Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет

телекоммуникаций и информатики»

**Лабораторная работа по теме:**

**«Модель Фонга»**

Выполнил:

Студент 4 курса

ИВТ, гр. ИП-712

Алексеев С.В.

Новосибирск 2020 г.

Оглавление

[Задание 2](#_Toc58688895)

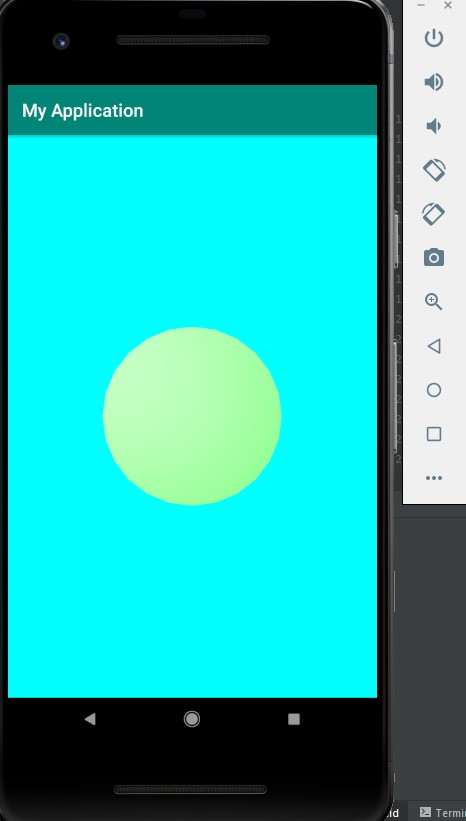
[Скриншоты 3](#_Toc58688896)

[Листинг 3](#_Toc58688897)

# Задание

Создать сферу произвольного цвета, освещенную по модели Фонга.

# Скриншоты



# Листинг

MainActivity.java

package com.example.modelphong;

import androidx.appcompat.app.AppCompatActivity;

import android.opengl.GLSurfaceView;

import android.os.Bundle;

import android.view.WindowManager;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

getWindow().setFlags(WindowManager.LayoutParams.FLAG\_FULLSCREEN, WindowManager.LayoutParams.FLAG\_FULLSCREEN);

GLSurfaceView view = new GLSurfaceView(this);

view.setEGLContextClientVersion(2);

view.setRenderer(new sphereRenderer(this));

setContentView(view);

}

}

Shader.java

package com.example.modelphong;

import android.opengl.GLES20;

import java.nio.FloatBuffer;

public class Shader {

//будем хранить ссылку на шейдерную программу внутри класса как локальное поле

private int program\_Handle;

//при создании объекта класса передаем в конструктор строки кода вершинного и фрагментного шейдера

public Shader(String vertexShaderCode, String fragmentShaderCode) {

//вызываем метод, создающий шейдерную программу при этом заполняется поле program\_Handle

createProgram(vertexShaderCode, fragmentShaderCode);

}

// метод, который создает шейдерную программу, вызывается в конструкторе

private void createProgram(String vertexShaderCode, String fragmentShaderCode) {

int vertexShader\_Handle = GLES20.glCreateShader(GLES20.GL\_VERTEX\_SHADER); // получаем ссылку на вершинный шейдер

GLES20.glShaderSource(vertexShader\_Handle, vertexShaderCode); // присоединяем к вершинному шейдеру его код

GLES20.glCompileShader(vertexShader\_Handle); // компилируем вершинный шейдер

int fragmentShader\_Handle = GLES20.glCreateShader(GLES20.GL\_FRAGMENT\_SHADER); //получаем ссылку на фрагментный шейдер

GLES20.glShaderSource(fragmentShader\_Handle, fragmentShaderCode); //присоединяем к фрагментному шейдеру его код

GLES20.glCompileShader(fragmentShader\_Handle); // компилируем фрагментный шейдер

program\_Handle = GLES20.glCreateProgram(); //получаем ссылку на шейдерную программу

GLES20.glAttachShader(program\_Handle, vertexShader\_Handle); //присоединяем к шейдерной программе вершинный шейдер

GLES20.glAttachShader(program\_Handle, fragmentShader\_Handle); //присоединяем к шейдерной программе фрагментный шейдер

GLES20.glLinkProgram(program\_Handle); //компилируем шейдерную программу

}

// метод, который связывает буфер координат вершин vertexBuffer с атрибутом a\_vertex

public void linkVertexBuffer(FloatBuffer vertexBuffer) {

GLES20.glUseProgram(program\_Handle); //устанавливаем активную программу

int a\_vertex\_Handle = GLES20.glGetAttribLocation(program\_Handle, "a\_vertex"); //получаем ссылку на атрибут a\_vertex

GLES20.glEnableVertexAttribArray(a\_vertex\_Handle); //включаем использование атрибута a\_vertex

//связываем буфер координат вершин vertexBuffer с атрибутом a\_vertex

GLES20.glVertexAttribPointer(a\_vertex\_Handle, 3, GLES20.GL\_FLOAT, false, 0, vertexBuffer);

}

//метод, который связывает буфер координат векторов нормалей normalBuffer с атрибутом a\_normal

public void linkNormalBuffer(FloatBuffer normalBuffer) {

GLES20.glUseProgram(program\_Handle); //устанавливаем активную программу

int a\_normal\_Handle = GLES20.glGetAttribLocation(program\_Handle, "a\_normal"); //получаем ссылку на атрибут a\_normal

GLES20.glEnableVertexAttribArray(a\_normal\_Handle); //включаем использование атрибута a\_normal

//связываем буфер нормалей normalBuffer с атрибутом a\_normal

GLES20.glVertexAttribPointer(a\_normal\_Handle, 3, GLES20.GL\_FLOAT, false, 0, normalBuffer);

}

//метод, который связывает буфер цветов вершин colorBuffer с атрибутом a\_color

public void linkColorBuffer(FloatBuffer colorBuffer) {

GLES20.glUseProgram(program\_Handle); //устанавливаем активную программу

//получаем ссылку на атрибут a\_color

int a\_color\_Handle = GLES20.glGetAttribLocation(program\_Handle, "a\_color");

GLES20.glEnableVertexAttribArray(a\_color\_Handle); //включаем использование атрибута a\_color

//связываем буфер нормалей colorBuffer с атрибутом a\_color

GLES20.glVertexAttribPointer(a\_color\_Handle, 4, GLES20.GL\_FLOAT, false, 0, colorBuffer);

}

// метод, который связывает матрицу модели-вида-проекции modelViewProjectionMatrix с униформой u\_modelViewProjectionMatrix

public void linkModelViewProjectionMatrix(float[] modelViewProjectionMatrix) {

GLES20.glUseProgram(program\_Handle); //устанавливаем активную программу

//получаем ссылку на униформу u\_modelViewProjectionMatrix

int u\_modelViewProjectionMatrix\_Handle = GLES20.glGetUniformLocation(program\_Handle, "u\_modelViewProjectionMatrix");

//связываем массив modelViewProjectionMatrix с униформой u\_modelViewProjectionMatrix

GLES20.glUniformMatrix4fv(u\_modelViewProjectionMatrix\_Handle, 1, false, modelViewProjectionMatrix, 0);

}

// метод, который связывает координаты камеры с униформой u\_camera

public void linkCamera(float xCamera, float yCamera, float zCamera) {

GLES20.glUseProgram(program\_Handle); //устанавливаем активную программу

int u\_camera\_Handle = GLES20.glGetUniformLocation(program\_Handle, "u\_camera"); //получаем ссылку на униформу u\_camera

GLES20.glUniform3f(u\_camera\_Handle, xCamera, yCamera, zCamera); // связываем координаты камеры с униформой u\_camera

}

// метод, который связывает координаты источника света с униформой u\_lightPosition

public void linkLightSource(float xLightPosition, float yLightPosition, float zLightPosition) {

//устанавливаем активную программу

GLES20.glUseProgram(program\_Handle);

//получаем ссылку на униформу u\_lightPosition

int u\_lightPosition\_Handle = GLES20.glGetUniformLocation(program\_Handle, "u\_lightPosition");

// св€зываем координаты источника света с униформой u\_lightPosition

GLES20.glUniform3f(u\_lightPosition\_Handle, xLightPosition, yLightPosition, zLightPosition);

}

// метод, который делает шейдерную программу данного класса активной

public void useProgram() {

GLES20.glUseProgram(program\_Handle);

}

}

Sphere.java

package com.example.modelphong;

import java.nio.ByteBuffer;

import java.nio.ByteOrder;

import java.nio.FloatBuffer;

public class Sphere {

public int count = 0;

public FloatBuffer mVertexBuffer;

public FloatBuffer textureBuffer;

public Sphere(float R) {

int n = 0;

int dtheta = 15, dphi = 15;

float DTOR = (float) (Math.PI / 180.0f);

ByteBuffer byteBuf = ByteBuffer.allocateDirect(5000 \* 3 \* 4); // выделение памяти из основной кучи JVM

byteBuf.order(ByteOrder.nativeOrder()); // извлекает собственный порядок байтов базовой платформы

mVertexBuffer = byteBuf.asFloatBuffer();

byteBuf = ByteBuffer.allocateDirect(5000 \* 2 \* 4);

byteBuf.order(ByteOrder.nativeOrder());

textureBuffer = byteBuf.asFloatBuffer();

for (int theta = -90; theta <= 90 - dtheta; theta += dtheta) {

for (int phi = 0; phi <= 360 - dphi; phi += dphi) {

count++;

mVertexBuffer.put((float) (Math.cos(theta \* DTOR) \* Math.cos(phi \* DTOR)) \* R);

mVertexBuffer.put((float) (Math.cos(theta \* DTOR) \* Math.sin(phi \* DTOR)) \* R);

mVertexBuffer.put((float) (Math.sin(theta \* DTOR)) \* R);

double cosM = Math.cos((theta + dtheta) \* DTOR);

mVertexBuffer.put((float) (cosM \* Math.cos(phi \* DTOR)) \* R);

mVertexBuffer.put((float) (cosM \* Math.sin(phi \* DTOR)) \* R);

double sinM = Math.sin((theta + dtheta) \* DTOR);

mVertexBuffer.put((float) sinM \* R);

mVertexBuffer.put((float) (cosM \* Math.cos((phi + dphi) \* DTOR)) \* R);

mVertexBuffer.put((float) (cosM \* Math.sin((phi + dphi) \* DTOR)) \* R);

mVertexBuffer.put((float) sinM \* R);

mVertexBuffer.put((float) (Math.cos(theta \* DTOR) \* Math.cos((phi + dphi) \* DTOR)) \* R);

mVertexBuffer.put((float) (Math.cos(theta \* DTOR) \* Math.sin((phi + dphi) \* DTOR)) \* R);

mVertexBuffer.put((float) (Math.sin(theta \* DTOR)) \* R);

n += 4;

textureBuffer.put((float) (phi / 360.0f));

textureBuffer.put((float) ((90 + theta) / 180.0f));

textureBuffer.put((float) (phi / 360.0f));

textureBuffer.put((float) ((90 + theta + dtheta) / 180.0f));

textureBuffer.put((float) ((phi + dphi) / 360.0f));

textureBuffer.put((float) ((90 + theta + dtheta) / 180.0f));

textureBuffer.put((float) ((phi + dphi) / 360.0f));

textureBuffer.put((float) ((90 + theta) / 180.0f));

}

}

mVertexBuffer.position(0);

textureBuffer.position(0);

}

}

sphereRenderer.java

package com.example.modelphong;

import android.content.Context;

import android.opengl.GLES20;

import android.opengl.GLSurfaceView;

import android.opengl.Matrix;

import java.nio.ByteBuffer;

import java.nio.ByteOrder;

import java.nio.FloatBuffer;

import javax.microedition.khronos.egl.EGLConfig;

import javax.microedition.khronos.opengles.GL10;

public class sphereRenderer implements GLSurfaceView.Renderer {

private Context context;

private Sphere sphere = new Sphere(0.3f);

private float xСamera, yCamera, zCamera; //координаты камеры

private int n;

private final float[] modelMatrix = new float[16];

private final float[] viewMatrix = new float[16];

private final float[] modelViewMatrix = new float[16];

private final float[] projectionMatrix = new float[16];

private final float[] modelViewProjectionMatrix = new float[16];

private float[] vertexArray;

private float[] normalArray;

static float[] colorArray;

private FloatBuffer vertexBuffer, normalBuffer, colorBuffer;

private Shader shader;

float[] lightDir = {-1.0f, 1.0f, 5.0f}; // расположение блика на сфере

public sphereRenderer(Context context) {

n = sphere.count;

// мы не будем двигать объекты поэтому сбрасываем модельную матрицу на единичную

Matrix.setIdentityM(modelMatrix, 0);

//координаты камеры

xСamera = 0.0f;

yCamera = 0.0f;

zCamera = 1.0f;

// пусть камера смотрит на начало координат и верх у камеры будет вдоль оси Y зная координаты камеры получаем матрицу вида

Matrix.setLookAtM(viewMatrix, 0, xСamera, yCamera, zCamera, 0f, 0f, 0f, 0f, 1f, 0f);

// умножая матрицу вида на матрицу модели получаем матрицу модели-вида:

Matrix.multiplyMM(modelViewMatrix, 0, viewMatrix, 0, modelMatrix, 0);

// массив координат вершин

float[] vertexArray = {

-1f, 1f, 0f,

-1f, -1f, 0f,

1f, 1f, 0f,

1f, -1f, 0f

};

//создадим буфер для хранения координат вершин

ByteBuffer tBuffer = ByteBuffer.allocateDirect(vertexArray.length \* 4);

tBuffer.order(ByteOrder.nativeOrder());

vertexBuffer = tBuffer.asFloatBuffer();

//перепишем координаты вершин из массива в буфер

vertexBuffer.put(vertexArray);

vertexBuffer.position(0);

//вектор нормали перпендикулярен плоскости и направлен вдоль оси Y

//нормаль одинакова для всех вершин

float[] normalArray = new float[n \* 3];

for (int i = 0; i < n \* 3; i++) {

if (i % 3 == 2) normalArray[i] = 1f;

else normalArray[i] = 0f;

}

//создадим буфер для хранения координат векторов нормали

tBuffer = ByteBuffer.allocateDirect(normalArray.length \* 4);

tBuffer.order(ByteOrder.nativeOrder());

normalBuffer = tBuffer.asFloatBuffer();

normalBuffer.put(normalArray);

normalBuffer.position(0);

//перепишем координаты нормалей из массива в буфер

normalBuffer.put(normalArray);

normalBuffer.position(0);

float[] colorArray = new float[n \* 4];

for (int i = 0; i < n \* 3; i++) {

if (i % 4 == 1) colorArray[i] = 0f;

else colorArray[i] = 1f;

}

// буфер для хранения цветов вершин

tBuffer = ByteBuffer.allocateDirect(colorArray.length \* 4);

tBuffer.order(ByteOrder.nativeOrder());

colorBuffer = tBuffer.asFloatBuffer();

colorBuffer.position(0);

// перепишем цвета вершин из массива в буфер

colorBuffer.put(colorArray);

colorBuffer.position(0);

}

// метод, который срабатывает при изменении размеров экрана в нем мы получим матрицу проекции и матрицу модели-вида-проекции

@Override

public void onSurfaceChanged(GL10 gl, int width, int height) {

// устанавливаем glViewport

GLES20.glViewport(0, 0, width, height);

float ratio = (float) width / height;

float k = 0.055f;

float left = -k \* ratio;

float right = k \* ratio;

float bottom = -k;

float top = k;

float near = 0.1f;

float far = 10.0f;

// получаем матрицу проекции

Matrix.frustumM(projectionMatrix, 0, left, right, bottom, top, near, far);

// матрица проекции изменилась, поэтому нужно пересчитать матрицу модели-вида-проекции

Matrix.multiplyMM(modelViewProjectionMatrix, 0, projectionMatrix, 0, modelViewMatrix, 0);

}

// метод, который срабатывает при создании экрана здесь мы создаем шейдерный объект

@Override

public void onSurfaceCreated(GL10 gl, EGLConfig config) {

GLES20.glClearColor(0.0f, 1.0f, 1.0f, 0.0f);

//включаем тест глубины

GLES20.glEnable(GLES20.GL\_DEPTH\_TEST);

//включаем отсечение невидимых граней

GLES20.glEnable(GLES20.GL\_CULL\_FACE);

//включаем сглаживание текстур

GLES20.glHint(GLES20.GL\_GENERATE\_MIPMAP\_HINT, GLES20.GL\_NICEST);

// записываем код вершинного шейдера в виде строки

String vertexShaderCode =

"uniform mat4 u\_modelViewProjectionMatrix;" +

"attribute vec3 a\_vertex;" +

"attribute vec3 a\_normal;" +

"attribute vec4 a\_color;" +

"varying vec3 v\_vertex;" +

"varying vec3 v\_normal;" +

"varying vec4 v\_color;" +

"void main() {" +

"v\_vertex=a\_vertex;" +

"vec3 n\_normal=normalize(a\_normal);" +

"v\_normal=n\_normal;" +

"v\_color=a\_color;" +

"gl\_Position = u\_modelViewProjectionMatrix \* vec4(a\_vertex,1.0);" +

"}";

//записываем код фрагментного шейдера в виде строки

String fragmentShaderCode =

"precision mediump float;" +

"uniform vec3 u\_camera;" +

"uniform vec3 u\_lightPosition;" +

"varying vec3 v\_vertex;" +

"varying vec3 v\_normal;" +

"varying vec4 v\_color;" +

"void main() {" +

"vec3 n\_normal=normalize(v\_normal);" +

"vec3 lightvector = normalize(u\_lightPosition - v\_vertex);" +

"vec3 lookvector = normalize(u\_camera - v\_vertex);" +

"float ambient=0.2;" +

"float k\_diffuse=0.8;" +

"float k\_specular=0.4;" +

"float diffuse = k\_diffuse \* max(dot(n\_normal, lightvector), 0.0);" +

"vec3 reflectvector = reflect(-lightvector, n\_normal);" +

"float specular = k\_specular \* pow( max(dot(lookvector,reflectvector),0.0), 40.0 );" +

"vec4 one=vec4(1.0,5.0,1.0,1.0);" +

"gl\_FragColor = (ambient+diffuse+specular)\*one;" +

"}";

//создадим шейдерный объект

shader = new Shader(vertexShaderCode, fragmentShaderCode);

//свяжем буфер вершин с атрибутом a\_vertex в вершинном шейдере

shader.linkVertexBuffer(vertexBuffer);

//свяжем буфер нормалей с атрибутом a\_normal в вершинном шейдере

shader.linkNormalBuffer(normalBuffer);

//свяжем буфер цветов с атрибутом a\_color в вершинном шейдере

shader.linkColorBuffer(colorBuffer);

//связь атрибутов с буферами сохраняется до тех пор, пока не будет уничтожен шейдерный объект

}

@Override

public void onDrawFrame(GL10 gl) {

GLES20.glClear(GLES20.GL\_COLOR\_BUFFER\_BIT | GLES20.GL\_DEPTH\_BUFFER\_BIT); //очищаем кадр

GLES20.glEnable(GL10.GL\_BLEND);

shader.linkVertexBuffer(sphere.mVertexBuffer);

shader.linkColorBuffer(colorBuffer); //связь буфера цветов с атрибутом a\_color в вершинном шейдере

//в отличие от атрибутов связь униформ с внешними параметрами не сохраняется, поэтому перед рисованием каждого кадра

//нужно связывать униформы заново передаем в шейдерный объект матрицу модели-вида-проекции

shader.linkModelViewProjectionMatrix(modelViewProjectionMatrix);

shader.linkCamera(xСamera, yCamera, zCamera); //передаем в шейдерный объект координаты камеры

shader.linkLightSource(-0.6f, 0.4f, 0.3f); //передаем в шейдерный объект координаты источника света

shader.useProgram();

for (int i = 0; i < n - 3; i += 4) {

GLES20.glDrawArrays(GLES20.GL\_TRIANGLE\_FAN, i, 4);

}

GLES20.glDisable(GL10.GL\_BLEND);

}

}