Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра прикладной математики и кибернетики

Отчёт
по курсовой работе
«Приложение с использованием OpenGL ES 2.0»

Выполнил:

студент группы ИП-114

Флат Е.В
ФИО студента

Работу проверил: Осипова У.В.
ФИО преподавателя

Новосибирск 2024 г.

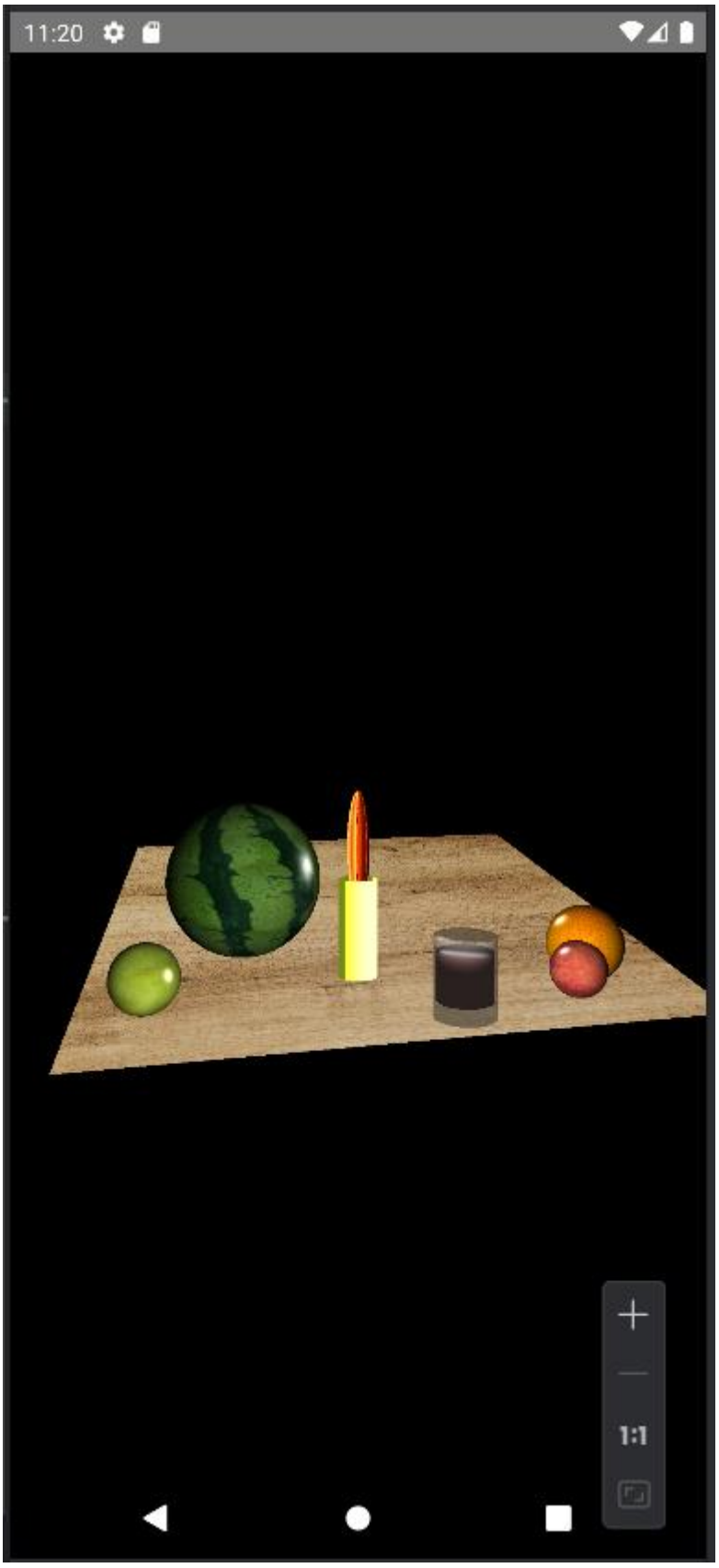
Содержание

Задание

Создайте программу, в которой нарисован стол на OpenGL ES 2.0.

На столе лежат различные фрукты/овощи (не менее 4 различных), стакан с напитком. Имеется свеча, дающее освещение (по модели Фонга), пламя динамически двигается волной.

Результаты работы программы



Листинг программы

MainActivity.kt

```
package com.example.course

import android.content.Context
import android.opengl.GLSurfaceView
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.runtime.Composable
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.viewinterop.AndroidView

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            OpenGLView(LocalContext.current)
        }
    }
    @Composable
    fun OpenGLView(context: Context) {
        AndroidView(factory = {
            GLSurfaceView(context).apply {
                setEGLContextClientVersion(2)
                setRenderer(Renderer(context))
                renderMode = GLSurfaceView.RENDERMODE_CONTINUOUSLY
            }
        })
    }
}
```

Renderer.kt

```
package com.example.course

import android.content.Context
import android.graphics.BitmapFactory
import android.opengl.GLES20
import android.opengl.GLSurfaceView
import android.opengl.GLUtils
import android.opengl.Matrix
import javax.microedition.khronos.egl.EGLConfig
import javax.microedition.khronos.opengles.GL10

class Renderer(private val context: Context) : GLSurfaceView.Renderer {

    private val projectionMatrix = FloatArray(16)
    private val viewMatrix = FloatArray(16)
    private val modelMatrix = FloatArray(16)
    private val mVPMatrix = FloatArray(16)
    private val normalMatrix = FloatArray(16)
    private val lightPos = floatArrayOf(0f, -0.2f, 2.8f)

    private lateinit var table: Table
    private lateinit var glass: Glass
    private lateinit var candle: Candle
    private lateinit var candleFire: CandleFire
    private lateinit var apple: Sphere
    private lateinit var watermelon: Sphere
    private lateinit var orange: Sphere
    private lateinit var granat: Sphere

    private var appleTexture: Int = 0
    private var watermelonTexture: Int = 0
    private var orangeTexture: Int = 0
    private var granatTexture: Int = 0
    private var candleFireTexture: Int = 0

    override fun onSurfaceCreated(arg0: GL10?, arg1: EGLConfig?) {
        GLES20.glClearColor(0f,0f,0f, 0.0f)
        GLES20.glEnable(GLES20.GL_DEPTH_TEST)
        GLES20.glDepthFunc(GLES20.GL_LESS)

        GLES20.glEnable(GLES20.GL_BLEND)
        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,
GLES20.GL_ONE_MINUS_SRC_ALPHA)



        table = Table(context)

        apple = Sphere(radius = 0.3f/3)
        appleTexture = loadTexture(R.drawable.apple)

        glass = Glass(context)

        candle = Candle(context)
        candleFire = CandleFire(context, radius = 0.035f)
        candleFire.initialize()
        //candleFireTexture = loadTexture(R.drawable.fire1.png)
```

```
        watermelon = Sphere(radius = 0.8f/3)
        watermelonTexture = loadTexture(R.drawable.watermelon)

        orange = Sphere(radius = 0.35f/3)
        orangeTexture = loadTexture(R.drawable.orange)

        granat = Sphere(radius = 0.25f/3)
        granatTexture = loadTexture(R.drawable.granat)
    }

    override fun onDrawFrame(arg0: GL10?) {
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT or
GLES20.GL_DEPTH_BUFFER_BIT)

        //установление единичной матрицы (дефолт позиция, масштаб, поворот)
        Matrix.setIdentityM(modelMatrix, 0)
        //точка обзора
        Matrix.setLookAtM(viewMatrix, 0, 0f, 0f, 5f, 0f, 0f, 0f, 0f, 1f, 0f)
        //преобразование из 3D в 2D
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        //инверсия, чтоб правильно вычислить нормали
        Matrix.invertM(normalMatrix, 0, viewMatrix, 0)
        //перемещение вдоль осей
        Matrix.translateM(modelMatrix, 0, 0f, -1.3f, -1f)
        //поворот
        Matrix.rotateM(modelMatrix, 0, 15f, 0.1f, 0.1f, 0f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        //итоговая матрица для рендеринга
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        Matrix.setIdentityM(modelMatrix, 0)
        table.draw(mVPMatrix)

        Matrix.setIdentityM(modelMatrix, 0)
        //Matrix.setLookAtM(viewMatrix, 0, 0f, 0f, 5f, 0f, 0f, 0f, 0f, 1f,
0f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        //Matrix.invertM(normalMatrix, 0, viewMatrix, 0)
        Matrix.translateM(modelMatrix, 0, -0.6f, -0.6f, 3f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        apple.draw(mVPMatrix, normalMatrix, lightPos, viewMatrix,
appleTexture)

        Matrix.setIdentityM(modelMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.translateM(modelMatrix, 0, -0.4f, -0.4f, 2.5f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        watermelon.draw(mVPMatrix, normalMatrix, lightPos, viewMatrix,
watermelonTexture)

        Matrix.setIdentityM(modelMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.translateM(modelMatrix, 0, 0.65f, -0.6f, 2.9f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
```

```
        granat.draw(mVPMatrix, normalMatrix, lightPos, viewMatrix,
granatTexture)

        Matrix.setIdentityM(modelMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.translateM(modelMatrix, 0, 0.7f, -0.55f, 2.8f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        orange.draw(mVPMatrix, normalMatrix, lightPos, viewMatrix,
orangeTexture)

        Matrix.setIdentityM(modelMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.translateM(modelMatrix, 0, 0.33f, -0.65f, 2.8f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        glass.draw(mVPMatrix, lightPos, viewMatrix)

        Matrix.setIdentityM(modelMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.translateM(modelMatrix, 0, 0f, -0.5f, 2.8f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        candle.draw(mVPMatrix, lightPos, viewMatrix)


        val time = System.currentTimeMillis() % 10000L / 1000.0f // Время в
секундах

        Matrix.setIdentityM(modelMatrix, 0)
        Matrix.translateM(modelMatrix, 0, lightPos[0], lightPos[1],
lightPos[2])
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        candleFire.draw(mVPMatrix, time)
    }

    override fun onSurfaceChanged(arg0: GL10?, width: Int, height: Int) {
        GLES20.glViewport(0, 0, width, height)
        val ratio: Float = width.toFloat() / height.toFloat()
        Matrix.frustumM(projectionMatrix, 0, -ratio, ratio, -1f, 1f, 1f, 10f)
        Matrix.setLookAtM(viewMatrix, 0, 0f, 0f, 5f, 0f, 0f, 0f, 0f, 1f, 0f)
    }

    private fun loadTexture(resourceId: Int): Int {
        val textureIds = IntArray(1)
        GLES20.glGenTextures(1, textureIds, 0)
        val textureId = textureIds[0]

        val bitmap = BitmapFactory.decodeResource(context.resources,
resourceId)

        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId)
        GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR)
        GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR)

        GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0)
```

```
            bitmap.recycle()

            return textureId
        }
    }
```

Table.kt

```
package com.example.course

import android.content.Context
import android.graphics.BitmapFactory
import android.opengl.GLES20
import android.opengl.GLUtils
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.nio.FloatBuffer
import java.nio.ShortBuffer

class Table(private val context: Context) {

    private val vertexBuffer: FloatBuffer
    private val texCoordBuffer: FloatBuffer
    private val indexBuffer: ShortBuffer
    private val normalBuffer: FloatBuffer
    private var program: Int
    private var textureId: Int = 0

    private val vertices = floatArrayOf(
        // Столешница
        -2f, 0.1f, 2f,    // 0: передняя левая верх
        -2f, 0.1f, -2f,   // 1: задняя левая верх
        2f, 0.1f, -2f,    // 2: задняя правая верх
        2f, 0.1f, 2f,     // 3: передняя правая верх
        -2f, 0f, 2f,      // 4: передняя левая низ
        -2f, 0f, -2f,     // 5: задняя левая низ
        2f, 0f, -2f,      // 6: задняя правая низ
        2f, 0f, 2f        // 7: передняя правая низ
    )

    private val normals = floatArrayOf(
        // Нормали для каждого вершины
        0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f,
        0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f
    )

    private val texCoords = floatArrayOf(
        0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
        0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f
    )

    private val indices = shortArrayOf(
        0, 1, 2, 0, 2, 3, // Верх столешницы

    )

    init {
        // Создаем буферы
        vertexBuffer = ByteBuffer.allocateDirect(vertices.size * 4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer().apply {
                put(vertices)
                position(0)
            }

        normalBuffer = ByteBuffer.allocateDirect(normals.size * 4)
```

```
            .order(ByteOrder.nativeOrder()).asFloatBuffer().apply {
                put(normals)
                position(0)
            }

        texCoordBuffer = ByteBuffer.allocateDirect(texCoords.size * 4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer().apply {
                put(texCoords)
                position(0)
            }

        indexBuffer = ByteBuffer.allocateDirect(indices.size * 2)
            .order(ByteOrder.nativeOrder()).asShortBuffer().apply {
                put(indices)
                position(0)
            }

        // Компиляция и линковка шейдеров
        val vertexShader = loadShader(GLES20.GL_VERTEX_SHADER,
VERTEX_SHADER_CODE)
        val fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER,
FRAGMENT_SHADER_CODE)
        program = GLES20.glCreateProgram().apply {
            GLES20.glAttachShader(this, vertexShader)
            GLES20.glAttachShader(this, fragmentShader)
            GLES20.glLinkProgram(this)
        }

        // Загрузка текстуры
        textureId = loadTexture(context, R.drawable.wood_texture)
    }

    private fun loadTexture(context: Context, resourceId: Int): Int {
        val textureIds = IntArray(1)
        GLES20.glGenTextures(1, textureIds, 0)
        val bitmap = BitmapFactory.decodeResource(context.resources,
resourceId)
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureIds[0])
        GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR)
        GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR)
        GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0)
        bitmap.recycle()
        return textureIds[0]
    }

    fun draw(mVPMatrix: FloatArray) {
        GLES20.glUseProgram(program)

        // Привязка атрибутов и униформов
        val positionHandle = GLES20.glGetAttribLocation(program, "vPosition")
        GLES20.glEnableVertexAttribArray(positionHandle)
        GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT,
false, 12, vertexBuffer)

        val texCoordHandle = GLES20.glGetAttribLocation(program, "aTexCoord")
        GLES20.glEnableVertexAttribArray(texCoordHandle)
```

```
        GLES20.glVertexAttribPointer(texCoordHandle, 2, GLES20.GL_FLOAT,
false, 8, texCoordBuffer)

        val matrixHandle = GLES20.glGetUniformLocation(program, "uMVPMatrix")
        GLES20.glUniformMatrix4fv(matrixHandle, 1, false, mVPMatrix, 0)

        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId)
        GLES20.glDrawElements(GLES20.GL_TRIANGLES, indices.size,
GLES20.GL_UNSIGNED_SHORT, indexBuffer)

        GLES20.glDisableVertexAttribArray(positionHandle)
        GLES20.glDisableVertexAttribArray(texCoordHandle)
    }

    private fun loadShader(type: Int, shaderCode: String): Int {
        return GLES20.glCreateShader(type).also { shader ->
            GLES20.glShaderSource(shader, shaderCode)
            GLES20.glCompileShader(shader)
        }
    }

    companion object {
        private const val VERTEX_SHADER_CODE = """
            uniform mat4 uMVPMatrix;
            attribute vec4 vPosition;
            attribute vec2 aTexCoord;
            varying vec2 vTexCoord;

            void main() {
                gl_Position = uMVPMatrix * vPosition;
                vTexCoord = aTexCoord;
            }
        """

        private const val FRAGMENT_SHADER_CODE = """
            precision mediump float;
            varying vec2 vTexCoord;
            uniform sampler2D uTexture;

            void main() {
                gl_FragColor = texture2D(uTexture, vTexCoord);
            }
        """
    }
}
```

Glass.kt

```
package com.example.course

import android.content.Context
import android.opengl.GLES20
import android.opengl.Matrix
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.nio.FloatBuffer

class Glass(private val context: Context) {

    private val vertexBufferGlass: FloatBuffer
    private val colorBufferGlass: FloatBuffer
    private val vertexBufferLiquid: FloatBuffer
    private val colorBufferLiquid: FloatBuffer
    private val normalBufferGlass: FloatBuffer
    private val normalBufferLiquid: FloatBuffer

    private var program: Int
    private val modelMatrix = FloatArray(16)

    // Вершины стакана (цилиндр) и жидкости (чуть меньше стакана)
    private val cylinderVertices: FloatArray = generateCylinderVertices(0.1f,
0.25f, 30)
    private val liquidVertices: FloatArray = generateCylinderVertices(0.09f,
0.15f, 30)

    // Нормали для стакана и жидкости
    private val cylinderNormals: FloatArray = generateCylinderNormals(0.1f,
0.25f, 30)
    private val liquidNormals: FloatArray = generateCylinderNormals(0.09f,
0.15f, 30)

    // Цвета для стакана (полупрозрачные)
    private val cylinderColors: FloatArray = FloatArray(cylinderVertices.size
/ 3 * 4).apply {
        for (i in indices step 4) {
            this[i] = 0.7f  // Красный компонент
            this[i + 1] = 0.7f  // Зеленый компонент
            this[i + 2] = 0.7f  // Синий компонент
            this[i + 3] = 0.5f  // Прозрачность (0.5f для полупрозрачности)
        }
    }

    // Цвета для жидкости (например, коричневый напиток)
    private val liquidColors: FloatArray = FloatArray(liquidVertices.size / 3
* 4).apply {
        for (i in indices step 4) {
            this[i] = 0.6f  // Красный компонент
            this[i + 1] = 0.3f  // Зеленый компонент
            this[i + 2] = 0.3f  // Синий компонент
            this[i + 3] = 0.9f
        }
    }

    init {
        // Инициализация буферов для стакана
```

```
        vertexBufferGlass = ByteBuffer.allocateDirect(cylinderVertices.size *
4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        vertexBufferGlass.put(cylinderVertices).position(0)

        colorBufferGlass = ByteBuffer.allocateDirect(cylinderColors.size * 4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        colorBufferGlass.put(cylinderColors).position(0)

        // Инициализация буферов для жидкости
        vertexBufferLiquid = ByteBuffer.allocateDirect(liquidVertices.size *
4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        vertexBufferLiquid.put(liquidVertices).position(0)

        colorBufferLiquid = ByteBuffer.allocateDirect(liquidColors.size * 4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        colorBufferLiquid.put(liquidColors).position(0)

        // Инициализация буферов для нормалей
        normalBufferGlass = ByteBuffer.allocateDirect(cylinderNormals.size *
4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        normalBufferGlass.put(cylinderNormals).position(0)

        normalBufferLiquid = ByteBuffer.allocateDirect(liquidNormals.size *
4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        normalBufferLiquid.put(liquidNormals).position(0)

        // Компиляция и линковка шейдеров
        val vertexShader = loadShader(GLES20.GL_VERTEX_SHADER,
vertexShaderCode)
        val fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER,
fragmentShaderCode)
        program = GLES20.glCreateProgram().apply {
            GLES20.glAttachShader(this, vertexShader)
            GLES20.glAttachShader(this, fragmentShader)
            GLES20.glLinkProgram(this)
        }
    }

    fun draw(mVPMatrix: FloatArray, lightPos: FloatArray, viewPos:
FloatArray) {
        // Включение смешивания
        GLES20.glEnable(GLES20.GL_BLEND)
        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,
GLES20.GL_ONE_MINUS_SRC_ALPHA)

        GLES20.glUseProgram(program)

        // Отрисовка жидкости внутри стакана
        drawObject(vertexBufferLiquid, colorBufferLiquid, normalBufferLiquid,
mVPMatrix, lightPos, viewPos)

        // Отрисовка стакана (полупрозрачного)
        drawObject(vertexBufferGlass, colorBufferGlass, normalBufferGlass,
mVPMatrix, lightPos, viewPos)
```

```
        // Отключение смешивания после отрисовки
        GLES20.glDisable(GLES20.GL_BLEND)
    }

    private fun drawObject(vertexBuffer: FloatBuffer, colorBuffer:
FloatBuffer, normalBuffer: FloatBuffer, mVPMatrix: FloatArray, lightPos:
FloatArray, viewPos: FloatArray) {
        // Применение трансформаций
        Matrix.setIdentityM(modelMatrix, 0)
        val finalMatrix = FloatArray(16)
        Matrix.multiplyMM(finalMatrix, 0, mVPMatrix, 0, modelMatrix, 0)

        // Связка атрибутов вершин
        val positionHandle = GLES20.glGetAttribLocation(program, "vPosition")
        GLES20.glEnableVertexAttribArray(positionHandle)
        GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT,
false, 12, vertexBuffer)

        // Связка атрибутов цветов
        val colorHandle = GLES20.glGetAttribLocation(program, "vColor")
        GLES20.glEnableVertexAttribArray(colorHandle)
        GLES20.glVertexAttribPointer(colorHandle, 4, GLES20.GL_FLOAT, false,
16, colorBuffer)

        // Связка атрибутов нормалей
        val normalHandle = GLES20.glGetAttribLocation(program, "a_Normal")
        GLES20.glEnableVertexAttribArray(normalHandle)
        GLES20.glVertexAttribPointer(normalHandle, 3, GLES20.GL_FLOAT, false,
12, normalBuffer)

        // Связка матрицы трансформации
        val matrixHandle = GLES20.glGetUniformLocation(program, "uMVPMatrix")
        GLES20.glUniformMatrix4fv(matrixHandle, 1, false, finalMatrix, 0)

        // Связка для света и камеры
        val lightPosHandle = GLES20.glGetUniformLocation(program,
"u_LightPos")
        val viewPosHandle = GLES20.glGetUniformLocation(program, "u_ViewPos")
        GLES20.glUniform3fv(lightPosHandle, 1, lightPos, 0)
        GLES20.glUniform3fv(viewPosHandle, 1, viewPos, 0)

        // Отрисовка цилиндра как треугольных полос
        GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, vertexBuffer.limit()
/ 3)

        // Отключение атрибутов
        GLES20.glDisableVertexAttribArray(positionHandle)
        GLES20.glDisableVertexAttribArray(colorHandle)
        GLES20.glDisableVertexAttribArray(normalHandle)
    }

    // Генерация вершин цилиндра
    private fun generateCylinderVertices(radius: Float, height: Float,
segments: Int): FloatArray {
        val vertices = ArrayList<Float>()
        val angleStep = (2 * Math.PI / segments).toFloat()

        for (i in 0..segments) {
            val angle = i * angleStep
```

```
            val x = (radius * Math.cos(angle.toDouble())).toFloat()
            val z = (radius * Math.sin(angle.toDouble())).toFloat()

            // Верхнее основание
            vertices.add(x)
            vertices.add(height / 2)
            vertices.add(z)

            // Нижнее основание
            vertices.add(x)
            vertices.add(-height / 2)
            vertices.add(z)
        }

        return vertices.toFloatArray()
    }

    // Генерация нормалей для цилиндра
    private fun generateCylinderNormals(radius: Float, height: Float,
segments: Int): FloatArray {
        val normals = ArrayList<Float>()
        val angleStep = (2 * Math.PI / segments).toFloat()

        for (i in 0..segments) {
            val angle = i * angleStep
            val x = (radius * Math.cos(angle.toDouble())).toFloat()
            val z = (radius * Math.sin(angle.toDouble())).toFloat()

            // Нормали для цилиндра
            normals.add(x)
            normals.add(0.0f) // Нормали по Y для боковой поверхности
            normals.add(z)

            // Нормали для дна стакана
            normals.add(0.0f)
            normals.add(-1.0f)
            normals.add(0.0f)
        }

        return normals.toFloatArray()
    }

    // Компиляция шейдера
    private fun loadShader(type: Int, shaderCode: String): Int {
        return GLES20.glCreateShader(type).also { shader ->
            GLES20.glShaderSource(shader, shaderCode)
            GLES20.glCompileShader(shader)
        }
    }

    companion object {
        // Вершинный шейдер
        private const val vertexShaderCode =
            """
            uniform mat4 uMVPMatrix;
            attribute vec4 vPosition;
            attribute vec4 vColor;
            attribute vec3 a_Normal;
            varying vec4 outColor;
```

```
            varying vec3 v_Normal;
            varying vec3 v_Position;

            void main() {
                gl_Position = uMVPMatrix * vPosition;
                v_Position = vec3(gl_Position); // Позиция в мировых
координатах
                v_Normal = a_Normal; // Нормали
                outColor = vColor;
            }
            """

        // Фрагментный шейдер
        private const val fragmentShaderCode =
            """
            precision mediump float;
            varying vec4 outColor;
            varying vec3 v_Normal;
            varying vec3 v_Position;

            uniform vec3 u_LightPos;
            uniform vec3 u_ViewPos;

            void main() {
                // Ambient
                vec3 ambient = 0.2 * outColor.rgb;

                // Diffuse
                vec3 norm = normalize(v_Normal);
                vec3 lightDir = normalize(u_LightPos - v_Position);
                float diff = max(dot(norm, lightDir), 0.0);
                vec3 diffuse = diff * outColor.rgb;

                // Specular
                vec3 viewDir = normalize(u_ViewPos - v_Position);
                vec3 reflectDir = reflect(-lightDir, norm);
                float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
                vec3 specular = vec3(1.0) * spec; // White specular color

                // Финальный цвет
                vec3 finalColor = ambient + diffuse + specular;
                gl_FragColor = vec4(finalColor, outColor.a);
            }
            """
    }
}
```

Candle.kt

```
package com.example.course

import android.content.Context
import android.opengl.GLES20
import android.opengl.Matrix
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.nio.FloatBuffer

class Candle(private val context: Context) {

    private val vertexBufferLiquid: FloatBuffer
    private val colorBufferLiquid: FloatBuffer
    private val normalBuffer: FloatBuffer

    private var program: Int
    private val modelMatrix = FloatArray(16)

    private val liquidVertices: FloatArray = generateCylinderVertices(0.06f,
0.3f, 30)
    private val liquidNormals: FloatArray = generateCylinderNormals(0.06f,
0.3f, 30)


    private val liquidColors: FloatArray = FloatArray(liquidVertices.size / 3
* 4).apply {
        for (i in indices step 4) {
            this[i] = 4.6f  // Красный компонент
            this[i + 1] = 5.3f  // Зеленый компонент
            this[i + 2] = 0.9f  // Синий компонент
            this[i + 3] = 7.9f
        }
    }

    init {
        // Инициализация буферов для жидкости
        vertexBufferLiquid = ByteBuffer.allocateDirect(liquidVertices.size *
4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        vertexBufferLiquid.put(liquidVertices).position(0)

        colorBufferLiquid = ByteBuffer.allocateDirect(liquidColors.size * 4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        colorBufferLiquid.put(liquidColors).position(0)

        normalBuffer = ByteBuffer.allocateDirect(liquidNormals.size * 4)
            .order(ByteOrder.nativeOrder()).asFloatBuffer()
        normalBuffer.put(liquidNormals).position(0)

        val vertexShader = loadShader(GLES20.GL_VERTEX_SHADER,
vertexShaderCode)
        val fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER,
fragmentShaderCode)
        program = GLES20.glCreateProgram().apply {
            GLES20.glAttachShader(this, vertexShader)
            GLES20.glAttachShader(this, fragmentShader)
            GLES20.glLinkProgram(this)
```

```
        }
    }

    fun draw(mVPMatrix: FloatArray, lightPos: FloatArray, viewPos:
FloatArray) {
        GLES20.glEnable(GLES20.GL_BLEND)
        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,
GLES20.GL_ONE_MINUS_SRC_ALPHA)

        GLES20.glUseProgram(program)

        drawObject(vertexBufferLiquid, colorBufferLiquid, normalBuffer,
mVPMatrix, lightPos, viewPos)

        GLES20.glDisable(GLES20.GL_BLEND)
    }

    private fun drawObject(vertexBuffer: FloatBuffer, colorBuffer:
FloatBuffer, normalBuffer: FloatBuffer, mVPMatrix: FloatArray, lightPos:
FloatArray, viewPos: FloatArray) {
        // Применение трансформаций
        Matrix.setIdentityM(modelMatrix, 0)
        val finalMatrix = FloatArray(16)
        Matrix.multiplyMM(finalMatrix, 0, mVPMatrix, 0, modelMatrix, 0)

        // Связка атрибутов вершин
        val positionHandle = GLES20.glGetAttribLocation(program, "vPosition")
        GLES20.glEnableVertexAttribArray(positionHandle)
        GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT,
false, 12, vertexBuffer)

        // Связка атрибутов цветов
        val colorHandle = GLES20.glGetAttribLocation(program, "vColor")
        GLES20.glEnableVertexAttribArray(colorHandle)
        GLES20.glVertexAttribPointer(colorHandle, 4, GLES20.GL_FLOAT, false,
16, colorBuffer)

        // Связка атрибутов нормалей
        val normalHandle = GLES20.glGetAttribLocation(program, "a_Normal")
        GLES20.glEnableVertexAttribArray(normalHandle)
        GLES20.glVertexAttribPointer(normalHandle, 3, GLES20.GL_FLOAT, false,
12, normalBuffer)

        // Связка матрицы трансформации
        val matrixHandle = GLES20.glGetUniformLocation(program, "uMVPMatrix")
        GLES20.glUniformMatrix4fv(matrixHandle, 1, false, finalMatrix, 0)

        // Связка для света и камеры
        val lightPosHandle = GLES20.glGetUniformLocation(program,
"u_LightPos")
        val viewPosHandle = GLES20.glGetUniformLocation(program, "u_ViewPos")
        GLES20.glUniform3fv(lightPosHandle, 1, lightPos, 0)
        GLES20.glUniform3fv(viewPosHandle, 1, viewPos, 0)

        // Отрисовка цилиндра как треугольных полос
        GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, vertexBuffer.limit()
/ 3)

        // Отключение атрибутов
```

```
        GLES20.glDisableVertexAttribArray(positionHandle)
        GLES20.glDisableVertexAttribArray(colorHandle)
        GLES20.glDisableVertexAttribArray(normalHandle)
    }

    // Генерация вершин цилиндра
    private fun generateCylinderVertices(radius: Float, height: Float,
segments: Int): FloatArray {
        val vertices = ArrayList<Float>()
        val angleStep = (2 * Math.PI / segments).toFloat()

        for (i in 0..segments) {
            val angle = i * angleStep
            val x = (radius * Math.cos(angle.toDouble())).toFloat()
            val z = (radius * Math.sin(angle.toDouble())).toFloat()

            // Верхнее основание
            vertices.add(x)
            vertices.add(height / 2)
            vertices.add(z)

            // Нижнее основание
            vertices.add(x)
            vertices.add(-height / 2)
            vertices.add(z)
        }

        return vertices.toFloatArray()
    }

    // Генерация нормалей цилиндра
    private fun generateCylinderNormals(radius: Float, height: Float,
segments: Int): FloatArray {
        val normals = ArrayList<Float>()
        val angleStep = (2 * Math.PI / segments).toFloat()

        for (i in 0..segments) {
            val angle = i * angleStep
            val x = (radius * Math.cos(angle.toDouble())).toFloat()
            val z = (radius * Math.sin(angle.toDouble())).toFloat()

            // Нормали направлены наружу
            normals.add(x)
            normals.add(0f) // Нормаль по Y - цилиндр, поэтому 0
            normals.add(z)

            normals.add(x)
            normals.add(0f) // Нормаль по Y - цилиндр, поэтому 0
            normals.add(z)
        }

        return normals.toFloatArray()
    }

    // Компиляция шейдера
    private fun loadShader(type: Int, shaderCode: String): Int {
        return GLES20.glCreateShader(type).also { shader ->
            GLES20.glShaderSource(shader, shaderCode)
            GLES20.glCompileShader(shader)
```

```
        }
    }

    companion object {
        // Вершинный шейдер
        private const val vertexShaderCode =
            """
            uniform mat4 uMVPMatrix;
            attribute vec4 vPosition;
            attribute vec4 vColor;
            attribute vec3 a_Normal;
            varying vec4 outColor;
            varying vec3 v_Normal;
            varying vec3 v_LightDir;
            varying vec3 v_ViewDir;

            void main() {
                gl_Position = uMVPMatrix * vPosition;
                outColor = vColor;

                // Нормали для освещения
                v_Normal = normalize(a_Normal);
                v_LightDir = normalize(vec3(1.0, 1.0, 1.0)); // Направление
света (можно изменить)
                v_ViewDir = normalize(-vec3(gl_Position)); // Направление
взгляда
            }
            """

        // Фрагментный шейдер
        private const val fragmentShaderCode =
            """
            precision mediump float;
            varying vec4 outColor;
            varying vec3 v_Normal;
            varying vec3 v_LightDir;
            varying vec3 v_ViewDir;

            void main() {
                // Основное освещение
                vec3 norm = normalize(v_Normal);
                vec3 lightDir = normalize(v_LightDir);
                vec3 viewDir = normalize(v_ViewDir);

                // Ambient Light
                vec3 ambient = 0.1 * outColor.rgb;

                // Diffuse Light
                float diff = max(dot(norm, lightDir), 0.0);
                vec3 diffuse = diff * outColor.rgb;

                // Specular Light
                vec3 reflectDir = reflect(-lightDir, norm);
                float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
                vec3 specular = spec * vec3(1.0); // Цвет блики

                // Финальный цвет
                vec3 finalColor = ambient + diffuse + specular;
                gl_FragColor = vec4(finalColor, outColor.a);
```

```
                }
                """
        }
}
```

CandleFire.kt

```
package com.example.course

import android.content.Context
import android.graphics.BitmapFactory
import android.opengl.GLES20
import android.opengl.GLUtils
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.nio.FloatBuffer
import java.nio.ShortBuffer
import kotlin.math.cos
import kotlin.math.sin

class CandleFire(
    private val context: Context,
    private val latitudeBands: Int = 40,
    private val longitudeBands: Int = 40,
    private val radius: Float = 1.0f
) {
    private lateinit var shaderProgram: ShaderProgram
    private lateinit var vertexBuffer: FloatBuffer
    private lateinit var indexBuffer: ShortBuffer
    private lateinit var textureBuffer: FloatBuffer
    private val heightOffset: Float = 1.5f // Установите значение по
умолчанию
    private val vertices: FloatArray
    private val indices: ShortArray
    private val textureCoords: FloatArray

    // Список текстур для анимации огня
    private val fireTextures = listOf(
        loadTexture(R.drawable.fire0),
        loadTexture(R.drawable.fire1),
        loadTexture(R.drawable.fire3),
    )


    fun loadTexture(resId: Int): Int {
        val textureHandle = IntArray(1)
        GLES20.glGenTextures(1, textureHandle, 0)

        if (textureHandle[0] == 0) {
            throw RuntimeException("Error generating texture")
        }

        // Загружаем изображение из ресурсов
        val options = BitmapFactory.Options().apply { inScaled = false } //
Не масштабируем изображение
        val bitmap = BitmapFactory.decodeResource(context.resources, resId,
options)

        // Привязываем текстуру и настраиваем параметры
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureHandle[0])

        GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR)
```

```
        GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR)

        // Загружаем текстуру в OpenGL
        GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0)

        // Освобождаем ресурсы изображения, так как оно теперь загружено в
OpenGL
        bitmap.recycle()

        return textureHandle[0]
    }


    private var currentFrame = 0
    private val frameInterval = 1000 // Интервал смены кадров (миллисекунды)
    private var lastFrameTime = System.currentTimeMillis()

    init {
        val vertexList = mutableListOf<Float>()
        val indexList = mutableListOf<Short>()
        val textureList = mutableListOf<Float>()

        for (lat in 0..latitudeBands) {
            val theta = lat * Math.PI / latitudeBands
            val sinTheta = sin(theta).toFloat()
            val cosTheta = cos(theta).toFloat()

            for (long in 0..longitudeBands) {
                val phi = long * 2 * Math.PI / longitudeBands
                val sinPhi = sin(phi).toFloat()
                val cosPhi = cos(phi).toFloat()

                val x = cosPhi * sinTheta
                val y = cosTheta * 5 - heightOffset
                val z = sinPhi * sinTheta

                vertexList.add(x * radius)
                vertexList.add(y * radius)
                vertexList.add(z * radius)

                val u = 1f - (long / longitudeBands.toFloat())
                val v = 1f - (lat / latitudeBands.toFloat())
                textureList.add(u)
                textureList.add(v)
            }
        }

        for (lat in 0 until latitudeBands) {
            for (long in 0 until longitudeBands) {
                val first = (lat * (longitudeBands + 1) + long).toShort()
                val second = (first + longitudeBands + 1).toShort()

                indexList.add(first)
                indexList.add(second)
                indexList.add((first + 1).toShort())

                indexList.add(second)
                indexList.add((second + 1).toShort())
```

```
                indexList.add((first + 1).toShort())
            }
        }

        vertices = vertexList.toFloatArray()
        indices = indexList.toShortArray()
        textureCoords = textureList.toFloatArray()
    }

    fun initialize() {
        shaderProgram = ShaderProgram(VERTEX_SHADER_CODE,
FRAGMENT_SHADER_CODE)

        vertexBuffer = ByteBuffer.allocateDirect(vertices.size * 4).run {
            order(ByteOrder.nativeOrder())
            asFloatBuffer().apply {
                put(vertices)
                position(0)
            }
        }

        indexBuffer = ByteBuffer.allocateDirect(indices.size * 2).run {
            order(ByteOrder.nativeOrder())
            asShortBuffer().apply {
                put(indices)
                position(0)
            }
        }

        textureBuffer = ByteBuffer.allocateDirect(textureCoords.size * 4).run
{
            order(ByteOrder.nativeOrder())
            asFloatBuffer().apply {
                put(textureCoords)
                position(0)
            }
        }
    }

    fun draw(mvpMatrix: FloatArray, time: Float) {
        val currentTime = System.currentTimeMillis()
        if (currentTime - lastFrameTime > frameInterval) {
            currentFrame = (currentFrame + 1) % fireTextures.size
            lastFrameTime = currentTime
        }
        val textureId = fireTextures[currentFrame]

        shaderProgram.use()

        val positionHandle =
GLES20.glGetAttribLocation(shaderProgram.programId, "a_Position")
        val texCoordHandle =
GLES20.glGetAttribLocation(shaderProgram.programId, "a_TexCoord")
        val mvpMatrixHandle =
GLES20.glGetUniformLocation(shaderProgram.programId, "u_MVPMatrix")
        val timeHandle = GLES20.glGetUniformLocation(shaderProgram.programId,
"u_Time")

        GLES20.glUniformMatrix4fv(mvpMatrixHandle, 1, false, mvpMatrix, 0)
```

```
        GLES20.glUniform1f(timeHandle, time)

        GLES20.glEnableVertexAttribArray(positionHandle)
        GLES20.glEnableVertexAttribArray(texCoordHandle)

        GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT,
false, 0, vertexBuffer)
        GLES20.glVertexAttribPointer(texCoordHandle, 2, GLES20.GL_FLOAT,
false, 0, textureBuffer)

        GLES20.glActiveTexture(GLES20.GL_TEXTURE0)
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId)

        GLES20.glDrawElements(GLES20.GL_TRIANGLES, indices.size,
GLES20.GL_UNSIGNED_SHORT, indexBuffer)

        GLES20.glDisableVertexAttribArray(positionHandle)
        GLES20.glDisableVertexAttribArray(texCoordHandle)
    }

    companion object {
        private const val VERTEX_SHADER_CODE = """
    attribute vec4 a_Position;
    attribute vec2 a_TexCoord;
    uniform mat4 u_MVPMatrix;
    uniform float u_Time;
    varying vec2 v_TexCoord;
    varying float v_Offset;

    void main() {
        float waveHeightX = cos(a_Position.z * 1.0 + u_Time * 2.0) * 0.01; //
Adjusted to create snake-like motion
        float waveHeightZ = sin(a_Position.x * 1.0 + u_Time * 2.0) * 0.0; //
Adjusted to create snake-like motion
        float waveHeightY = cos(u_Time * 2.0) * 0.0; // Add vertical wave for
upward movement

        vec4 modPosition = a_Position;
        modPosition.x += waveHeightX;
        modPosition.z += waveHeightZ;
        modPosition.y += waveHeightY; // Apply the vertical wave

        gl_Position = u_MVPMatrix * modPosition;

        // Set texture coordinates
        // Use the y-coordinate to decide the texture portion
        if (a_Position.x > 0.0) {
            // Top half of the flame texture
            v_TexCoord = vec2(a_TexCoord.x * 0.9, a_TexCoord.x); // Left side
        } else {
            // Bottom half of the flame texture
            v_TexCoord = vec2(a_TexCoord.x * 0.9 + 0.5, a_TexCoord.x); //
Right side
        }

        v_Offset = waveHeightY; // Send the offset for use in the fragment
shader
    }
"""
```

```
        private const val FRAGMENT_SHADER_CODE = """
            precision mediump float;
            varying vec2 v_TexCoord;
            varying float v_Offset;
            uniform sampler2D u_Texture;

            void main() {
                vec4 texColor = texture2D(u_Texture, v_TexCoord);
                float flicker = sin(v_Offset * 1.0) * 0.1;
                texColor.r += flicker * 0.1;
                texColor.g -= flicker * 0.1;
                texColor.a = 1.0 - flicker * 0.1;

                gl_FragColor = texColor;
            }
        """
    }
}
```

Sphere.kt

```
package com.example.course

import android.opengl.GLES20
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.nio.FloatBuffer
import java.nio.ShortBuffer
import kotlin.math.cos
import kotlin.math.sin

class Sphere(private val latitudeBands: Int = 40, private val longitudeBands:
Int = 40, val radius: Float = 1.0f) {
    private lateinit var shaderProgram: ShaderProgram
    private lateinit var vertexBuffer: FloatBuffer
    private lateinit var indexBuffer: ShortBuffer
    private lateinit var textureBuffer: FloatBuffer
    private lateinit var normalBuffer: FloatBuffer

    private val vertices: FloatArray
    private val indices: ShortArray
    private val textureCoords: FloatArray
    private val normals: FloatArray

    init {
        val vertexList = mutableListOf<Float>()
        val indexList = mutableListOf<Short>()
        val textureList = mutableListOf<Float>()
        val normalList = mutableListOf<Float>()

        for (lat in 0..latitudeBands) {
            val theta = lat * Math.PI / latitudeBands
            val sinTheta = sin(theta).toFloat()
            val cosTheta = cos(theta).toFloat()

            for (long in 0..longitudeBands) {
                val phi = long * 2 * Math.PI / longitudeBands
                val sinPhi = sin(phi).toFloat()
                val cosPhi = cos(phi).toFloat()

                val x = cosPhi * sinTheta
                val y = cosTheta
                val z = sinPhi * sinTheta

                vertexList.add(x * radius)
                vertexList.add(y * radius)
                vertexList.add(z * radius)

                // Normal vectors for lighting calculations
                normalList.add(x) // Normal x
                normalList.add(y) // Normal y
                normalList.add(z) // Normal z

                val u = 1f - (long / longitudeBands.toFloat())
                val v = 1f - (lat / latitudeBands.toFloat())
                textureList.add(u)
                textureList.add(v)
            }
```

```
        }

        for (lat in 0 until latitudeBands) {
            for (long in 0 until longitudeBands) {
                val first = (lat * (longitudeBands + 1) + long).toShort()
                val second = (first + longitudeBands + 1).toShort()

                indexList.add(first)
                indexList.add(second)
                indexList.add((first + 1).toShort())

                indexList.add(second)
                indexList.add((second + 1).toShort())
                indexList.add((first + 1).toShort())
            }
        }

        vertices = vertexList.toFloatArray()
        indices = indexList.toShortArray()
        textureCoords = textureList.toFloatArray()
        normals = normalList.toFloatArray()
    }

    init {
        shaderProgram = ShaderProgram(VERTEX_SHADER_CODE,
FRAGMENT_SHADER_CODE)

        vertexBuffer = ByteBuffer.allocateDirect(vertices.size * 4).run {
            order(ByteOrder.nativeOrder())
            asFloatBuffer().apply {
                put(vertices)
                position(0)
            }
        }

        indexBuffer = ByteBuffer.allocateDirect(indices.size * 2).run {
            order(ByteOrder.nativeOrder())
            asShortBuffer().apply {
                put(indices)
                position(0)
            }
        }

        textureBuffer = ByteBuffer.allocateDirect(textureCoords.size * 4).run
{
            order(ByteOrder.nativeOrder())
            asFloatBuffer().apply {
                put(textureCoords)
                position(0)
            }
        }

        normalBuffer = ByteBuffer.allocateDirect(normals.size * 4).run {
            order(ByteOrder.nativeOrder())
            asFloatBuffer().apply {
                put(normals)
                position(0)
            }
        }
```

```
    }

    fun draw(mvpMatrix: FloatArray, normalMatrix: FloatArray, lightPos:
FloatArray, viewPos: FloatArray, textureId: Int) {
        shaderProgram.use()



        val positionHandle =
GLES20.glGetAttribLocation(shaderProgram.programId, "a_Position")
        val texCoordHandle =
GLES20.glGetAttribLocation(shaderProgram.programId, "a_TexCoord")
        val normalHandle =
GLES20.glGetAttribLocation(shaderProgram.programId, "a_Normal")
        val mvpMatrixHandle =
GLES20.glGetUniformLocation(shaderProgram.programId, "u_MVPMatrix")
        val normalMatrixHandle =
GLES20.glGetUniformLocation(shaderProgram.programId, "u_NormalMatrix")
        val lightPosHandle =
GLES20.glGetUniformLocation(shaderProgram.programId, "u_LightPos")
        val viewPosHandle =
GLES20.glGetUniformLocation(shaderProgram.programId, "u_ViewPos")

        GLES20.glUniformMatrix4fv(mvpMatrixHandle, 1, false, mvpMatrix, 0)
        GLES20.glUniformMatrix4fv(normalMatrixHandle, 1, false, normalMatrix,
0)
        GLES20.glUniform3fv(lightPosHandle, 1, lightPos, 0)
        GLES20.glUniform3fv(viewPosHandle, 1, viewPos, 0)

        GLES20.glEnableVertexAttribArray(positionHandle)
        GLES20.glEnableVertexAttribArray(texCoordHandle)
        GLES20.glEnableVertexAttribArray(normalHandle)

        GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT,
false, 0, vertexBuffer)
        GLES20.glVertexAttribPointer(texCoordHandle, 2, GLES20.GL_FLOAT,
false, 0, textureBuffer)
        GLES20.glVertexAttribPointer(normalHandle, 3, GLES20.GL_FLOAT, false,
0, normalBuffer)

        GLES20.glActiveTexture(GLES20.GL_TEXTURE0)
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId)

        GLES20.glDrawElements(
            GLES20.GL_TRIANGLES,
            indices.size,
            GLES20.GL_UNSIGNED_SHORT,
            indexBuffer
        )

        GLES20.glDisableVertexAttribArray(positionHandle)
        GLES20.glDisableVertexAttribArray(texCoordHandle)
        GLES20.glDisableVertexAttribArray(normalHandle)
    }

    companion object {
        private const val VERTEX_SHADER_CODE = """
            attribute vec4 a_Position;
            attribute vec2 a_TexCoord;
```

```
            attribute vec3 a_Normal;
            uniform mat4 u_MVPMatrix;
            uniform mat4 u_NormalMatrix;
            uniform vec3 u_LightPos;
            uniform vec3 u_ViewPos;
            varying vec2 v_TexCoord;
            varying vec3 v_Normal;
            varying vec3 v_LightDir;
            varying vec3 v_ViewDir;

            void main() {
                gl_Position = u_MVPMatrix * a_Position;

                // Transform the normal to eye space
                v_Normal = normalize(vec3(u_NormalMatrix * vec4(a_Normal,
0.0)));

                // Compute light direction and view direction
                v_LightDir = normalize(u_LightPos - vec3(gl_Position));
                v_ViewDir = normalize(u_ViewPos - vec3(gl_Position));

                v_TexCoord = a_TexCoord;
            }
        """

        private const val FRAGMENT_SHADER_CODE = """
            precision mediump float;
            varying vec2 v_TexCoord;
            varying vec3 v_Normal;
            varying vec3 v_LightDir;
            varying vec3 v_ViewDir;
            uniform sampler2D u_Texture;

            void main() {
                vec4 texColor = texture2D(u_Texture, v_TexCoord);

                // Normalize the normal vector
                vec3 norm = normalize(v_Normal);

                // Compute the diffuse and specular lighting
                float diff = max(dot(norm, v_LightDir), 0.0);
                vec3 reflectDir = reflect(-v_LightDir, norm);
                float spec = pow(max(dot(v_ViewDir, reflectDir), 0.0), 32.0);
// Shininess factor

                // Combine the color and lighting
                vec3 ambient = vec3(0.1) * texColor.rgb; // Ambient light
                vec3 diffuse = diff * texColor.rgb; // Diffuse light
                vec3 specular = spec * vec3(1.0); // Specular light color
(white)

                vec3 finalColor = ambient + diffuse + specular;
                gl_FragColor = vec4(finalColor, texColor.a);
            }
        """
    }
}
```

ShaderProgram.kt

```
package com.example.course

import android.opengl.GLES20

class ShaderProgram(vertexShaderCode: String, fragmentShaderCode: String) {
    var programId: Int

    init {
        val vertexShader = compileShader(GLES20.GL_VERTEX_SHADER,
vertexShaderCode)
        val fragmentShader = compileShader(GLES20.GL_FRAGMENT_SHADER,
fragmentShaderCode)

        programId = GLES20.glCreateProgram().also {
            GLES20.glAttachShader(it, vertexShader)
            GLES20.glAttachShader(it, fragmentShader)
            GLES20.glLinkProgram(it)
        }

        val linkStatus = IntArray(1)
        GLES20.glGetProgramiv(programId, GLES20.GL_LINK_STATUS, linkStatus,
0)
        print(programId)
    }

    fun use() {
        GLES20.glUseProgram(programId)
    }

    fun getAttributeLocation(name: String): Int {
        return GLES20.glGetAttribLocation(programId, name)
    }

    fun getUniformLocation(name: String): Int {
        return GLES20.glGetUniformLocation(programId, name)
    }

    private fun compileShader(type: Int, shaderCode: String): Int {
        val shader = GLES20.glCreateShader(type)
        GLES20.glShaderSource(shader, shaderCode)
        GLES20.glCompileShader(shader)

        val compileStatus = IntArray(1)
        GLES20.glGetShaderiv(shader, GLES20.GL_COMPILE_STATUS, compileStatus,
0)
        if (compileStatus[0] == 0) {
            val errorMsg = GLES20.glGetShaderInfoLog(shader)
            throw RuntimeException("Ошибка компиляции шейдера: $errorMsg")
        }


        return shader
    }

    init {
        val vertexShader = compileShader(GLES20.GL_VERTEX_SHADER,
vertexShaderCode)
```

```
        val fragmentShader = compileShader(GLES20.GL_FRAGMENT_SHADER,
fragmentShaderCode)

        programId = GLES20.glCreateProgram().also {
            GLES20.glAttachShader(it, vertexShader)
            GLES20.glAttachShader(it, fragmentShader)
            GLES20.glLinkProgram(it)

            val linkStatus = IntArray(1)
            GLES20.glGetProgramiv(programId, GLES20.GL_LINK_STATUS,
linkStatus, 0)
        }
    }
}
```