

Текстурирование выполнено разбиванием каждой грани куба на 9 частей и её последующей раскраской.

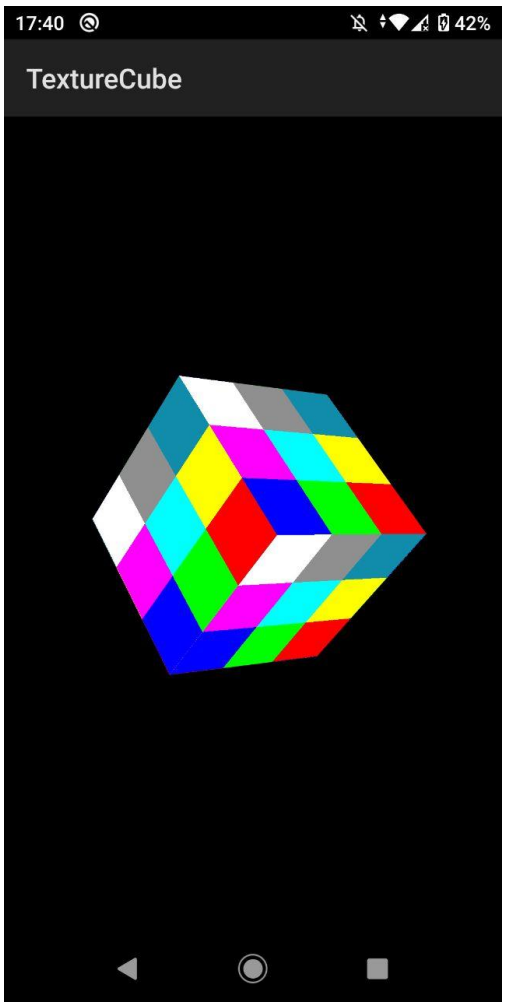

Рис. 1 – демонстрация работы программы.

Листинг программы

NativeLibrary.java

```
package com.arm.malideveloper.openglessdk.texturecube;

public class NativeLibrary
{
    static
    {
        System.loadLibrary("Native");
    }
    public static native void init(int width, int height);
    public static native void step();
}
```

TextureCube.java

```
public class TextureCube extends Activity
{
    private static String LOGTAG = "TextureCube";
    protected TutorialView graphicsView;
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
```

```
        super.onCreate(savedInstanceState);
        Log.i(LOGTAG, "Creating New Tutorial View");
        graphicsView = new TutorialView(getApplication());
        setContentView(graphicsView);
    }
    @Override protected void onPause()
    {
        super.onPause();
        graphicsView.onPause();
    }
    @Override protected void onResume()
    {
        super.onResume();
        graphicsView.onResume();
    }

}
```

TutorialView.java

```
package com.arm.malideveloper.openglessdk.texturecube;

import android.content.Context;
import android.opengl.GLSurfaceView;
import javax.microedition.khronos.egl.EGL10;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.egl.EGLContext;
import javax.microedition.khronos.egl.EGLDisplay;
import javax.microedition.khronos.opengles.GL10;

class TutorialView extends GLSurfaceView
{
    protected int redSize = 5;
    protected int greenSize = 6;
    protected int blueSize = 5 ;
    protected int alphaSize = 0;
    protected int depthSize = 16;
    protected int sampleSize = 4;
    protected int stencilSize = 0;
    protected int[] value = new int [1];
    public TutorialView(Context context)
    {
        super(context);
        setEGLContextFactory(new ContextFactory());
        setEGLConfigChooser(new ConfigChooser());
        setRenderer(new Renderer());
    }

    private static class ContextFactory implements GLSurfaceView.EGLContextFactory
    {
        public EGLContext createContext(EGL10 egl, EGLDisplay display, EGLConfig
eglConfig)
        {
            final int EGL_CONTEXT_CLIENT_VERSION = 0x3098;
            int[] attrib_list = {EGL_CONTEXT_CLIENT_VERSION, 2, EGL10.EGL_NONE };
            EGLContext context = egl.eglCreateContext(display, eglConfig,
EGL10.EGL_NO_CONTEXT, attrib_list);
            return context;
        }
```

```
        public void destroyContext(EGL10 egl, EGLDisplay display, EGLContext context)
        {
            egl.eglDestroyContext(display,  context);
        }
    }

    protected class ConfigChooser implements GLSurfaceView.EGLConfigChooser
    {
        public EGLConfig chooseConfig(EGL10 egl, EGLDisplay display)
        {
            final int EGL_OPENGL_ES2_BIT = 4;
            int[] configAttributes =
            {
                EGL10.EGL_RED_SIZE, redSize,
                EGL10.EGL_GREEN_SIZE, greenSize,
                EGL10.EGL_BLUE_SIZE, blueSize,
                EGL10.EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
                EGL10.EGL_SAMPLES, sampleSize,
                EGL10.EGL_DEPTH_SIZE, depthSize,
                EGL10.EGL_STENCIL_SIZE, stencilSize,
                EGL10.EGL_NONE
            };
            int[] num_config = new int[1];
            egl.eglChooseConfig(display, configAttributes, null, 0, num_config);
            int numConfigs = num_config[0];
            EGLConfig[] configs = new EGLConfig[numConfigs];
            egl.eglChooseConfig(display, configAttributes, configs, numConfigs,
num_config);
            return configs[0];
        }
    }
        private static class Renderer implements GLSurfaceView.Renderer
        {
            public void onDrawFrame(GL10 gl)
            {
                NativeLibrary.step();
            }

            public void onSurfaceChanged(GL10 gl, int width, int height)
            {
                NativeLibrary.init(width, height);
            }

            public void onSurfaceCreated(GL10 gl, EGLConfig config)
            {

            }
        }
}
```

Matrix.cpp

```
#include "Matrix.h"
#include <cmath>

void matrixIdentityFunction(float* matrix)
{
    if(matrix == NULL)
```

```
    {
        return;
    }

    matrix[0] = 1.0f;
    matrix[1] = 0.0f;
    matrix[2] = 0.0f;
    matrix[3] = 0.0f;
    matrix[4] = 0.0f;
    matrix[5] = 1.0f;
    matrix[6] = 0.0f;
    matrix[7] = 0.0f;
    matrix[8] = 0.0f;
    matrix[9] = 0.0f;
    matrix[10] = 1.0f;
    matrix[11] = 0.0f;
    matrix[12] = 0.0f;
    matrix[13] = 0.0f;
    matrix[14] = 0.0f;
    matrix[15] = 1.0f;
}

void matrixTranslate(float* matrix, float x, float y, float z)
{
    float temporaryMatrix[16];
    matrixIdentityFunction(temporaryMatrix);
    temporaryMatrix[12] = x;
    temporaryMatrix[13] = y;
    temporaryMatrix[14] = z;
    matrixMultiply(matrix,temporaryMatrix,matrix);
}

void matrixMultiply(float* destination, float* operand1, float* operand2)
{
    float theResult[16];
    int row, column = 0;
    int i,j = 0;
    for(i = 0; i < 4; i++)
    {
        for(j = 0; j < 4; j++)
        {
            theResult[4 * i + j] = operand1[j] * operand2[4 * i] + operand1[4 + j] *
operand2[4 * i + 1] +
                operand1[8 + j] * operand2[4 * i + 2] + operand1[12 + j] * operand2[4
* i + 3];
        }
    }

    for(int i = 0; i < 16; i++)
    {
        destination[i] = theResult[i];
    }
}

void matrixFrustum(float* matrix, float left, float right, float bottom, float top,
float zNear, float zFar)
{
    float temp, xDistance, yDistance, zDistance;
    temp = 2.0 *zNear;
    xDistance = right - left;
    yDistance = top - bottom;
    zDistance = zFar - zNear;
```

```
    matrixIdentityFunction(matrix);
    matrix[0] = temp / xDistance;
    matrix[5] = temp / yDistance;
    matrix[8] = (right + left) / xDistance;
    matrix[9] = (top + bottom) / yDistance;
    matrix[10] = (-zFar - zNear) / zDistance;
    matrix[11] = -1.0f;
    matrix[14] = (-temp * zFar) / zDistance;
    matrix[15] = 0.0f;
}

void matrixPerspective(float* matrix, float fieldOfView, float aspectRatio, float
zNear, float zFar)
{
    float ymax, xmax;
    ymax = zNear * tanf(fieldOfView * M_PI / 360.0);
    xmax = ymax * aspectRatio;
    matrixFrustum(matrix, -xmax, xmax, -ymax, ymax, zNear, zFar);
}

void matrixRotateX(float* matrix, float angle)
{
    float tempMatrix[16];
    matrixIdentityFunction(tempMatrix);

    tempMatrix[5] = cos(matrixDegreesToRadians(angle));
    tempMatrix[9] = -sin(matrixDegreesToRadians(angle));
    tempMatrix[6] = sin(matrixDegreesToRadians(angle));
    tempMatrix[10] = cos(matrixDegreesToRadians(angle));
    matrixMultiply(matrix, tempMatrix, matrix);
}

void matrixRotateY(float *matrix, float angle)
{
    float tempMatrix[16];
    matrixIdentityFunction(tempMatrix);

    tempMatrix[0] = cos(matrixDegreesToRadians(angle));
    tempMatrix[8] = sin(matrixDegreesToRadians(angle));
    tempMatrix[2] = -sin(matrixDegreesToRadians(angle));
    tempMatrix[10] = cos(matrixDegreesToRadians(angle));
    matrixMultiply(matrix, tempMatrix, matrix);
}

void matrixRotateZ(float *matrix, float angle)
{
    float tempMatrix[16];
    matrixIdentityFunction(tempMatrix);

    tempMatrix[0] = cos(matrixDegreesToRadians(angle));
    tempMatrix[4] = -sin(matrixDegreesToRadians(angle));
    tempMatrix[1] = sin(matrixDegreesToRadians(angle));
    tempMatrix[5] = cos(matrixDegreesToRadians(angle));
    matrixMultiply(matrix, tempMatrix, matrix);
}

void matrixScale(float* matrix, float x, float y, float z)
{
    float tempMatrix[16];
    matrixIdentityFunction(tempMatrix);

    tempMatrix[0] = x;
```

```
    tempMatrix[5] = y;
    tempMatrix[10] = z;
    matrixMultiply(matrix, tempMatrix, matrix);
}

float matrixDegreesToRadians(float degrees)
{
        return M_PI * degrees / 180.0f;
}
```

Native.cpp

```
#include <jni.h>
#include <android/log.h>

#include <GLES2/gl2.h>
#include <GLES2/gl2ext.h>

#include <cstdio>
#include <cstdlib>
#include <cmath>

#include "Matrix.h"
#include "Texture.h"

#define LOG_TAG "libNative"
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_TAG, __VA_ARGS__)

/* [shaders] */
static const char glVertexShader[] =
        "attribute vec4 vertexPosition;\n"
        "attribute vec2 vertexTextureCord;\n"
        "varying vec2 textureCord;\n"
        "uniform mat4 projection;\n"
        "uniform mat4 modelView;\n"
        "void main()\n"
        "{\n"
        "    gl_Position = projection * modelView * vertexPosition;\n"
        "    textureCord = vertexTextureCord;\n"
        "}\n";

static const char glFragmentShader[] =
        "precision mediump float;\n"
        "uniform sampler2D texture;\n"
        "varying vec2 textureCord;\n"
        "void main()\n"
        "{\n"
        "    gl_FragColor = texture2D(texture, textureCord);\n"
        "}\n";
/* [shaders] */

GLuint loadShader(GLenum shaderType, const char* shaderSource)
{
    GLuint shader = glCreateShader(shaderType);
    if (shader != 0)
    {
        glShaderSource(shader, 1, &shaderSource, NULL);
        glCompileShader(shader);

        GLint compiled = 0;
```

```
        glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);

        if (compiled != GL_TRUE)
        {
            GLint infoLen = 0;
            glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);

            if (infoLen > 0)
            {
                char * logBuffer = (char*) malloc(infoLen);

                if (logBuffer != NULL)
                {
                    glGetShaderInfoLog(shader, infoLen, NULL, logBuffer);
                    LOGE("Could not Compile Shader %d:\n%s\n", shaderType,
logBuffer);
                    free(logBuffer);
                    logBuffer = NULL;
                }

                glDeleteShader(shader);
                shader = 0;
            }
        }
    }

    return shader;
}

GLuint createProgram(const char* vertexSource, const char * fragmentSource)
{
    GLuint vertexShader = loadShader(GL_VERTEX_SHADER, vertexSource);
    if (vertexShader == 0)
    {
        return 0;
    }

    GLuint fragmentShader = loadShader(GL_FRAGMENT_SHADER, fragmentSource);
    if (fragmentShader == 0)
    {
        return 0;
    }

    GLuint program = glCreateProgram();

    if (program != 0)
    {
        glAttachShader(program , vertexShader);
        glAttachShader(program, fragmentShader);
        glLinkProgram(program);
        GLint linkStatus = GL_FALSE;
        glGetProgramiv(program , GL_LINK_STATUS, &linkStatus);

        if(linkStatus != GL_TRUE)
        {
            GLint bufLength = 0;

            glGetProgramiv(program, GL_INFO_LOG_LENGTH, &bufLength);

            if (bufLength > 0)
            {
                char* logBuffer = (char*) malloc(bufLength);
```

```
                if (logBuffer != NULL)
                {
                    glGetProgramInfoLog(program, bufLength, NULL, logBuffer);
                    LOGE("Could not link program:\n%s\n", logBuffer);
                    free(logBuffer);
                    logBuffer = NULL;
                }
            }
            glDeleteProgram(program);
            program = 0;
        }
    }
    return program;
}

GLuint glProgram;
GLuint vertexLocation;
GLuint samplerLocation;
GLuint projectionLocation;
GLuint modelViewLocation;
GLuint textureCordLocation;
GLuint textureId;

float projectionMatrix[16];
float modelViewMatrix[16];
float angle = 0;

/* [setupGraphicsUpdate] */
bool setupGraphics(int width, int height)
{
    glProgram = createProgram(glVertexShader, glFragmentShader);

    if (!glProgram)
    {
        LOGE ("Could not create program");
        return false;
    }

    vertexLocation = glGetAttribLocation(glProgram, "vertexPosition");
    textureCordLocation = glGetAttribLocation(glProgram, "vertexTextureCord");
    projectionLocation = glGetUniformLocation(glProgram, "projection");
    modelViewLocation = glGetUniformLocation(glProgram, "modelView");
    samplerLocation = glGetUniformLocation(glProgram, "texture");

    /* Setup the perspective. */
    matrixPerspective(projectionMatrix, 45, (float)width / (float)height, 0.1f, 100);
    glEnable(GL_DEPTH_TEST);

    glViewport(0, 0, width, height);

    /* Load the Texture. */
    textureId = loadSimpleTexture();
    if(textureId == 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

```
/* [setupGraphicsUpdate] */
/* [verticesAndTexture] */
GLfloat cubeVertices[] = {-1.0f,  1.0f, -1.0f, /* Back. */
                           1.0f,  1.0f, -1.0f,
                          -1.0f, -1.0f, -1.0f,
                           1.0f, -1.0f, -1.0f,
                          -1.0f,  1.0f,  1.0f, /* Front. */
                           1.0f,  1.0f,  1.0f,
                          -1.0f, -1.0f,  1.0f,
                           1.0f, -1.0f,  1.0f,
                          -1.0f,  1.0f, -1.0f, /* Left. */
                          -1.0f, -1.0f, -1.0f,
                          -1.0f, -1.0f,  1.0f,
                          -1.0f,  1.0f,  1.0f,
                           1.0f,  1.0f, -1.0f, /* Right. */
                           1.0f, -1.0f, -1.0f,
                           1.0f, -1.0f,  1.0f,
                           1.0f,  1.0f,  1.0f,
                          -1.0f, 1.0f, -1.0f, /* Top. */
                          -1.0f, 1.0f,  1.0f,
                           1.0f, 1.0f,  1.0f,
                           1.0f, 1.0f, -1.0f,
                          -1.0f, - 1.0f, -1.0f, /* Bottom. */
                          -1.0f,  -1.0f,  1.0f,
                           1.0f, - 1.0f,  1.0f,
                           1.0f,  -1.0f, -1.0f
                         };

GLfloat textureCords[] = { 1.0f, 1.0f, /* Back. */
                           0.0f, 1.0f,
                           1.0f, 0.0f,
                           0.0f, 0.0f,
                           0.0f, 1.0f, /* Front. */
                           1.0f, 1.0f,
                           0.0f, 0.0f,
                           1.0f, 0.0f,
                           0.0f, 1.0f, /* Left. */
                           0.0f, 0.0f,
                           1.0f, 0.0f,
                           1.0f, 1.0f,
                           1.0f, 1.0f, /* Right. */
                           1.0f, 0.0f,
                           0.0f, 0.0f,
                           0.0f, 1.0f,
                           0.0f, 1.0f, /* Top. */
                           0.0f, 0.0f,
                           1.0f, 0.0f,
                           1.0f, 1.0f,
                           0.0f, 0.0f, /* Bottom. */
                           0.0f, 1.0f,
                           1.0f, 1.0f,
                           1.0f, 0.0f
};
/* [verticesAndTexture] */

GLushort indicies[] = {0, 3, 2, 0, 1, 3, 4, 6, 7, 4, 7, 5,  8, 9, 10, 8, 11, 10, 12,
13, 14, 15, 12, 14, 16, 17, 18, 16, 19, 18, 20, 21, 22, 20, 23, 22};

void renderFrame()
{
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
```

```
    matrixIdentityFunction(modelViewMatrix);

    matrixRotateX(modelViewMatrix, angle);
    matrixRotateY(modelViewMatrix, angle);

    matrixTranslate(modelViewMatrix, 0.0f, 0.0f, -10.0f);

    glUseProgram(glProgram);
    glVertexAttribPointer(vertexLocation, 3, GL_FLOAT, GL_FALSE, 0, cubeVertices);
    glEnableVertexAttribArray(vertexLocation);

    /* [enableAttributes] */
    glVertexAttribPointer(textureCordLocation, 2, GL_FLOAT, GL_FALSE, 0,
textureCords);
    glEnableVertexAttribArray(textureCordLocation);
    glUniformMatrix4fv(projectionLocation, 1, GL_FALSE,projectionMatrix);
    glUniformMatrix4fv(modelViewLocation, 1, GL_FALSE, modelViewMatrix);

    /* Set the sampler texture unit to 0. */
    glUniform1i(samplerLocation, 0);
    /* [enableAttributes] */
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, indicies);

    angle += 1;
    if (angle > 360)
    {
        angle -= 360;
    }
}

extern "C"
{
    JNIEXPORT void JNICALL
Java_com_arm_malideveloper_openglessdk_texturecube_NativeLibrary_init (JNIEnv * env,
jobject obj, jint width, jint height);
    JNIEXPORT void JNICALL
Java_com_arm_malideveloper_openglessdk_texturecube_NativeLibrary_step(
            JNIEnv * env, jobject obj);
};

JNIEXPORT void JNICALL
Java_com_arm_malideveloper_openglessdk_texturecube_NativeLibrary_init(
        JNIEnv * env, jobject obj, jint width, jint height)
{
    setupGraphics(width, height);
}

JNIEXPORT void JNICALL
Java_com_arm_malideveloper_openglessdk_texturecube_NativeLibrary_step(
        JNIEnv * env, jobject obj)
{
    renderFrame();
}
```

Texture.cpp

```
#include "Texture.h"
```

```
#include <GLES2/gl2ext.h>
#include <cstdio>
#include <cstdlib>

GLuint loadSimpleTexture()
{
    /* Texture Object Handle. */
    GLuint textureId;

    /* 3 x 3 Image,  R G B A Channels RAW Format. */
    GLubyte pixels[9 * 4] =
    {
            18,  140, 171, 255, /* Some Colour Bottom Left. */
            143, 143, 143, 255, /* Some Colour Bottom Middle. */
            255, 255, 255, 255, /* Some Colour Bottom Right. */
            255, 255, 0,   255, /* Yellow Middle Left. */
            0,   255, 255, 255, /* Some Colour Middle. */
            255, 0,   255, 255, /* Some Colour Middle Right. */
            255, 0,   0,   255, /* Red Top Left. */
            0,   255, 0,   255, /* Green Top Middle. */
            0,   0,   255, 255, /* Blue Top Right. */
    };
    /* [includeTextureDefinition] */

    /* [placeTextureInMemory] */
    /* Use tightly packed data. */
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    /* Generate a texture object. */
    glGenTextures(1, &textureId);

    /* Activate a texture. */
    glActiveTexture(GL_TEXTURE0);

    /* Bind the texture object. */
    glBindTexture(GL_TEXTURE_2D, textureId);

    /* Load the texture. */
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 3, 3, 0, GL_RGBA, GL_UNSIGNED_BYTE,
pixels);

    /* Set the filtering mode. */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    return textureId;
}
```