Федеральное государственное бюджетное образовательное учреждение
«Сибирский государственный университет телекоммуникаций и информатики»

Кафедра ПМиК

Курсовая работа

по дисциплине

«Программирование мобильных устройств»

Выполнил:
студент 4 курса
ИВТ, гр. ИП-114
Яворский Д.И.

Проверила:
Старший преподаватель
Павлова У.В.

Новосибирск 2024

Оглавление

Постановка задачи

В рамках выполнения курсовой работы необходимо создать программу в которой нарисован стол на OpenGL ES.

Предлагаются следующие варианты:

- Задание на оценку удовлетворительно (OpenGL ES 1.0):
  На столе лежат различные фрукты/овощи (не менее 4 различных), стакан с напитком. На объекты наложены текстуры.

- Задание на оценку хорошо (OpenGL ES 2.0):
  На столе лежат различные фрукты/овощи (не менее 4 различных), стакан с напитком. Имеется освещение по модели Фонга.

- Задание на оценку отлично (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.

Выполнение

Я выбрал вариант на оценку хорошо: на столе лежат различные фрукты/овощи (не менее 4 различных), стакан с напитком. Имеется освещение по модели Фонга.

Большинство предметов было реализовано с помощью класса «Sphere». Данный класс создает 3D-модель сферы с возможностью деформации по осям (растяжение или сжатие) и поддержкой текстур и освещения.

Стол и стакан имеют отдельные классы:
«Table» для отрисовки 3D-модели стола, он содержит вершины, нормали, текстурные координаты и индексы для отрисовки как столешницы, так и ножек стола;
«Glass» реализует отрисовку стакана с жидкостью внутри.

Класс «ShaderProgram» предназначен для работы с шейдерами в OpenGL ES 2.0. Он отвечает за создание, компиляцию, привязку шейдеров и использование шейдеров.
Класс «Renderer» создаёт сцену, состоящую из различных объектов (сфера, стакан, стол), которые текстурированы, позиционируются и отображаются с учетом освещения и перспективы.

Освещение по модели Фонга для всех предметов прописано в каждом классе.

Результаты работы

Заключение

В результате выполнения данной курсовой работы я создал свою первую программу с OpenGL ES 2 с 5 предметами различной формой и наложенными текстурами, реализованными в разных классах с освещением по модели Фонга.

Листинг

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(0.00f, 0.3f, 0f)
    private val cucumberInGlass = floatArrayOf(0.5f, 0.0f, -0.1f)

    private lateinit var table: Table
    private lateinit var glass: Glass
    private lateinit var apple: Sphere
    private lateinit var pumpkin: Sphere
    private lateinit var orange: Sphere
    private lateinit var cucumber: Sphere

    private var appleTexture: Int = 0
    private var pumpkinTexture: Int = 0
    private var orangeTexture: Int = 0
```

```
private var cucumberTexture: 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)

    pumpkin = Sphere(radius = 0.9f/3)
    pumpkinTexture = loadTexture(R.drawable.tikva)

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

    cucumber = Sphere(radius = 0.05f, stretchFactorY = 5f)
    cucumberTexture = loadTexture(R.drawable.cucumber)

}

override fun onDrawFrame(arg0: GL10?) {
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT or GLES20.GL_DEPTH_BUFFER_BIT)
    Matrix.setLookAtM(viewMatrix, 0, 1f, 1f, 2f, 0f, 0f, 0f, 0f, 1f, 0f)

    // Установление единичной матрицы (дефолт позиция, масштаб, поворот)
    Matrix.setIdentityM(modelMatrix, 0)

    // Инверсия, чтобы правильно вычислить нормали
    Matrix.invertM(normalMatrix, 0, viewMatrix, 0)

    // Перемещение вдоль осей
    Matrix.translateM(modelMatrix, 0, 0f, -0.3f, 0f)

    // Поворот
    Matrix.rotateM(modelMatrix, 0, 0f, 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, normalMatrix, lightPos, viewMatrix)

    // Тыква
    Matrix.setIdentityM(modelMatrix, 0)
    Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
    Matrix.translateM(modelMatrix, 0, -0.0f, 0.1f, -0.5f)
    Matrix.rotateM(modelMatrix, 0, 180f, 0f, 1f, 0f)
    Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
    Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
    pumpkin.draw(mVPMatrix, normalMatrix, lightPos, viewMatrix, pumpkinTexture)

    // Яблоко
    Matrix.setIdentityM(modelMatrix, 0)
    Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
    Matrix.translateM(modelMatrix, 0, 0.75f, 0.0f, 0.6f)
    Matrix.rotateM(modelMatrix, 0, 180f, 0f, 1f, 0f)
    Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
    Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
    apple.draw(mVPMatrix, normalMatrix, lightPos, viewMatrix, appleTexture)

    // Апельсин
    Matrix.setIdentityM(modelMatrix, 0)
```

```
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.translateM(modelMatrix, 0, -0.3f, 0.0f, 0.9f)
        Matrix.rotateM(modelMatrix, 0, 180f, 0f, 1f, 0f)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        orange.draw(mVPMatrix, normalMatrix, lightPos, viewMatrix, orangeTexture)

        // Огурец
        Matrix.setIdentityM(modelMatrix, 0)
        Matrix.translateM(modelMatrix, 0, -0.0f, 0f, 0.2f)
        Matrix.rotateM(modelMatrix, 0, 90f, 3f, 1f, 5f) // Поворот вокруг оси Y
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        cucumber.draw(mVPMatrix, normalMatrix, lightPos, viewMatrix, cucumberTexture)

        // Стакан
        Matrix.setIdentityM(modelMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.translateM(modelMatrix,0, cucumberInGlass[0],
            cucumberInGlass[1],
            cucumberInGlass[2]
        )
        Matrix.multiplyMM(mVPMatrix, 0, projectionMatrix, 0, viewMatrix, 0)
        Matrix.multiplyMM(mVPMatrix, 0, mVPMatrix, 0, modelMatrix, 0)
        glass.draw(mVPMatrix, lightPos, viewMatrix)
    }

    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, 20f) // увеличен диапазон дальности

        Matrix.setLookAtM(viewMatrix, 0, 1f, 1f, 2f, 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
    }
}
```

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)
    }

    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)
        }
    }
}
```

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
import kotlin.math.sqrt

class Sphere(
    private val latitudeBands: Int = 40,
    private val longitudeBands: Int = 40,
    val radius: Float = 1.0f,
    private val stretchFactorX: Float = 1.0f, // Коэффициент вытягивания по оси X
    private val stretchFactorY: Float = 1.0f, // Коэффициент вытягивания по оси Y
    private val stretchFactorZ: Float = 1.0f  // Коэффициент вытягивания по оси Z
) {
```

```
private var shaderProgram: ShaderProgram =
    ShaderProgram(VERTEX_SHADER_CODE, FRAGMENT_SHADER_CODE)
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>()
    val startPhi = 0.0 // Начало дольки
    val endPhi = Math.PI / 3 // 1/6 часть сферы
    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 * stretchFactorX
            val y = cosTheta * stretchFactorY
            val z = sinPhi * sinTheta * stretchFactorZ

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

            // Normal vectors for lighting calculations
            val normalX = x / stretchFactorX
            val normalY = y / stretchFactorY
            val normalZ = z / stretchFactorZ
            val normalLength = sqrt(normalX * normalX + normalY * normalY + normalZ * normalZ)
            normalList.add(normalX / normalLength) // Normal x
            normalList.add(normalY / normalLength) // Normal y
            normalList.add(normalZ / normalLength) // 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())
        }
    }

    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 = startPhi + (long / longitudeBands.toFloat()) * (endPhi - startPhi)
                val sinPhi = sin(phi).toFloat()
                val cosPhi = cos(phi).toFloat()

                val x = cosPhi * sinTheta * stretchFactorX
                val y = cosTheta * stretchFactorY
                val z = sinPhi * sinTheta * stretchFactorZ

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

                val normalX = x / stretchFactorX
                val normalY = y / stretchFactorY
                val normalZ = z / stretchFactorZ
                val normalLength = sqrt(normalX * normalX + normalY * normalY + normalZ * normalZ)
                normalList.add(normalX / normalLength)
                normalList.add(normalY / normalLength)
                normalList.add(normalZ / normalLength)

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

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

    init {

        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);
}

        """
    }
}
```

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, 0.37f, 30) // Высота стакана увеличена
    private val liquidVertices: FloatArray = generateCylinderVertices(0.09f, 0.2f, 30) // Высота жидкости уменьшена

    // 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)
        }
    }

    // Изменения в цветах для жидкости (должен быть голубоватый оттенок)
    private val liquidColors: FloatArray = FloatArray(liquidVertices.size / 3 * 4).apply {
        for (i in indices step 4) {
            this[i] = 0.7f  // Красный компонент (0 для чистого голубого)
            this[i + 1] = 0.1f  // Зеленый компонент (чуть выше для дополнительной яркости)
            this[i + 2] = 0.1f  // Синий компонент (максимальная насыщенность)
            this[i + 3] = 0.8f  // Прозрачность жидкости (оптимальная для яркого цвета)
        }
    }

    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);
        }
        """
    }
}
```

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(
        // Tabletop top face
        -1f, 0.1f, 1f,   // 0
        -1f, 0.1f, -1f,  // 1
        1f, 0.1f, -1f,   // 2
        1f, 0.1f, 1f,    // 3
        // Tabletop bottom face
        -1f, -0.1f, 1f,  // 4
        -1f, -0.1f, -1f, // 5
        1f, -0.1f, -1f,  // 6
        1f, -0.1f, 1f,   // 7
        // Tabletop front face
        -1f, 0.1f, 1f,   // 8
        -1f, -0.1f, 1f,  // 9
        1f, -0.1f, 1f,   // 10
        1f, 0.1f, 1f,    // 11
        // Tabletop back face
        -1f, 0.1f, -1f,  // 12
        -1f, -0.1f, -1f, // 13
        1f, -0.1f, -1f,  // 14
        1f, 0.1f, -1f,   // 15
        // Tabletop left face
        -1f, 0.1f, 1f,   // 16
        -1f, -0.1f, 1f,  // 17
        -1f, -0.1f, -1f, // 18
        -1f, 0.1f, -1f,  // 19
        // Tabletop right face
        1f, 0.1f, 1f,    // 20
        1f, -0.1f, 1f,   // 21
        1f, -0.1f, -1f,  // 22
        1f, 0.1f, -1f,   // 23
        // Front left leg
        -0.8f, 0f, 0.8f, // 24
        -0.8f, -2f, 0.8f,// 25
        -0.7f, -2f, 0.8f,// 26
        -0.7f, 0f, 0.8f, // 27
```

```
    -0.8f, 0f, 0.7f, // 28
    -0.8f, -2f, 0.7f,// 29
    -0.7f, -2f, 0.7f,// 30
    -0.7f, 0f, 0.7f, // 31
    // Front right leg
    0.7f, 0f, 0.8f,  // 32
    0.7f, -2f, 0.8f, // 33
    0.8f, -2f, 0.8f, // 34
    0.8f, 0f, 0.8f,  // 35
    0.7f, 0f, 0.7f,  // 36
    0.7f, -2f, 0.7f, // 37
    0.8f, -2f, 0.7f, // 38
    0.8f, 0f, 0.7f,  // 39
    // Back left leg
    -0.8f, 0f, -0.7f,// 40
    -0.8f, -2f, -0.7f,//41
    -0.7f, -2f, -0.7f,//42
    -0.7f, 0f, -0.7f, //43
    -0.8f, 0f, -0.8f, //44
    -0.8f, -2f, -0.8f,//45
    -0.7f, -2f, -0.8f,//46
    -0.7f, 0f, -0.8f, //47
    // Back right leg
    0.7f, 0f, -0.8f,  //48
    0.7f, -2f, -0.8f, //49
    0.8f, -2f, -0.8f, //50
    0.8f, 0f, -0.8f,  //51
    0.7f, 0f, -0.7f,  //52
    0.7f, -2f, -0.7f, //53
    0.8f, -2f, -0.7f, //54
    0.8f, 0f, -0.7f   //55
)

private val normals = floatArrayOf(
    // Tabletop top face
    0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f,
    // Tabletop bottom face
    0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f,
    // Tabletop front face
    0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f,
    // Tabletop back face
    0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f,
    // Tabletop left face
    -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f,
    // Tabletop right face
    1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f,
    // Front left leg
    0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f,
    0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f,
    // Front right leg
    0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f, 0f, 0f, 1f,
    0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f,
    // Back left leg
    0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f,
    0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f,
    // Back right leg
    0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f,
    0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f, 0f, -1f, 0f
)


private val texCoords = floatArrayOf(
    // Tabletop top face
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    // Tabletop bottom face
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    // Tabletop front face
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    // Tabletop back face
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
```

```
    // Tabletop left face
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    // Tabletop right face
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    // Front left leg
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    // Front right leg
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    // Back left leg
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    // Back right leg
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f,
    0f, 0f, 0f, 1f, 1f, 1f, 1f, 0f
)
private val indices = shortArrayOf(
    // Tabletop top face
    0, 1, 2, 0, 2, 3,
    // Tabletop bottom face
    4, 5, 6, 4, 6, 7,
    // Tabletop front face
    8, 9, 10, 8, 10, 11,
    // Tabletop back face
    12, 13, 14, 12, 14, 15,
    // Tabletop left face
    16, 17, 18, 16, 18, 19,
    // Tabletop right face
    20, 21, 22, 20, 22, 23,
    // Front left leg
    24, 25, 26, 24, 26, 27,
    28, 29, 30, 28, 30, 31,
    // Front right leg
    32, 33, 34, 32, 34, 35,
    36, 37, 38, 36, 38, 39,
    // Back left leg
    40, 41, 42, 40, 42, 43,
    44, 45, 46, 44, 46, 47,
    // Back right leg
    48, 49, 50, 48, 50, 51,
    52, 53, 54, 52, 54, 55
)

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.table)
    }

    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, normalMatrix: FloatArray, lightPos: FloatArray, viewPos: 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 normalHandle = GLES20.glGetAttribLocation(program, "aNormal")
        GLES20.glEnableVertexAttribArray(normalHandle)
        GLES20.glVertexAttribPointer(normalHandle, 3, GLES20.GL_FLOAT, false, 12, normalBuffer)

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

        val normalMatrixHandle = GLES20.glGetUniformLocation(program, "uNormalMatrix")
        GLES20.glUniformMatrix4fv(normalMatrixHandle, 1, false, normalMatrix, 0)

        val lightPosHandle = GLES20.glGetUniformLocation(program, "uLightPos")
        GLES20.glUniform3fv(lightPosHandle, 1, lightPos, 0)

        val viewPosHandle = GLES20.glGetUniformLocation(program, "uViewPos")
        GLES20.glUniform3fv(viewPosHandle, 1, viewPos, 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)
        GLES20.glDisableVertexAttribArray(normalHandle)
    }

    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;
    uniform mat4 uNormalMatrix;
    uniform vec3 uLightPos;
    uniform vec3 uViewPos;
    attribute vec4 vPosition;
    attribute vec2 aTexCoord;
    attribute vec3 aNormal;
    varying vec2 vTexCoord;
    varying vec3 vNormal;
    varying vec3 vLightDir;
    varying vec3 vViewDir;

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

        // Compute normal in eye space
        vNormal = normalize(mat3(uNormalMatrix) * aNormal); // Преобразуем нормаль с нормализацией

        // Compute light direction and view direction in eye space
        vec3 worldPos = vec3(uMVPMatrix * vPosition);
        vLightDir = normalize(uLightPos - worldPos);
        vViewDir = normalize(uViewPos - worldPos);
    }
      """

        private const val FRAGMENT_SHADER_CODE = """
    precision mediump float;
    varying vec2 vTexCoord;
    varying vec3 vNormal;
    varying vec3 vLightDir;
    varying vec3 vViewDir;
    uniform sampler2D uTexture;

    void main() {
        vec4 texColor = texture2D(uTexture, vTexCoord);

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

        // Compute the diffuse and specular lighting
        float diff = max(dot(norm, normalize(vLightDir)), 0.0); // Нормализация светового направления
        vec3 reflectDir = reflect(-normalize(vLightDir), norm); // Инвертируем световое направление для отражения
        float spec = pow(max(dot(normalize(vViewDir), reflectDir), 0.0), 64.0); // 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);
    }
    """
    }
}
```