Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра ПМиК

Лабораторная работа №5
по дисциплине
«Программирование мобильных устройств»

Выполнил:
студент гр. ИП-813
Бурдуковский И.А.

Проверила:
Павлова У.В.

Новосибирск 2021

Оглавление

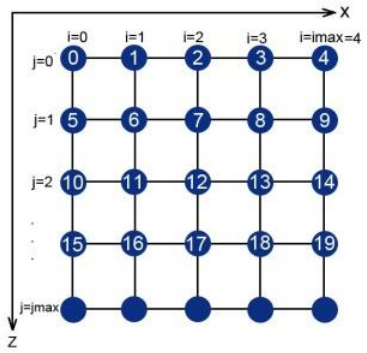
Задание

Создать водную поверхность, прозрачную до дна (взять произвольный рисунок).

По поверхности должна идти волна.

Выполнение

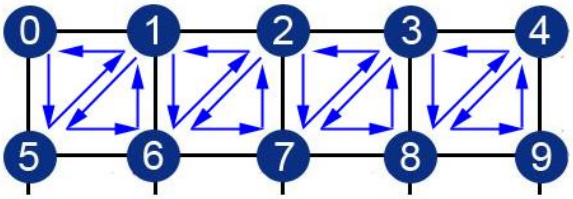
Пусть сетка будет лежать в плоскости XZ, а значение Y будет вычисляться как функция от X и Z, т.е. y=f(x,z). В узлах сетки будут находиться вершины. Обозначили порядковый номер узла сетки вдоль оси X как i, а вдоль оси Z как j. Номера узлов могут меняться от нуля до imax или jmax соответственно.



Обозначили шаг сетки вдоль оси X как dx, а шаг сетки вдоль оси Z как dz. Тогда мы можем легко вычислить координаты X и Z для всех вершин. Координата Y у нас меняется в зависимости от двух координат других координат X и Z. Поэтому будем хранить ее в двумерном массиве.

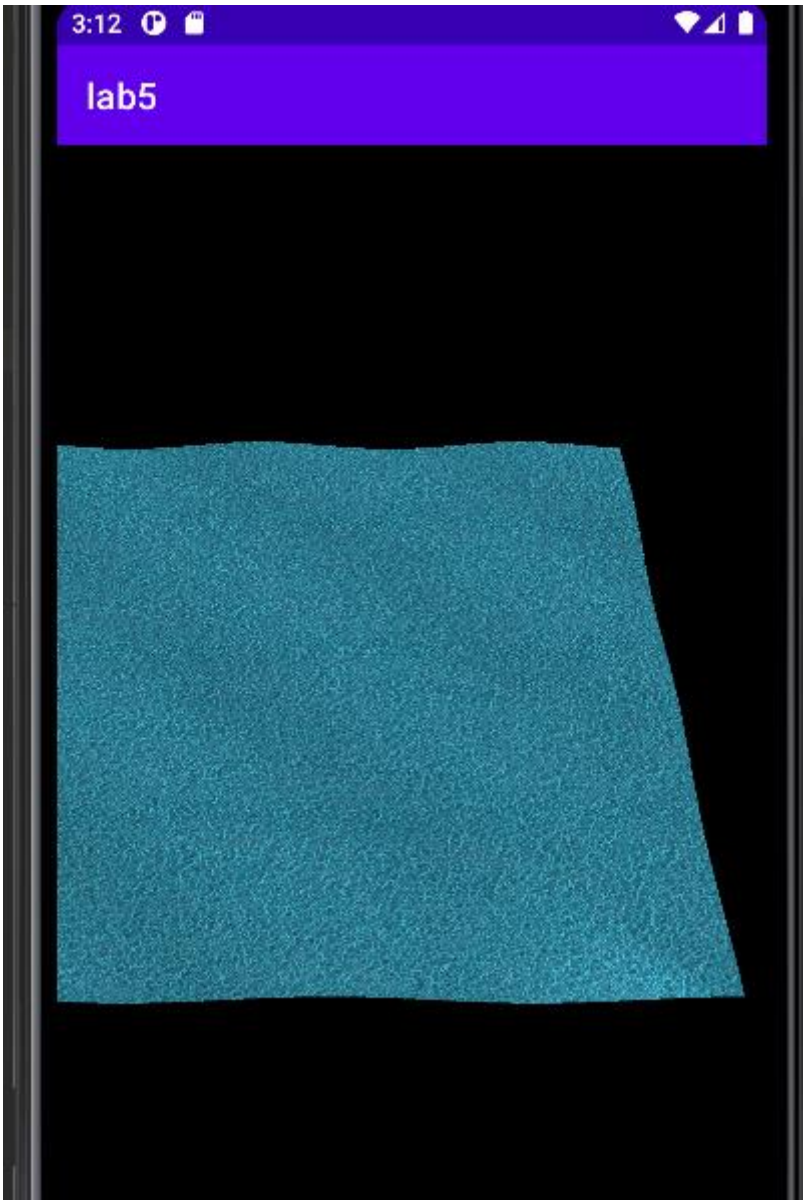
Для того чтобы использовать освещение нужно было вычислить нормаль для каждой вершины сетки. Вектор нормали - это вектор единичной длины, перпендикулярный к поверхности в данной точке этой поверхности и направленный от обратной стороны поверхности к лицевой стороне. Чтобы нарисовать поверхность, потребовалось ее разбить на множество треугольников. Лицевой стороной треугольника является сторона, которая при рисовании обходится по вершинам против часовой стрелки:

Далее рисовали поверхность в одном проходе, т.е. с использованием одной команды glDrawElements и правила обхода вершин GL_TRIANGLE_STRIP. Разбили сетку на ленты из треугольников. Правило GL_TRIANGLE_STRIP автоматически создает ряд треугольников 0-5-1, 1-5-6, 1-6-2, 2-6-7, 2-7-3, 3-7-8, 3-8-4, 4-8-9. Порядок перечисления вершин выбран так, чтобы обход выполнялся против часовой стрелки и верхняя сторона сетки считалась лицевой.



Таким образом, чтобы сделать единую цепь вершин для правила GL_TRIANGLE_STRIP нужно дублировать последнюю вершину в каждой ленте. Порядок перечисления вершин называют массивом индексов и передают OpenGL в виде буфера.

Результат:



Листинг проекта

MainActivity.java

```
package com.example.lab5;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.opengl.GLSurfaceView;


public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        MyRenderer render = new MyRenderer(this);
        super.onCreate(savedInstanceState);
        GLSurfaceView canvas = new GLSurfaceView(this);
        canvas.setEGLContextClientVersion(2);
        canvas.setRenderer(render);
        canvas.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
        setContentView(canvas);
    }
}
```

MyRenderer.java

```
package com.example.lab5;

import android.annotation.SuppressLint;
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.opengl.GLES20;
import android.opengl.GLSurfaceView;
import android.opengl.GLUtils;
import android.opengl.Matrix;

import java.io.InputStream;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.nio.ShortBuffer;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;


class MyRenderer implements GLSurfaceView.Renderer {
    Context c;
    private int[] texture = new int [1];
    private float x_camera, y_camera, z_camera;
    private float x_light_position, y_light_position, z_light_position;
    private float[] model_matrix;
    private float[] view_matrix;
    private float[] model_view_matrix;
    private float[] projection_matrix;
    private float[] model_view_projection_matrix;
    private int max_size_x = 60;
    private int max_size_z = 60;
```

```
private int size_index;
private float x0 =- 1.3f;
private float z0 = -2f;
private float dx = 0.05f;
private float dz = 0.05f;
private float [] x;
private float [][] y;
private float [] z;
private float [] vertexes;
private float [][] normales_x;
private float [][] normales_y;
private float [][] normales_z;
private float [] normales;
private FloatBuffer vertexes_buffer, normales_buffer;
private ShortBuffer index_buffer;
private Shader m_shader;

public MyRenderer(Context context) {
   c = context;
   x_light_position = 5f;
   y_light_position = 30f;
   z_light_position = 5f;
   model_matrix = new float [16];
   view_matrix = new float [16];
   model_view_matrix = new float [16];
   projection_matrix = new float [16];
   model_view_projection_matrix = new float [16];
   Matrix.setIdentityM(model_matrix, 0);
   x_camera = -3.0f;
   y_camera = 3.0f;
   z_camera = 0.0f;
   Matrix.setLookAtM(view_matrix, 0, x_camera, y_camera, z_camera, -0.5f, 0.2f, 0, 0, 1, 0);
   Matrix.multiplyMM(model_view_matrix, 0, view_matrix, 0, model_matrix, 0);
   x = new float [max_size_x + 1];
   z = new float [max_size_z + 1];
   y = new float [max_size_z + 1][max_size_x + 1];
   vertexes = new float [(max_size_z + 1) * (max_size_x + 1) * 3];
   normales_x = new float [max_size_z + 1][max_size_x + 1];
   normales_y = new float [max_size_z + 1][max_size_x + 1];
   normales_z = new float [max_size_z + 1][max_size_x + 1];
   normales = new float [(max_size_z + 1) * (max_size_x + 1) * 3];
   for (int i = 0; i <= max_size_x; i++) {
      x[i] = x0 + i * dx;
   }
   for (int j = 0; j <= max_size_z; j++) {
      z[j] = z0 + j * dz;
   }
   ByteBuffer vb = ByteBuffer.allocateDirect((max_size_z + 1) * (max_size_x + 1) * 3 * 4);
   vb.order(ByteOrder.nativeOrder());
   vertexes_buffer = vb.asFloatBuffer();
   vertexes_buffer.position(0);
   ByteBuffer nb = ByteBuffer.allocateDirect((max_size_z + 1) * (max_size_x + 1) * 3 * 4);
   nb.order(ByteOrder.nativeOrder());
   normales_buffer = nb.asFloatBuffer();
   normales_buffer.position(0);
   short[] index;
   size_index = 2 * (max_size_x + 1) * max_size_z + (max_size_z - 1);
   index = new short [size_index];
   int k = 0;
   int j = 0;
```

```
    while (j < max_size_z) {
        for (int i = 0; i <= max_size_x; i++) {
            index[k] = chain(j, i);
            k++;
            index[k] = chain(j + 1, i);
            k++;
        }
        if (j < max_size_z - 1) {
            index[k] = chain(j + 1, max_size_x);
            k++;
        }
        j++;
        if (j < max_size_z) {
            for (int i = max_size_x; i >= 0; i--) {
                index[k] = chain(j, i);
                k++;
                index[k] = chain(j + 1, i);
                k++;
            }
            if (j < max_size_z - 1) {
                index[k] = chain(j + 1,0);
                k++;
            }
            j++;
        }
    }
    ByteBuffer bi = ByteBuffer.allocateDirect(size_index * 2);
    bi.order(ByteOrder.nativeOrder());
    index_buffer = bi.asShortBuffer();
    index_buffer.put(index);
    index_buffer.position(0);
    get_vertexes();
    get_normales();
}

private short chain(int j, int i) {
    return (short) (i + j * (max_size_x + 1));
}

private void get_vertexes() {
    double time = System.currentTimeMillis();
    for (int j = 0; j <= max_size_z; j++) {
        for (int i = 0; i <= max_size_x; i++){
            y[j][i] = 0.02f * (float) Math.cos(0.005 * time + 5 * (z[j] + x[i]));
        }
    }
    int k = 0;
    for (int j = 0; j <= max_size_z; j++) {
        for (int i = 0; i <= max_size_x; i++) {
            vertexes[k] = x[i];
            k++;
            vertexes[k] = y[j][i];
            k++;
            vertexes[k] = z[j];
            k++;
        }
    }
    vertexes_buffer.put(vertexes);
    vertexes_buffer.position(0);
}
```

```
private void get_normales() {
    for (int j = 0; j < max_size_z; j++) {
        for (int i = 0; i < max_size_x; i++) {
            normales_x[j][i] = -(y[j][i+1] - y[j][i]) * dz;
            normales_y[j][i] = dx * dz;
            normales_z[j][i] = -dx * (y[j+1][i] - y[j][i]);
        }
    }
    for (int j = 0; j < max_size_z; j++) {
        normales_x[j][max_size_x] = (y[j][max_size_x - 1] - y[j][max_size_x]) * dz;
        normales_y[j][max_size_x] = dx * dz;
        normales_z[j][max_size_x] = -dx * (y[j + 1][max_size_x] - y[j][max_size_x]);
    }
    for (int i = 0; i < max_size_x; i++) {
        normales_x[max_size_z][i] = -(y[max_size_z][i + 1] - y[max_size_z][i]) * dz;
        normales_y[max_size_z][i] = dx * dz;
        normales_z[max_size_z][i] = dx * (y[max_size_z - 1][i] - y[max_size_z][i]);
    }
    normales_x[max_size_z][max_size_x]= (y[max_size_z][max_size_x - 1] - y[max_size_z][max_size_x]) * dz;
    normales_y[max_size_z][max_size_x] = dx * dz;
    normales_z[max_size_z][max_size_x] = dx * (y[max_size_z - 1][max_size_x] - y[max_size_z][max_size_x]);
    int k = 0;
    for (int j = 0; j <= max_size_z; j++) {
        for (int i = 0; i <= max_size_x; i++) {
            normales[k] = normales_x[j][i];
            k++;
            normales[k] = normales_y[j][i];
            k++;
            normales[k] = normales_z[j][i];
            k++;
        }
    }
    normales_buffer.put(normales);
    normales_buffer.position(0);
}

@SuppressLint("ResourceType")
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    gl.glGenTextures(1, texture, 0);
    InputStream stream;
    Bitmap bitmap;
    stream = c.getResources().openRawResource(R.drawable.water);
    bitmap = BitmapFactory.decodeStream(stream);
    gl.glBindTexture(GL10.GL_TEXTURE_2D, texture[0]);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,
            GL10.GL_LINEAR);
    GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
    GLES20.glEnable(GLES20.GL_DEPTH_TEST);
    String vertex_shader =
            "uniform mat4 u_modelViewProjectionMatrix;"+
                    "attribute vec3 a_vertex;"+
                    "attribute vec3 a_normal;"+
                    "varying vec3 v_vertex;"+
                    "varying vec3 v_normal;"+
                    "void main() {"+
                    "v_vertex = a_vertex;"+
                    "vec3 n_normal = normalize(a_normal);"+
                    "v_normal = n_normal;"+
```

```
                "gl_Position = u_modelViewProjectionMatrix * vec4(a_vertex, 1.0);"+
                "}";

    String fragment_shader =
            "precision mediump float;" +
                "uniform vec3 u_camera;" +
                "uniform vec3 u_lightPosition;" +
                "uniform sampler2D u_texture0;" +
                "varying vec3 v_vertex;" +
                "varying vec3 v_normal;" +
                "vec3 myrefract(vec3 IN, vec3 NORMAL, float k) {" +
                "    float nv = dot(NORMAL,IN);" +
                "    float v2 = dot(IN,IN);" +
                "    float knormal = (sqrt(((k * k - 1.0) * v2) / (nv * nv) + 1.0) - 1.0) * nv;" +
                "    vec3 OUT = IN + (knormal * NORMAL);" +
                "    return OUT;" +
                "}" +
                "void main() {" +
                "    vec3 n_normal = normalize(v_normal);" +
                "    vec3 lightvector = normalize(u_lightPosition - v_vertex);" +
                "    vec3 lookvector = normalize(u_camera - v_vertex);" +
                "    float ambient = 0.1;" +
                "    float k_diffuse = 0.7;" +
                "    float k_specular = 0.3;" +
                "    float diffuse = k_diffuse * max(dot(n_normal, lightvector), 0.0);" +
                "    vec3 reflectvector = reflect(-lightvector, n_normal);" +
                "    float specular = k_specular * pow( max(dot(lookvector,reflectvector),0.0), 40.0);" +
                "    vec4 one = vec4(1.0,1.0,1.0,1.0);" +
                "    vec4 lightColor = (ambient + diffuse + specular) * one;" +
                "    vec3 OUT = myrefract(-lookvector, n_normal, 1.2);" +
                "    float ybottom = -1.0;" +
                "    float xbottom = v_vertex.x + OUT.x * (ybottom - v_vertex.y) / OUT.y;" +
                "    float zbottom = v_vertex.z + OUT.z * (ybottom - v_vertex.y) / OUT.y;" +
                "    vec2 texCoord = vec2(xbottom, zbottom);" +
                "    vec4 textureColor = texture2D(u_texture0, texCoord);" +
                "    gl_FragColor = lightColor * textureColor;" +
                "}";
    m_shader = new Shader(vertex_shader, fragment_shader);
    m_shader.link_vertex_buffer(vertexes_buffer);
    m_shader.link_normal_buffer(normales_buffer);
    m_shader.link_texture(texture);
}

@Override
public void onSurfaceChanged(GL10 gl, int width, int height) {
    gl.glViewport(0, 0, width, height);
    float ratio = (float) width / height;
    float k = 0.055f;
    float left = -k * ratio;
    float right = k * ratio;
    float bottom = -k;
    float top = k;
    float near = 0.1f;
    float far = 10.0f;
    Matrix.frustumM(projection_matrix, 0, left, right, bottom, top, near, far);
    Matrix.multiplyMM(model_view_projection_matrix, 0, projection_matrix, 0, model_view_matrix, 0);
}

@Override
public void onDrawFrame(GL10 gl) {
```

```
        m_shader.link_model_view_projection_matrix(model_view_projection_matrix);
        m_shader.link_camera(x_camera, y_camera, z_camera);
        m_shader.link_light_source(x_light_position, y_light_position, z_light_position);
        get_vertexes();
        get_normales();
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT | GLES20.GL_DEPTH_BUFFER_BIT);
        GLES20.glDrawElements(GLES20.GL_TRIANGLE_STRIP, size_index, GLES20.GL_UNSIGNED_SHORT,
index_buffer);
    }
}
```

Shader.java

```
package com.example.lab5;

import android.opengl.GLES20;
import java.nio.FloatBuffer;

public class Shader {
    private int program_handle;

    public Shader(String vertex_shader, String fragment_shader) {
        create_program(vertex_shader, fragment_shader);
    }

    private void create_program(String vertex_shader, String fragment_shader) {
        int vertex_shader_handle = GLES20.glCreateShader(GLES20.GL_VERTEX_SHADER);
        GLES20.glShaderSource(vertex_shader_handle, vertex_shader);
        GLES20.glCompileShader(vertex_shader_handle);
        int fragment_shader_handle = GLES20.glCreateShader(GLES20.GL_FRAGMENT_SHADER);
        GLES20.glShaderSource(fragment_shader_handle, fragment_shader);
        GLES20.glCompileShader(fragment_shader_handle);
        program_handle = GLES20.glCreateProgram();
        GLES20.glAttachShader(program_handle, vertex_shader_handle);
        GLES20.glAttachShader(program_handle, fragment_shader_handle);
        GLES20.glLinkProgram(program_handle);
    }

    public void link_vertex_buffer(FloatBuffer vertexBuffer) {
        GLES20.glUseProgram(program_handle);
        int a_vertex_handle = GLES20.glGetAttribLocation(program_handle, "a_vertex");
        GLES20.glEnableVertexAttribArray(a_vertex_handle);
        GLES20.glVertexAttribPointer(a_vertex_handle, 3, GLES20.GL_FLOAT, false, 0, vertexBuffer);
    }

    public void link_normal_buffer(FloatBuffer normalBuffer) {
        GLES20.glUseProgram(program_handle);
        int a_normal_handle = GLES20.glGetAttribLocation(program_handle, "a_normal");
        GLES20.glEnableVertexAttribArray(a_normal_handle);
        GLES20.glVertexAttribPointer(a_normal_handle, 3, GLES20.GL_FLOAT, false, 0, normalBuffer);
    }

    public void link_model_view_projection_matrix(float[] modelViewProjectionMatrix) {
        GLES20.glUseProgram(program_handle);
        int u_model_view_projection_matrix_handle = GLES20.glGetUniformLocation(program_handle,
"u_modelViewProjectionMatrix");
        GLES20.glUniformMatrix4fv(u_model_view_projection_matrix_handle, 1, false, modelViewProjectionMatrix, 0);
    }

    public void link_camera(float xCamera, float yCamera, float zCamera) {
```

```
        GLES20.glUseProgram(program_handle);
        int u_camera_handle = GLES20.glGetUniformLocation(program_handle, "u_camera");
        GLES20.glUniform3f(u_camera_handle, xCamera, yCamera, zCamera);
    }

    public void link_light_source(float xLightPosition, float yLightPosition, float zLightPosition) {
        GLES20.glUseProgram(program_handle);
        int u_light_source_handle = GLES20.glGetUniformLocation(program_handle, "u_lightPosition");
        GLES20.glUniform3f(u_light_source_handle, xLightPosition, yLightPosition, zLightPosition);
    }

    public void link_texture(int[] texture) {
        int u_texture_Handle = GLES20.glGetUniformLocation(program_handle, "u_texture0");
        GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texture[0]);
        GLES20.glUniform1i(u_texture_Handle, 0);
    }

    public void use_program() {
        GLES20.glUseProgram(program_handle);
    }

}
```