Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики (ПМ и К)

Курсовая работа

по дисциплине «Программирование для мобильных устройств»

Выполнил: студент факультета ИВТ, группы ИП - 712 Алексеев С.В.

Проверил:

Доцент кафедры ПмиК Нечта И.В.

Новосибирск 2020

# Содержание

[Содержание 2](#_Toc61426830)

[Постановка задачи 3](#_Toc61426831)

[Теоретические сведения 3](#_Toc61426832)

[OpenGL 3](#_Toc61426833)

[OpenGL ES 2.0 3](#_Toc61426834)

[Blender 3](#_Toc61426835)

[OBJ 3](#_Toc61426836)

[Экспорт моделей в OpenGL 3](#_Toc61426837)

[Карта глубины 4](#_Toc61426838)

[Описание основных функций. 4](#_Toc61426839)

[Скриншоты 6](#_Toc61426840)

[Листинг кода 8](#_Toc61426841)

[MainActivity.java: 9](#_Toc61426842)

[Objects.java 9](#_Toc61426843)

[Plane.java 12](#_Toc61426844)

[RenderProgram.java 13](#_Toc61426845)

[ShadowRenderer.java 15](#_Toc61426846)

[Список используемой литературы 21](#_Toc61426847)

# Постановка задачи

Написать программу, в которой нарисован стол на OpenGL ES 2.0. На столе лежат различные фрукты/овощи, стакан с напитком.

# Теоретические сведения

## OpenGL

(Open Graphics Library) — спецификация, определяющая независимый от языка программирования платформонезависимый программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику.

## OpenGL ES 2.0

Был публично выпущен в марте 2007 года. Он примерно основан на OpenGL 2.0, но устраняет большую часть конвейера рендеринга с фиксированными функциями в пользу программируемого, аналогично переходу с OpenGL 3.0 на 3.1. Поток управления в шейдерах обычно ограничивается прямым ветвлением и циклами, где максимальное количество итераций может быть легко определено во время компиляции. Почти все функции рендеринга на этапе преобразования и освещения, такие как спецификация материалов и параметров освещения, ранее задававшаяся API фиксированных функций, заменены шейдерами, написанными графическим программистом.

## Blender

Профессиональное cвободное и открытое программное обеспечение для создания трёхмерной компьютерной графики, включающее в себя средства моделирования, скульптинга, анимации, симуляции, рендеринга, постобработки и монтажа видео со звуком, компоновки с помощью «узлов» (Node Compositing), а также создания 2D-анимаций.

## OBJ

Это простой текстовый формат данных, который представляет только 3D геометрические объекты.

## Экспорт моделей в OpenGL

Представление OBJ содержит геометрические данные для 3D-модели на основе вершин. Эти данные разделены на следующие категории:

Вершина (v): положение вершины в пространстве XYZ.

Координаты текстуры (vt): тексель (элемент текстуры) для выборки в UVпространстве. Вы можете думать об этом как о способе сопоставить каждую вершину с позицией на текстуре, откуда она должна получать значение цвета. Эти значения варьируются от (0, 0) (нижний левый угол текстуры) до (1, 1) (верхний правый угол текстуры).

Нормали (vn): нормаль к поверхности вершинной плоскости (треугольника) в пространстве XYZ. Вы можете думать об этом как о векторе, который указывает «прямо» из передней части плоскости в вершину. Это значение необходимо для обеспечения правильного освещения.

Грани (f): плоский треугольник, определяемый тремя вершинами, координатами текстуры и нормалями.

## Карта глубины

Карта глубины (или «теневая карта», «карта теней») — это текстура глубины, визуализируемая с точки зрения света, которую мы будем использовать для теста теней.

Тень — это отсутствие света. Если лучи от источника света не попадают на объект, так как поглощаются другим объектом, то первый объект находится в тени. Тени добавляют реализма к изображению и дают увидеть взаимное расположение объектов. Благодаря ним сцена приобретает "глубину".

## Описание основных функций.

MainActivity.java - главный класс программы. В нём мы объявляем нашу область для рисования (SurfaceView)

Space.java - класс для отрисовки плоскости. Метод render аналогичен методу в классе Objects. Местоположение вершин, нормалей и цвет определены заранее.

Objects.java - Класс, отвечающий за хранение объекта и его отрисовку. В конструктор получает цвет объекта и название файла типа obj, в котором хранятся данные о нём. Также имеет метод render, который принимает на вход местоположение в шейдере атрибута позиции, нормали и цвета. Также принимает булеву переменную onlyPosition, которое отвечает за то, куда происходит отрисовка – на сцену или же в буфер глубины (в буфер глубины нам нужно только местоположение)

MyRender.java - класс, предназначенный для создания, инициализации и загрузки шейдеров в программу. На вход принимает две строки (либо два ID ресурса), одна из которых является кодом вершинного шейдера, а другая – кодом фрагментного шейдера. Позже с помощью get метода можно получить готовую программу

MyGl20Renderer.java - главный класс для отрисовки наших объектов:

MainActivity mShadowsActivity – главный класс программы. Необходим для того, чтобы изменять отрисовку, в зависимости от выбранного пункта в меню.

MyRender mSimpleShadowProgram – программа для отрисовку простых теней private MyRender mDepthMapProgram - программа для заполнения буфера глубины

Объекты, для отрисовки:

private Objects Table; private Space mSpace; private Objects Teapot; private Objects Apple; private Objects Bottle; private Objects Title; private Objects Stakan;

public void onSurfaceCreated(GL10 unused, EGLConfig config) - Метод, вызываемый при создании пространства для отрисовки. В нём мы инициализируем все объекты и программы.

public void onSurfaceChanged(GL10 unused, int width, int height) - создание GLSurface. Будет вызываться например при смене ориентации экрана и первоначальной загрузки. Нужные параметры - int width, int height, ширина(x) и высота(y) соответственно. public void onDrawFrame(GL10 unused) - Вызывается для отрисовки каждого кадра. В нём мы отрисовываем карту теней и сцену. private void renderShadowMap() - Метод, который генерирует карту теней. private void renderScene() - Метод для отрисовки всех объектов с учётом карты теней.

# Скриншоты







# Листинг кода

## MainActivity.java:

|  |
| --- |
| package com.example.user.newcurswork;  import android.app.Activity; import android.opengl.GLSurfaceView; import android.os.Bundle; import android.view.Menu; import android.view.MenuInflater; import android.view.MenuItem; import android.widget.Toast;  public class MainActivity extends Activity {   @Override  public void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);   GLSurfaceView mGLSurfaceView = new GLSurfaceView(this);   mGLSurfaceView.setEGLContextClientVersion(2);   ShadowsRenderer renderer = new ShadowsRenderer(this, this);  mGLSurfaceView.setRenderer(renderer);   setContentView(mGLSurfaceView);  } } |

## Objects.java

|  |
| --- |
| package com.example.user.newcurswork;  import java.io.IOException; import java.nio.ByteBuffer; import java.nio.ByteOrder; import java.nio.FloatBuffer; import java.nio.ShortBuffer; import java.util.ArrayList; import java.util.List; import java.util.Scanner;  import android.content.Context; import android.opengl.GLES20;  class Objects {  private FloatBuffer colorBuffer;   private FloatBuffer verticesBuffer;  private FloatBuffer normalBuffer;   private ShortBuffer facesVertexBuffer;  private ShortBuffer facesNormalBuffer;   private List<String> facesList;   Objects(Context c, float[] color, String ObjName) {  List<String> verticesList = new ArrayList<>();  facesList = new ArrayList<>();  List<String> normalList = new ArrayList<>();  try {  Scanner scanner = new Scanner(c.getAssets().open(ObjName));  while (scanner.hasNextLine()) {  String line = scanner.nextLine();  if (line.startsWith("v ")) {  verticesList.add(line);  } else if (line.startsWith("f ")) {  facesList.add(line);  } else if (line.startsWith("vn ")) {  normalList.add(line);  } else if (line.startsWith("vt ")) {  continue;  }  }  ByteBuffer buffer1 = ByteBuffer.*allocateDirect*(verticesList.size() \* 3 \* 4);  buffer1.order(ByteOrder.*nativeOrder*());  verticesBuffer = buffer1.asFloatBuffer();   ByteBuffer buffer2 = ByteBuffer.*allocateDirect*(normalList.size() \* 3 \* 4);  buffer2.order(ByteOrder.*nativeOrder*());  normalBuffer = buffer2.asFloatBuffer();   ByteBuffer buffer3 = ByteBuffer.*allocateDirect*(facesList.size() \* 3 \* 2);  buffer3.order(ByteOrder.*nativeOrder*());  facesVertexBuffer = buffer3.asShortBuffer();   ByteBuffer buffer4 = ByteBuffer.*allocateDirect*(facesList.size() \* 3 \* 2);  buffer4.order(ByteOrder.*nativeOrder*());  facesNormalBuffer = buffer4.asShortBuffer();   for (String vertex : verticesList) {  String coords[] = vertex.split(" ");  float x = Float.*parseFloat*(coords[1]);  float y = Float.*parseFloat*(coords[2]);  float z = Float.*parseFloat*(coords[3]);  verticesBuffer.put(x);  verticesBuffer.put(y);  verticesBuffer.put(z);  }  verticesBuffer.position(0);   for (String vertex : normalList) {  String coords[] = vertex.split(" ");  float x = Float.*parseFloat*(coords[1]);  float y = Float.*parseFloat*(coords[2]);  float z = Float.*parseFloat*(coords[3]);  normalBuffer.put(x);  normalBuffer.put(y);  normalBuffer.put(z);  }  normalBuffer.position(0);   for (String face : facesList) {  String vertexIndices[] = face.split(" ");  String coord1[] = vertexIndices[1].split("//");  String coord2[] = vertexIndices[2].split("//");  String coord3[] = vertexIndices[3].split("//");   short vertex1 = Short.*parseShort*(coord1[0]);  short vertex2 = Short.*parseShort*(coord2[0]);  short vertex3 = Short.*parseShort*(coord3[0]);  facesVertexBuffer.put((short) (vertex1 - 1));  facesVertexBuffer.put((short) (vertex2 - 1));  facesVertexBuffer.put((short) (vertex3 - 1));   vertex1 = Short.*parseShort*(coord1[1]);  vertex2 = Short.*parseShort*(coord2[1]);  vertex3 = Short.*parseShort*(coord3[1]);  facesNormalBuffer.put((short) (vertex1 - 1));  facesNormalBuffer.put((short) (vertex2 - 1));  facesNormalBuffer.put((short) (vertex3 - 1));  }  facesVertexBuffer.position(0);  facesNormalBuffer.position(0);   verticesList.clear();  normalList.clear();   scanner.close();  } catch (IOException e) {  e.printStackTrace();  }   float[] colorData = new float[facesList.size() \* 4];  for (int v = 0; v < facesList.size(); v++) {  colorData[4 \* v] = color[0];  colorData[4 \* v + 1] = color[1];  colorData[4 \* v + 2] = color[2];  colorData[4 \* v + 3] = color[3];  }   ByteBuffer bColor = ByteBuffer.*allocateDirect*(colorData.length \* 4);  bColor.order(ByteOrder.*nativeOrder*());  colorBuffer = bColor.asFloatBuffer();  colorBuffer.put(colorData).position(0);  }   void render(int positionAttribute, int normalAttribute, int colorAttribute, boolean onlyPosition) {  facesVertexBuffer.position(0);  facesNormalBuffer.position(0);  verticesBuffer.position(0);  normalBuffer.position(0);  colorBuffer.position(0);   GLES20.*glVertexAttribPointer*(positionAttribute, 3, GLES20.*GL\_FLOAT*, false,  0, verticesBuffer);  GLES20.*glEnableVertexAttribArray*(positionAttribute);   if (!onlyPosition) {  GLES20.*glVertexAttribPointer*(normalAttribute, 3, GLES20.*GL\_FLOAT*, false,  0, normalBuffer);  GLES20.*glEnableVertexAttribArray*(normalAttribute);   GLES20.*glVertexAttribPointer*(colorAttribute, 4, GLES20.*GL\_FLOAT*, false,  0, colorBuffer);  GLES20.*glEnableVertexAttribArray*(colorAttribute);  }   GLES20.*glDrawElements*(GLES20.*GL\_TRIANGLES*, facesList.size() \* 3,  GLES20.*GL\_UNSIGNED\_SHORT*, facesVertexBuffer);  } } |

## Plane.java

package com.example.user.newcurswork;  
  
import java.nio.ByteBuffer;  
import java.nio.ByteOrder;  
import java.nio.FloatBuffer;  
  
import android.opengl.GLES20;  
  
public class Plane {  
 private final FloatBuffer planePosition;  
 private final FloatBuffer planeNormal;  
 private final FloatBuffer planeColor;  
  
 Plane() {  
 float[] planePositionData = {  
 -25.0f, -0.0f, -25.0f,  
 -25.0f, -0.0f, 25.0f,  
 25.0f, -0.0f, -25.0f,  
 -25.0f, -0.0f, 25.0f,  
 25.0f, -0.0f, 25.0f,  
 25.0f, -0.0f, -25.0f  
 };  
 ByteBuffer bPos = ByteBuffer.*allocateDirect*(planePositionData.length \* 4);  
 bPos.order(ByteOrder.*nativeOrder*());  
 planePosition = bPos.asFloatBuffer();  
  
 float[] planeNormalData = {  
 0.0f, 1.0f, 0.0f,  
 0.0f, 1.0f, 0.0f,  
 0.0f, 1.0f, 0.0f,  
 0.0f, 1.0f, 0.0f,  
 0.0f, 1.0f, 0.0f,  
 0.0f, 1.0f, 0.0f  
 };  
 ByteBuffer bNormal = ByteBuffer.*allocateDirect*(planeNormalData.length \* 4);  
 bNormal.order(ByteOrder.*nativeOrder*());  
 planeNormal = bNormal.asFloatBuffer();  
  
 float[] planeColorData = {  
 0.3f, 0.3f, 0.3f, 1.0f,  
 0.3f, 0.3f, 0.3f, 1.0f,  
 0.3f, 0.3f, 0.3f, 1.0f,  
 0.3f, 0.3f, 0.3f, 1.0f,  
 0.3f, 0.3f, 0.3f, 1.0f,  
 0.3f, 0.3f, 0.3f, 1.0f  
 };  
 ByteBuffer bColor = ByteBuffer.*allocateDirect*(planeColorData.length \* 4);  
 bColor.order(ByteOrder.*nativeOrder*());  
 planeColor = bColor.asFloatBuffer();  
  
 planePosition.put(planePositionData).position(0);  
 planeNormal.put(planeNormalData).position(0);  
 planeColor.put(planeColorData).position(0);  
 }  
  
 void render(int positionAttribute, int normalAttribute, int colorAttribute, boolean onlyPosition) {  
 planePosition.position(0);  
 planeNormal.position(0);  
 planeColor.position(0);  
  
 GLES20.*glVertexAttribPointer*(positionAttribute, 3, GLES20.*GL\_FLOAT*, false,  
 0, planePosition);  
 GLES20.*glEnableVertexAttribArray*(positionAttribute);  
  
 if (!onlyPosition) {  
 GLES20.*glVertexAttribPointer*(normalAttribute, 3, GLES20.*GL\_FLOAT*, false,  
 0, planeNormal);  
 GLES20.*glEnableVertexAttribArray*(normalAttribute);  
 GLES20.*glVertexAttribPointer*(colorAttribute, 4, GLES20.*GL\_FLOAT*, false,  
 0, planeColor);  
 GLES20.*glEnableVertexAttribArray*(colorAttribute);  
 }  
  
 GLES20.*glDrawArrays*(GLES20.*GL\_TRIANGLES*, 0, 6);  
 }  
}

## RenderProgram.java

package com.example.user.newcurswork;  
  
import java.io.BufferedReader;  
import java.io.InputStream;  
import java.io.InputStreamReader;  
  
import android.content.Context;  
import android.opengl.GLES20;  
import android.util.Log;  
  
class RenderProgram {  
  
 private int mProgram;  
  
 private String mVertexS, mFragmentS;  
  
 RenderProgram(int vID, int fID, Context context) {  
 StringBuilder vs = new StringBuilder();  
 StringBuilder fs = new StringBuilder();  
  
 try {  
 InputStream inputStream = context.getResources().openRawResource(vID);  
 BufferedReader in = new BufferedReader(new InputStreamReader(inputStream));  
  
 String read = in.readLine();  
 while (read != null) {  
 vs.append(read).append("\n");  
 read = in.readLine();  
 }  
  
 vs.deleteCharAt(vs.length() - 1);  
  
 inputStream = context.getResources().openRawResource(fID);  
 in = new BufferedReader(new InputStreamReader(inputStream));  
  
 read = in.readLine();  
 while (read != null) {  
 fs.append(read).append("\n");  
 read = in.readLine();  
 }  
  
 fs.deleteCharAt(fs.length() - 1);  
 } catch (Exception e) {  
 Log.*d*("RenderProgram", "Could not read shader: " + e.getLocalizedMessage());  
 }  
 setup(vs.toString(), fs.toString());  
 }  
  
 private void setup(String vs, String fs) {  
 this.mVertexS = vs;  
 this.mFragmentS = fs;  
  
 if (createProgram() != 1) {  
 throw new RuntimeException("Error at creating shaders");  
 };  
 }  
  
 private int createProgram() {  
 int mVertexShader = loadShader(GLES20.*GL\_VERTEX\_SHADER*, mVertexS);  
 if (mVertexShader == 0) {  
 return 0;  
 }  
  
 int mPixelShader = loadShader(GLES20.*GL\_FRAGMENT\_SHADER*, mFragmentS);  
 if (mPixelShader == 0) {  
 return 0;  
 }  
  
 mProgram = GLES20.*glCreateProgram*();  
 if (mProgram != 0) {  
 GLES20.*glAttachShader*(mProgram, mVertexShader);  
 GLES20.*glAttachShader*(mProgram, mPixelShader);  
 GLES20.*glLinkProgram*(mProgram);  
 int[] linkStatus = new int[1];  
 GLES20.*glGetProgramiv*(mProgram, GLES20.*GL\_LINK\_STATUS*, linkStatus, 0);  
 if (linkStatus[0] != GLES20.*GL\_TRUE*) {  
 Log.*e*("RenderProgram", "Could not link \_program: ");  
 Log.*e*("RenderProgram", GLES20.*glGetProgramInfoLog*(mProgram));  
 GLES20.*glDeleteProgram*(mProgram);  
 mProgram = 0;  
 return 0;  
 }  
 }  
 else  
 Log.*d*("CreateProgram", "Could not create program");  
  
 return 1;  
 }  
  
 private int loadShader(int shaderType, String source) {  
 int shader = GLES20.*glCreateShader*(shaderType);  
 if (shader != 0) {  
 GLES20.*glShaderSource*(shader, source);  
 GLES20.*glCompileShader*(shader);  
 int[] compiled = new int[1];  
 GLES20.*glGetShaderiv*(shader, GLES20.*GL\_COMPILE\_STATUS*, compiled, 0);  
 if (compiled[0] == 0) {  
 Log.*e*("RenderProgram", "Could not compile shader " + shaderType + ":");  
 Log.*e*("RenderProgram", GLES20.*glGetShaderInfoLog*(shader));  
 GLES20.*glDeleteShader*(shader);  
 shader = 0;  
 }  
 }  
 return shader;  
 }  
  
 int getProgram() {  
 return mProgram;  
 }  
}

## ShadowRenderer.java

package com.example.user.newcurswork;  
  
import javax.microedition.khronos.egl.EGLConfig;  
import javax.microedition.khronos.opengles.GL10;  
  
import android.content.Context;  
import android.opengl.GLES20;  
import android.opengl.GLSurfaceView;  
import android.opengl.Matrix;  
import android.util.Log;  
  
public class ShadowsRenderer implements GLSurfaceView.Renderer {  
  
 private final MainActivity mShadowsActivity;  
  
 private RenderProgram mSimpleShadowProgram;  
  
 private RenderProgram mDepthMapProgram;  
  
 private int mActiveProgram;  
  
 private final float[] mMVPMatrix = new float[16];  
 private final float[] mMVMatrix = new float[16];  
 private final float[] mNormalMatrix = new float[16];  
 private final float[] mProjectionMatrix = new float[16];  
 private final float[] mViewMatrix = new float[16];  
 private final float[] mModelMatrix = new float[16];  
  
 private final float[] mLightMvpMatrix = new float[16];  
  
 private final float[] mLightProjectionMatrix = new float[16];  
  
 private final float[] mLightViewMatrix = new float[16];  
  
 private final float[] mLightPosInEyeSpace = new float[16];  
  
 private final float[] mLightPosModel = new float[]  
 {0.1f, 10.0f, 0.1f, 1.0f};  
 private float[] mActualLightPosition = new float[4];  
  
 private int mDisplayWidth;  
 private int mDisplayHeight;  
 private float s = 0;  
  
 private int mShadowMapWidth;  
 private int mShadowMapHeight;  
  
 private int[] fboId;  
 private int[] renderTextureId;  
  
 private int scene\_mvpMatrixUniform;  
 private int scene\_mvMatrixUniform;  
 private int scene\_normalMatrixUniform;  
 private int scene\_lightPosUniform;  
 private int scene\_shadowProjMatrixUniform;  
 private int scene\_textureUniform;  
 private int scene\_mapStepXUniform;  
 private int scene\_mapStepYUniform;  
  
 private int shadow\_mvpMatrixUniform;  
  
 private int scene\_positionAttribute;  
 private int scene\_normalAttribute;  
 private int scene\_colorAttribute;  
  
 private int shadow\_positionAttribute;  
  
 private Objects Table;  
  
 private Context c;  
  
 private Plane mPlane;  
 private Objects Teapot;  
 private Objects Torch;  
 private Objects Apple;  
 private Objects Cup;  
 private Objects Ufo;  
 private Objects Mug;  
 private Objects Title;  
 private Objects Mug2;  
 private Objects blackChess;  
 private Objects whiteChess;  
  
 ShadowsRenderer(final MainActivity shadowsActivity, Context c) {  
 mShadowsActivity = shadowsActivity;  
 this.c = c;  
 }  
  
 @Override  
 public void onSurfaceCreated(GL10 unused, EGLConfig config) {  
 GLES20.*glClearColor*(1.0f, 1.0f, 1.0f, 1.0f);  
  
 GLES20.*glEnable*(GLES20.*GL\_DEPTH\_TEST*);  
  
 GLES20.*glEnable*(GLES20.*GL\_CULL\_FACE*);  
  
 /\* Table = new Objects(c, new float[]{0.6f, 0.3f, 0.2f, 1.0f}, "sasha\_table.obj");  
 Teapot = new Objects(c, new float[]{0.5f, 0.5f, 0.6f, 1.0f}, "sasha\_tarelka.obj");  
 Torch = new Objects(c, new float[]{0.8f, 0.6f, 0.3f, 1.0f}, "sasha\_candle.obj");  
 Apple = new Objects(c, new float[]{0.9f, 0.2f, 0.2f, 1.0f}, "sasha\_apple.obj");  
 Cup = new Objects(c, new float[]{0.0f, 0.5f, 0.0f, 1.0f}, "sasha\_bottle.obj");  
 Title = new Objects(c, new float[]{0f, 0f, 0f, 1.0f}, "sasha\_name.obj");\*/  
  
  
 Table = new Objects(c, new float[]{0.6f, 0.3f, 0.2f, 1.0f}, "Table.obj");  
 Teapot = new Objects(c, new float[]{0.5f, 0.5f, 0.6f, 1.0f}, "teapot.obj");  
 Torch = new Objects(c, new float[]{0.8f, 0.6f, 0.3f, 1.0f}, "torch.obj");  
 Apple = new Objects(c, new float[]{0.9f, 0.2f, 0.2f, 1.0f}, "apple.obj");  
 Cup = new Objects(c, new float[]{0.4f, 0.2f, 0.3f, 1.0f}, "cup.obj");  
 Title = new Objects(c, new float[]{0f, 0f, 0f, 1.0f}, "Title.obj");  
  
 mPlane = new Plane();  
  
 mSimpleShadowProgram = new RenderProgram(R.raw.*depth\_tex\_v\_with\_shadow*, R.raw.*depth\_tex\_f\_with\_simple\_shadow*, mShadowsActivity);  
 mDepthMapProgram = new RenderProgram(R.raw.*depth\_tex\_v\_depth\_map*, R.raw.*depth\_tex\_f\_depth\_map*, mShadowsActivity);  
 mActiveProgram = mSimpleShadowProgram.getProgram();  
 }  
  
 private void generateShadowFBO() {  
 mShadowMapWidth = Math.*round*(mDisplayWidth);  
 mShadowMapHeight = Math.*round*(mDisplayHeight);  
  
 fboId = new int[1];  
 int[] depthTextureId = new int[1];  
 renderTextureId = new int[1];  
  
 GLES20.*glGenFramebuffers*(1, fboId, 0);  
  
 GLES20.*glGenRenderbuffers*(1, depthTextureId, 0);  
 GLES20.*glBindRenderbuffer*(GLES20.*GL\_RENDERBUFFER*, depthTextureId[0]);  
 GLES20.*glRenderbufferStorage*(GLES20.*GL\_RENDERBUFFER*, GLES20.*GL\_DEPTH\_COMPONENT16*, mShadowMapWidth, mShadowMapHeight);  
  
 GLES20.*glGenTextures*(1, renderTextureId, 0);  
 GLES20.*glBindTexture*(GLES20.*GL\_TEXTURE\_2D*, renderTextureId[0]);  
  
 GLES20.*glTexParameteri*(GLES20.*GL\_TEXTURE\_2D*, GLES20.*GL\_TEXTURE\_MIN\_FILTER*, GLES20.*GL\_NEAREST*);  
 GLES20.*glTexParameteri*(GLES20.*GL\_TEXTURE\_2D*, GLES20.*GL\_TEXTURE\_MAG\_FILTER*, GLES20.*GL\_NEAREST*);  
 GLES20.*glTexParameteri*(GLES20.*GL\_TEXTURE\_2D*, GLES20.*GL\_TEXTURE\_WRAP\_S*, GLES20.*GL\_CLAMP\_TO\_EDGE*);  
 GLES20.*glTexParameteri*(GLES20.*GL\_TEXTURE\_2D*, GLES20.*GL\_TEXTURE\_WRAP\_T*, GLES20.*GL\_CLAMP\_TO\_EDGE*);  
  
 GLES20.*glBindFramebuffer*(GLES20.*GL\_FRAMEBUFFER*, fboId[0]);  
  
 GLES20.*glTexImage2D*(GLES20.*GL\_TEXTURE\_2D*, 0, GLES20.*GL\_DEPTH\_COMPONENT*, mShadowMapWidth, mShadowMapHeight, 0, GLES20.*GL\_DEPTH\_COMPONENT*, GLES20.*GL\_UNSIGNED\_INT*, null);  
 GLES20.*glFramebufferTexture2D*(GLES20.*GL\_FRAMEBUFFER*, GLES20.*GL\_DEPTH\_ATTACHMENT*, GLES20.*GL\_TEXTURE\_2D*, renderTextureId[0], 0);  
  
   
 }  
  
 @Override  
 public void onSurfaceChanged(GL10 unused, int width, int height) {  
 mDisplayWidth = width;  
 mDisplayHeight = height;  
 GLES20.*glViewport*(0, 0, mDisplayWidth, mDisplayHeight);  
  
 generateShadowFBO();  
  
 float ratio = (float) mDisplayWidth / mDisplayHeight;  
 float bottom = -1.0f;  
 float top = 1.0f;  
 float near = 1.0f;  
 float far = 100.0f;  
  
 Matrix.*frustumM*(mProjectionMatrix, 0, -ratio, ratio, bottom, top, near, far);  
 Matrix.*frustumM*(mLightProjectionMatrix, 0, -1.1f \* ratio, 1.1f \* ratio, 1.1f \* bottom, 1.1f \* top, near, far);  
 }  
  
 @Override  
 public void onDrawFrame(GL10 unused) {  
 mActiveProgram = mSimpleShadowProgram.getProgram();  
  
 Matrix.*setLookAtM*(mViewMatrix, 0,  
 5, 4, 0,  
 0, 0, 0,  
 -1,0,0);  
  
 scene\_mvpMatrixUniform = GLES20.*glGetUniformLocation*(mActiveProgram, "uMVPMatrix");  
 scene\_mvMatrixUniform = GLES20.*glGetUniformLocation*(mActiveProgram, "uMVMatrix");  
 scene\_normalMatrixUniform = GLES20.*glGetUniformLocation*(mActiveProgram, "uNormalMatrix");  
 scene\_lightPosUniform = GLES20.*glGetUniformLocation*(mActiveProgram, "uLightPos");  
 scene\_shadowProjMatrixUniform = GLES20.*glGetUniformLocation*(mActiveProgram, "uShadowProjMatrix");  
 scene\_textureUniform = GLES20.*glGetUniformLocation*(mActiveProgram, "uShadowTexture");  
 scene\_positionAttribute = GLES20.*glGetAttribLocation*(mActiveProgram, "aPosition");  
 scene\_normalAttribute = GLES20.*glGetAttribLocation*(mActiveProgram, "aNormal");  
 scene\_colorAttribute = GLES20.*glGetAttribLocation*(mActiveProgram, "aColor");  
 scene\_mapStepXUniform = GLES20.*glGetUniformLocation*(mActiveProgram, "uxPixelOffset");  
 scene\_mapStepYUniform = GLES20.*glGetUniformLocation*(mActiveProgram, "uyPixelOffset");  
  
 int shadowMapProgram = mDepthMapProgram.getProgram();  
 shadow\_mvpMatrixUniform = GLES20.*glGetUniformLocation*(shadowMapProgram, "uMVPMatrix");  
 shadow\_positionAttribute = GLES20.*glGetAttribLocation*(shadowMapProgram, "aShadowPosition");  
  
 float[] basicMatrix = new float[16];  
  
 Matrix.*setIdentityM*(basicMatrix, 0);  
 Matrix.*multiplyMV*(mActualLightPosition, 0, basicMatrix, 0, mLightPosModel, 0);  
  
 Matrix.*setIdentityM*(mModelMatrix, 0);  
  
 Matrix.*setLookAtM*(mLightViewMatrix, 0,  
 mActualLightPosition[0], mActualLightPosition[1], mActualLightPosition[2],  
 mActualLightPosition[0], -mActualLightPosition[1], mActualLightPosition[2],  
 -mActualLightPosition[0], 0, -mActualLightPosition[2]);  
  
 GLES20.*glCullFace*(GLES20.*GL\_FRONT*);  
  
 s+=0.3f;  
 if (s >= 360) s-=360;  
 Matrix.*rotateM*(mModelMatrix, 0, s, 0,1,0);  
  
 renderShadowMap();  
  
 GLES20.*glCullFace*(GLES20.*GL\_BACK*);  
  
 renderScene();  
 }  
  
 private void renderShadowMap() {  
 GLES20.*glBindFramebuffer*(GLES20.*GL\_FRAMEBUFFER*, fboId[0]);  
  
 GLES20.*glViewport*(0, 0, mShadowMapWidth, mShadowMapHeight);  
  
 GLES20.*glClearColor*(1f, 1f, 1f, 1.0f);  
 GLES20.*glClear*(GLES20.*GL\_DEPTH\_BUFFER\_BIT* | GLES20.*GL\_COLOR\_BUFFER\_BIT*);  
  
 GLES20.*glUseProgram*(mDepthMapProgram.getProgram());  
  
 float[] tempResultMatrix = new float[16];  
  
 Matrix.*multiplyMM*(mLightMvpMatrix, 0, mLightViewMatrix, 0, mModelMatrix, 0);  
  
 Matrix.*multiplyMM*(tempResultMatrix, 0, mLightProjectionMatrix, 0, mLightMvpMatrix, 0);  
 System.*arraycopy*(tempResultMatrix, 0, mLightMvpMatrix, 0, 16);  
  
 GLES20.*glUniformMatrix4fv*(shadow\_mvpMatrixUniform, 1, false, mLightMvpMatrix, 0);  
 Table.render(shadow\_positionAttribute, 0, 0, true);  
 Teapot.render(shadow\_positionAttribute, 0, 0, true);  
 Cup.render(shadow\_positionAttribute, 0, 0, true);  
 Torch.render(shadow\_positionAttribute, 0, 0, true);  
 Apple.render(shadow\_positionAttribute, 0, 0, true);  
 Title.render(shadow\_positionAttribute, 0, 0, true);  
  
 }  
  
 private void renderScene() {  
 GLES20.*glBindFramebuffer*(GLES20.*GL\_FRAMEBUFFER*, 0);  
  
 GLES20.*glClear*(GLES20.*GL\_COLOR\_BUFFER\_BIT* | GLES20.*GL\_DEPTH\_BUFFER\_BIT*);  
  
 GLES20.*glUseProgram*(mActiveProgram);  
  
 GLES20.*glViewport*(0, 0, mDisplayWidth, mDisplayHeight);  
  
 GLES20.*glUniform1f*(scene\_mapStepXUniform, (float) (1.0 / mShadowMapWidth));  
 GLES20.*glUniform1f*(scene\_mapStepYUniform, (float) (1.0 / mShadowMapHeight));  
  
 float[] tempResultMatrix = new float[16];  
  
 float bias[] = new float[]{  
 0.5f, 0.0f, 0.0f, 0.0f,  
 0.0f, 0.5f, 0.0f, 0.0f,  
 0.0f, 0.0f, 0.5f, 0.0f,  
 0.5f, 0.5f, 0.5f, 1.0f};  
  
 float[] depthBiasMVP = new float[16];  
  
 Matrix.*multiplyMM*(tempResultMatrix, 0, mViewMatrix, 0, mModelMatrix, 0);  
 System.*arraycopy*(tempResultMatrix, 0, mMVMatrix, 0, 16);  
  
 GLES20.*glUniformMatrix4fv*(scene\_mvMatrixUniform, 1, false, mMVMatrix, 0);  
  
 Matrix.*invertM*(tempResultMatrix, 0, mMVMatrix, 0);  
 Matrix.*transposeM*(mNormalMatrix, 0, tempResultMatrix, 0);  
  
 GLES20.*glUniformMatrix4fv*(scene\_normalMatrixUniform, 1, false, mNormalMatrix, 0);  
  
 Matrix.*multiplyMM*(tempResultMatrix, 0, mProjectionMatrix, 0, mMVMatrix, 0);  
 System.*arraycopy*(tempResultMatrix, 0, mMVPMatrix, 0, 16);  
  
 GLES20.*glUniformMatrix4fv*(scene\_mvpMatrixUniform, 1, false, mMVPMatrix, 0);  
  
 Matrix.*multiplyMV*(mLightPosInEyeSpace, 0, mViewMatrix, 0, mActualLightPosition, 0);  
  
 GLES20.*glUniform3f*(scene\_lightPosUniform, mLightPosInEyeSpace[0], mLightPosInEyeSpace[1], mLightPosInEyeSpace[2]);  
  
 Matrix.*multiplyMM*(depthBiasMVP, 0, bias, 0, mLightMvpMatrix, 0);  
 System.*arraycopy*(depthBiasMVP, 0, mLightMvpMatrix, 0, 16);  
  
 GLES20.*glUniformMatrix4fv*(scene\_shadowProjMatrixUniform, 1, false, mLightMvpMatrix, 0);  
  
 Table.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
 Teapot.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
 Cup.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
 Apple.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
 Torch.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
 Title.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
 Apple.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
  
 mPlane.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
 }  
}

# Список используемой литературы

1. OPENGL. ТРЕХМЕРНАЯ ГРАФИКА И ЯЗЫК

ПРОГРАММИРОВАНИЯ ШЕЙДЕРОВ [Электронный ресурс]

(01.12.2020) URL: https://www.opengl.org.ru/opengl-trekhmernayagrafika-i-yazyk-programmirovaniya-sheiderov/opengl-trekhmernayagrafika-i-yazyk-programmirovaniya-sheiderov-page-0.html

1. OpenGL Red Book (русская версия) [Электронный ресурс] (05.12.2020)

URL:https://www.hardforum.ru/download/RedBook.pdf

1. Основы Blender 2.8+[Электронный ресурс] (03.12.2020) URL: https://blender3d.com.ua/blender-basics/