Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет

телекоммуникаций и информатики»

Лабораторная работа по теме: «Модель Фонга»

Выполнили:

студентки 4 курса

ИВТ, гр. ИП-712

Гервас А.В.

Онищенко А.В.

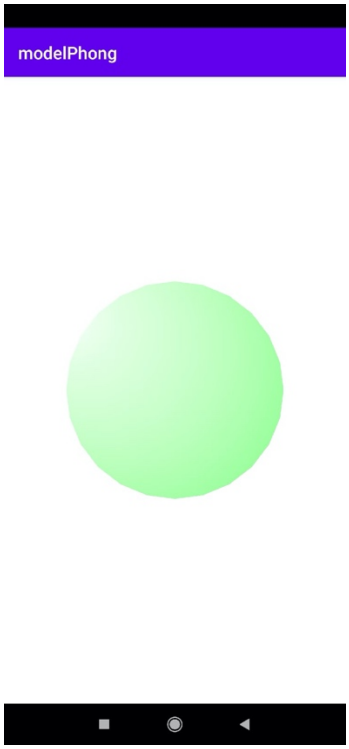
Новосибирск 2020

Оглавление

Задание

Создать сферу произвольного цвета, освещенную по модели Фонга.

Скриншоты



Листинг кода

Приложение написано на языке Java.

MainActivity.java

```
package ru.sibsutis.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);
    }
}
```

sphereRenderer.java

```
package ru.sibsutis.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(0.5f);
    private float xCamera, 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
        xCamera = 0.0f;
        yCamera = 0.0f;
        zCamera = 1.5f;

        // пусть камера смотрит на начало координат и верх у камеры будет
вдоль оси Y зная координаты камеры получаем матрицу вида
        Matrix.setLookAtM(viewMatrix, 0, xCamera, 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(1.0f, 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(xCamera, 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);
    }
}
```

Sphere.java

```
package ru.sibsutis.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);
   }
}
```

Shader.java

```
package ru.sibsutis.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 св€зывает 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);
    }
}
```