Федеральное агентство связи (Россвязь)

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Курсовая работа

по дисциплине «Программирование для мобильных устройств»

Выполнил:

студент гр. ИП-711 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Гофарова Я.Е. /

подпись

Проверил:

Доцент кафедры ПМиК

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / И. В. Нечта /

ОЦЕНКА, подпись

Оглавление

[Постановка задачи 3](#_Toc59875309)

[Демонстрация работы программы 4](#_Toc59875310)

[Исходный код 5](#_Toc59875311)

[MainActivity 5](#_Toc59875312)

[RenderProgram 5](#_Toc59875313)

[Plane 7](#_Toc59875314)

[Objects 9](#_Toc59875315)

[ShadowsRenderer 11](#_Toc59875316)

[Переменные 11](#_Toc59875317)

[Методы 12](#_Toc59875318)

[Порядок вызова функций в классе ShadowsRenderer 19](#_Toc59875319)

[Шейдеры 19](#_Toc59875320)

[depth\_tex\_v\_depth\_map.glsl 19](#_Toc59875321)

[depth\_tex\_f\_depth\_map 19](#_Toc59875322)

[depth\_tex\_f\_with\_pcf\_shadow.glsl 20](#_Toc59875323)

[depth\_tex\_f\_with\_pcf\_shadow\_dynamic\_bias.glsl 20](#_Toc59875324)

[depth\_tex\_f\_with\_simple\_shadow.glsl 21](#_Toc59875325)

[depth\_tex\_f\_with\_simple\_shadow\_dynamic\_bias.glsl 22](#_Toc59875326)

[depth\_tex\_v\_with\_shadow.glsl 22](#_Toc59875327)

Постановка задачи

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). Объекты отбрасывают тень.

Демонстрация работы программы

|  |  |
| --- | --- |
| https://sun9-3.userapi.com/impg/yaBQMKsIziNbKCyEZszOnHBtIDSfmsEMM_FZ4w/qN8Ox3mO3SE.jpg?size=305x504&quality=96&proxy=1&sign=d51300e849bc4161ea1ca75a84ae5c21&type=album | https://sun9-12.userapi.com/impg/Fdbe6U3SKLppdw7IuRYFEQzX5BPxdC4cgm4QbQ/2pHwLsGvhI4.jpg?size=305x483&quality=96&proxy=1&sign=38bb355db1554b14ac4b980305e5e5e4&type=album |

Исходный код

# MainActivity

Главный класс программы.

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);  
 }  
}

# RenderProgram

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.

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;  
 }  
}

# Plane

Класс для отрисовки плоскости. Метод render аналогичен методу в классе Objects. Местоположение вершин, нормалей и цвет определены заранее.

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 = {  
 1.0f, 0.3f, 0.3f, 1.0f,  
 1.0f, 0.3f, 0.3f, 1.0f,  
 1.0f, 0.3f, 0.3f, 1.0f,  
 1.0f, 0.3f, 0.3f, 1.0f,  
 1.0f, 0.3f, 0.3f, 1.0f,  
 1.0f, 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);  
 }  
}

# Objects

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).

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);  
 }  
}

# ShadowsRenderer

Главный класс для отрисовки, отдельно о нём будет написано далее.

## Переменные

* MainActivity mShadowsActivity – главный класс программы. Необходим для того, чтобы изменять отрисовку, в зависимости от выбранного пункта в меню.
* RenderProgram mSimpleShadowProgram – программа для отрисовку простых теней.
* RenderProgram mPCFShadowProgram – программа для отрисовку теней PCF.
* RenderProgram mSimpleShadowDynamicBiasProgram – программа для отрисовки простых теней с динамичным сдвигом карты теней.
* RenderProgram mPCFShadowDynamicBiasProgram – программа для отрисовки PCF теней с динамичным сдвигом карты теней.
* RenderProgram mDepthMapProgram – программа для заполнения буфера глубины.
* Int mActiveProgram – программа, которая используется в данный момент
* float[] mMVPMatrix – модель-видовая матрица проекции
* float[] mMVMatrix – модель-видовая матрица
* float[] mNormalMatrix – матрица нормалей
* float[] mProjectionMatrix – матрица проекции
* float[] mViewMatrix – видовая матрица
* float[] mModelMatrix – матрица моделей
* float[] mLightMvpMatrix – модель-видовая матрица проекции для отрисовки от источника света
* float[] mLightProjectionMatrix – матрица проекции для отрисовки от источника света
* float[] mLightViewMatrix - видовая матрица для отрисовки от источника света
* float[] mLightPosInEyeSpace – положение источника света с учётом видовой матрицы
* float[] mLightPosModel – положение источника света в пространстве
* int mDisplayWidth, int mDisplayHeight – размер экрана
* int mShadowMapWidth, int mShadowMapHeight – размер карты теней
* int[] fboId - фреймбуфер
* int[] depthTextureId – текстурный буфер для глубины
* int[] renderTextureId – буфер текстур для теней
* Все переменные в этом пункте отвечают за местоположение одноименной переменной в шейдере:

int scene\_mvpMatrixUniform;

int scene\_mvMatrixUniform;

int scene\_normalMatrixUniform;

int scene\_lightPosUniform;

int scene\_schadowProjMatrixUniform;

int scene\_textureUniform;

int scene\_mapStepXUniform;

int scene\_mapStepYUniform;

int shadow\_mvpMatrixUniform;

int scene\_positionAttribute;

int scene\_normalAttribute;

int scene\_colorAttribute;

int shadow\_positionAttribute;

* Все переменные в этом пункте – объекты для отрисовки:

Objects apple;

Objects banan;

Objects candle;

Objects morkov;

Objects stakan;

Objects table;

Plane mPlane;

## Методы

* ShadowsRenderer(final MainActivity shadowsActivity, Context c)

Конструктор, принимает главный класс и контекст.

* public void onSurfaceCreated(GL10 unused, EGLConfig config)

Метод, вызываемый при создании пространства для отрисовки. В нём мы инициализируем все объекты и программы.

* void generateShadowFBO()

Метод, в котором происходит генерация всех буферов.

* public void onSurfaceChanged(GL10 unused, int width, int height)

Метод, который вызывается при изменении размера пространства для отрисовки. Вызывается минимум один раз (при создании пространства для отрисовки). В нём мы задаём размеры этого пространства.

* 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.

* private void setRenderProgram()

Метод, который меняет программу для отрисовки в зависимости от выбранного пункта в меню.

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}, "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");  
  
  
 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);  
  
  
 }  
  
 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);  
 Apple.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
  
 mPlane.render(scene\_positionAttribute, scene\_normalAttribute, scene\_colorAttribute, false);  
 }  
}

# Порядок вызова функций в классе ShadowsRenderer

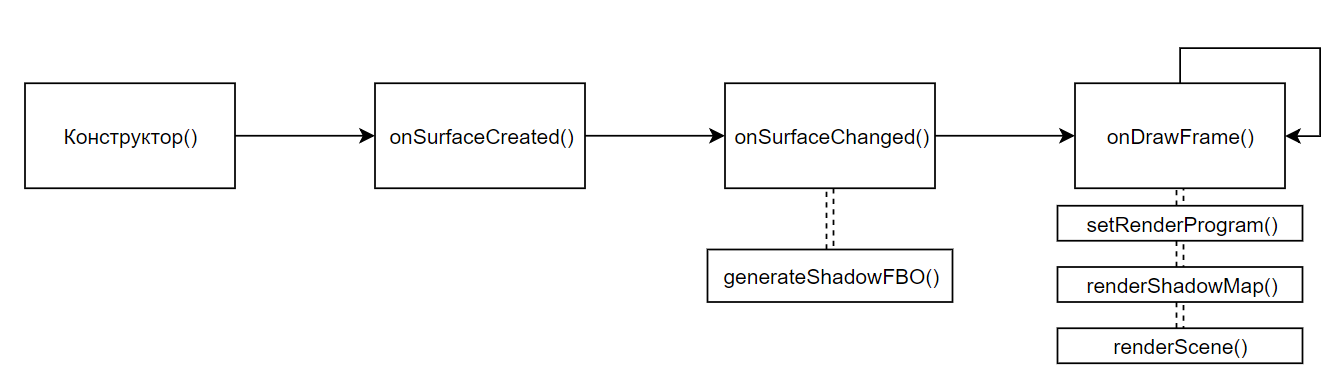


Рисунок 1. Порядок вызова функций. Пунктирными линиями связаны функции, которые вызываются последовательно в главных методах OpenGL

1. Сначала вызывается конструктор. В нём мы указываем наш главный класс и контекст. Это необходимо для доступа к ресурсам программы (в нашем случае – файлы шейдеров и объектов).
2. Вызывается onSurfaceCreated(). Устанавливаем цвет для очистки, включаем тест глубины и активируем устранение спрятанных поверхностей. Инициализируем объекты, видовую матрицу и все шейдеры.
3. Вызывается onSurfaceChanged(). Получаем размеры пространства для отрисовки и на основе этих данных генерируем все буферы в функции generateShadowFBO(). После все этого инициализируем Матрицу проекции и матрицу проекции для источника света.
4. После всех инициализаций циклично вызывается функция onDrawFrame(). Сначала с помощью функции setRenderProgram() устанавливаем программу для отрисовки (меняется с помощью пунктов выпадающего меню). После получаем местоположение всех переменных в выбранных шейдерах.  
   Инициализируем все матрицы так, будто мы смотрим на объекты с позиции источника света, устанавливаем устранение поверхностей в режим “FRONT” (устранение близлежащих к камере поверхностей, необходимо для того, что бы тени не накладывались на сам объект с передней стороны) и вызываем функцию renderShadowMap(). В ней мы отрисовываем все объекты в буфер глубины.  
   После создания карты теней мы устанавливаем устранение поверхностей в режим “BACK” (удаление не видимых нам поверхностей) и вызываем функцию renderScene(), в которой, непосредственно, и происходит отрисовки всех объектов.

# Шейдеры

## depth\_tex\_v\_depth\_map.glsl

precision highp float;

uniform mat4 uMVPMatrix;

attribute vec4 aShadowPosition;

void main() {

gl\_Position = uMVPMatrix \* aShadowPosition;

}

## depth\_tex\_f\_depth\_map

precision highp float;

void main() {

}

## depth\_tex\_f\_with\_pcf\_shadow.glsl

precision mediump float;

uniform vec3 uLightPos;

uniform sampler2D uShadowTexture;

uniform float uxPixelOffset;

uniform float uyPixelOffset;

varying vec3 vPosition;

varying vec4 vColor;

varying vec3 vNormal;

varying vec4 vShadowCoord;

float lookup( vec2 offSet) {

vec4 shadowMapPosition = vShadowCoord / vShadowCoord.w;

float distanceFromLight = texture2D(uShadowTexture, (shadowMapPosition + vec4(offSet.x \* uxPixelOffset, offSet.y \* uyPixelOffset, 0.05, 0.0)).st ).z;

float bias = 0.0005;

return float(distanceFromLight > shadowMapPosition.z - bias);

}

float shadowPCF(){

float shadow = 1.0;

for (float y = -1.5; y <= 1.5; y = y + 1.0) {

for (float x = -1.5; x <= 1.5; x = x + 1.0) {

shadow += lookup(vec2(x,y));

}

}

shadow /= 16.0;

shadow += 0.2;

return shadow;

}

void main(){

vec3 lightVec = uLightPos - vPosition;

lightVec = normalize(lightVec);

float diffuseComponent = max(0.0,dot(lightVec, vNormal) );

float ambientComponent = 0.3;

float shadow = 1.0;

if (vShadowCoord.w > 0.0) {

shadow = shadowPCF();

shadow = (shadow \* 0.8) + 0.2;

}

gl\_FragColor = (vColor \* (diffuseComponent + ambientComponent) \* shadow);

}

## depth\_tex\_f\_with\_pcf\_shadow\_dynamic\_bias.glsl

precision mediump float;

uniform vec3 uLightPos;

uniform sampler2D uShadowTexture;

uniform float uxPixelOffset;

uniform float uyPixelOffset;

varying vec3 vPosition;

varying vec4 vColor;

varying vec3 vNormal;

varying vec4 vShadowCoord;

float calcBias(){

float bias;

vec3 n = normalize( vNormal );

vec3 l = normalize( uLightPos );

float cosTheta = clamp( dot( n,l ), 0.0, 1.0 );

bias = 0.0001\*tan(acos(cosTheta));

bias = clamp(bias, 0.0, 0.01);

return bias;

}

float lookup( vec2 offSet){

vec4 shadowMapPosition = vShadowCoord / vShadowCoord.w;

float distanceFromLight = texture2D(uShadowTexture, (shadowMapPosition + vec4(offSet.x \* uxPixelOffset, offSet.y \* uyPixelOffset, 0.05, 0.0)).st ).z;

float bias = calcBias();

return float(distanceFromLight > shadowMapPosition.z - bias);

}

float shadowPCF(){

float shadow = 1.0;

for (float y = -1.5; y <= 1.5; y = y + 1.0) {

for (float x = -1.5; x <= 1.5; x = x + 1.0) {

shadow += lookup(vec2(x,y));

}

}

shadow /= 16.0;

shadow += 0.2;

return shadow;

}

void main(){

vec3 lightVec = uLightPos - vPosition;

lightVec = normalize(lightVec);

float diffuseComponent = max(0.0,dot(lightVec, vNormal) );

float ambientComponent = 0.3;

float shadow = 1.0;

if (vShadowCoord.w > 0.0) {

shadow = shadowPCF();

shadow = (shadow \* 0.8) + 0.2;

}

gl\_FragColor = (vColor \* (diffuseComponent + ambientComponent) \* shadow);

}

## depth\_tex\_f\_with\_simple\_shadow.glsl

precision mediump float;

uniform vec3 uLightPos;

uniform sampler2D uShadowTexture;

uniform float uxPixelOffset;

uniform float uyPixelOffset;

varying vec3 vPosition;

varying vec4 vColor;

varying vec3 vNormal;

varying vec4 vShadowCoord;

float shadowSimple(){

vec4 shadowMapPosition = vShadowCoord / vShadowCoord.w;

float distanceFromLight = texture2D(uShadowTexture, shadowMapPosition.st).z;

float bias = 0.0005;

return float(distanceFromLight > shadowMapPosition.z - bias);

}

void main() {

vec3 lightVec = uLightPos - vPosition;

lightVec = normalize(lightVec);

float diffuseComponent = max(0.0,dot(lightVec, vNormal) );

float ambientComponent = 0.3;

float shadow = 1.0;

if (vShadowCoord.w > 0.0) {

shadow = shadowSimple();

shadow = (shadow \* 0.8) + 0.2;

}

gl\_FragColor = (vColor \* (diffuseComponent + ambientComponent) \* shadow);

}

## depth\_tex\_f\_with\_simple\_shadow\_dynamic\_bias.glsl

precision mediump float;

uniform vec3 uLightPos;

uniform sampler2D uShadowTexture;

uniform float uxPixelOffset;

uniform float uyPixelOffset;

varying vec3 vPosition;

varying vec4 vColor;

varying vec3 vNormal;

varying vec4 vShadowCoord;

float calcBias(){

float bias;

vec3 n = normalize( vNormal );

vec3 l = normalize( uLightPos );

float cosTheta = clamp( dot( n,l ), 0.0, 1.0 );

bias = 0.0001\*tan(acos(cosTheta));

bias = clamp(bias, 0.0, 0.01);

return bias;

}

float shadowSimple(){

vec4 shadowMapPosition = vShadowCoord / vShadowCoord.w;

float distanceFromLight = texture2D(uShadowTexture, shadowMapPosition.st).z;

float bias = calcBias();

return float(distanceFromLight > shadowMapPosition.z - bias);

}

void main(){

vec3 lightVec = uLightPos - vPosition;

lightVec = normalize(lightVec);

float diffuseComponent = max(0.0,dot(lightVec, vNormal) );

float ambientComponent = 0.3;

float shadow = 1.0;

if (vShadowCoord.w > 0.0) {

shadow = shadowSimple();

shadow = (shadow \* 0.8) + 0.2;

}

gl\_FragColor = (vColor \* (diffuseComponent + ambientComponent) \* shadow);

}

## depth\_tex\_v\_with\_shadow.glsl

uniform mat4 uMVPMatrix;

uniform mat4 uMVMatrix;

uniform mat4 uNormalMatrix;

uniform mat4 uShadowProjMatrix;

attribute vec4 aPosition;

attribute vec4 aColor;

attribute vec3 aNormal;

varying vec3 vPosition;

varying vec4 vColor;

varying vec3 vNormal;

varying vec4 vShadowCoord;

void main() {

vPosition = vec3(uMVMatrix \* aPosition);

vColor = aColor;

vNormal = vec3(uNormalMatrix \* vec4(aNormal, 0.0));

vShadowCoord = uShadowProjMatrix \* aPosition;

gl\_Position = uMVPMatrix \* aPosition;

}