# МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

# федеральное государственное автономное образовательное учреждение высшего образования

# «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

# АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА №43

ОТЧЕТ

ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподователь |  |  |  | С.В. Щекин |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №1-8

КОМПЬЮТЕРНАЯ ГРАФИКА

по курсу: КОМПЬЮТЕРНАЯ ГРАФИКА

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | | 4131 |  |  |  | С.М. Огвоздин |
|  | номер группы | |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург

2022

**Цель работы:**

1. Запустить среду разработки, скомпилировать любое приложение с вызовами библиотеки Open GL, при необходимости настроить работу с библиотеками и заголовочными файлами
2. Создать приложение, выводящее объемный объект средствами Open GL
3. Включить источник света, задать отражающие свойства поверхностей: диффузное, зеркальное отражение, цвет поверхности. Вывести несколько объемных объектов. Каждый из них должен иметь различные свойства поверхностей (доминирующее зеркальное или диффузное отражение, различный цвет внешних поверхностей).
4. Включить 3 источника света, задать отражающие свойства поверхностей, положение и цвет источников света. Рекомендуется цвет поверхностей сделать одинаковым, а отражающие свойства – разными. Вывести несколько объемных объектов. Источники света должны иметь различный цвет.
5. Вывести несколько пересекающихся объектов. Включить 3 источника света, задать свойства поверхностей и источников. Организовать раздельное вращение источников света вокруг неподвижной сцены, управляемое с клавиатуры или мышью.
6. Вывести сложный трехмерный объект, сохранить его в виде списка и размножить в виде нескольких копий различного масштаба на экране путем повторного вывода списка. Наложить текстуру на выводимые объекты, сцена должна быть освещена.
7. Вывести трехмерную сцену с движущимся объектом, который отбрасывает тень на другой объект (напр. плоскость). Тень должна перемещаться вместе с движением объекта исходя из взаимного положения источника света, объекта, который отбрасывает тень и объектов, на которые тень проецируется
8. Выполнить на выбор один из вариантов: - Шейдер с сохранением данных в текстуре, - Процедурный, текстурный шейдер, - Традиционные шейдер, - Фрагментный шейдер, - Вершинный шейдер.

**Описание программы**

Программа использует отложенный рендеринг. Он делится на 3 этапа:

1. **Генерация G-буфера**. Сцена отрисовывается с позиции камеры. Данные о позиции поверхностей, их нормалях, а также диффузной и отражающей способностях сохраняются в текстуры, прикрепленные к G-буферу – кадровому буферу.
2. **Генерация карты теней.** Генерируется карта теней для направленного источника света с использованием метода Variance shadow mapping(Подробнее об этой технике можно прочитать [здесь](https://developer.nvidia.com/gpugems/gpugems3/part-ii-light-and-shadows/chapter-8-summed-area-variance-shadow-maps))
3. **Наложение направленного света**. На сцену накладывается освещение направленного света на сцену, используя данные из G-буфера и карты теней, сгенерированной ранее.
4. **Наложение точечных источников света**. На сцену накладывается освещение от точечных источников света. Точечные источники света рисуются поверх сцены с помощью сфер, освещение использует данные из G-буфера для расчета силы освещения поверхности.
5. **Пост-обработка.** После полной отрисовки сцены накладывается пост-обработка. В случае этой программы – гамма-коррекция.

**Описание класса Texture**

Текстура – класс-обертка над текстурой в OpenGL. Текстуру можно загрузить из стороннего файла с помощью метода load, принимающего в качестве параметра путь к файлу.

Текстуры в программе поддерживают multisampling, если задать тип текстуры GL\_TEXTURE\_2D\_MULTISAMPLE

**Файл Texture.h:**

#pragma once

#include <GL/glew.h>

#include <glm/glm.hpp>

#include "Asset.h"

class Texture : public IAsset

{

public:

Texture(GLenum target = GL\_TEXTURE\_2D, GLenum intenationalformat = GL\_RGBA, GLenum format = GL\_RGBA, GLenum type = GL\_UNSIGNED\_BYTE, glm::vec2 size = glm::vec2(1), bool multisampled = false);

~Texture();

GLuint GL();

GLenum target();

glm::vec2 size();

void resize(glm::vec2 size);

void load(const std::string& filePath) override;

void generateTextureFromData(int width, int height, const unsigned char\* data);

void setBorderColor(glm::vec4 color);

void setWrapMode(GLenum mode);

void genMipmaps();

private:

glm::vec2 m\_size;

GLenum m\_target;

GLenum m\_type;

GLuint m\_texture;

GLenum m\_format;

GLenum m\_internationalformat;

};

**Файл Texture.cpp**

#include "Texture.h"

#include <SOIL/SOIL.h>

#include "GLCheckError.h"

#include <iostream>

Texture::Texture(GLenum target, GLenum internationalformat, GLenum format, GLenum type, glm::vec2 size, bool multisampled)

{

m\_target = target;

m\_size = size;

m\_format = format;

m\_internationalformat = internationalformat;

m\_type = type;

glGenTextures(1, &m\_texture);

glBindTexture(m\_target, m\_texture);

if (m\_target == GL\_TEXTURE\_CUBE\_MAP) {

for (int i = 0; i < 6; i++) {

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, m\_internationalformat, (int)size.x, (int)size.y, 0, m\_format, m\_type, nullptr);

}

}

else if (m\_target == GL\_TEXTURE\_2D\_MULTISAMPLE)

glTexStorage2DMultisample(m\_target, 4, m\_internationalformat, (int)size.x, (int)size.y, GL\_TRUE);

else

glTexImage2D(m\_target, 0, m\_internationalformat, (int)size.x, (int)size.y, 0, m\_format, m\_type, nullptr);

glCheckError();

glTexParameteri(m\_target, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(m\_target, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(m\_target, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(m\_target, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glBindTexture(m\_target, 0);

}

Texture::~Texture()

{

glDeleteTextures(1, &m\_texture);

}

GLuint Texture::GL()

{

return m\_texture;

}

GLenum Texture::target()

{

return m\_target;

}

glm::vec2 Texture::size()

{

return m\_size;

}

void Texture::resize(glm::vec2 size) {

m\_size = size;

glBindTexture(m\_target, m\_texture);

glTexImage2D(m\_target, 0, m\_internationalformat, (int)size.x, (int)size.y, 0, m\_format, m\_type, nullptr);

glBindTexture(m\_target, 0);

}

void Texture::load(const std::string& filePath)

{

std::cout << "Try to load " << filePath << std::endl;

int width, height;

unsigned char\* image = SOIL\_load\_image(filePath.c\_str(), &width, &height, 0, SOIL\_LOAD\_RGBA);

if (sizeof(image) == 0) {

printf("File didnt open");

return;

}

generateTextureFromData(width, height, image);

SOIL\_free\_image\_data(image);

}

void Texture::generateTextureFromData(int width, int height, const unsigned char\* data)

{

m\_type = GL\_UNSIGNED\_BYTE;

m\_size = glm::vec2(width, height);

m\_format = GL\_RGBA;

m\_internationalformat = GL\_RGBA;

m\_target = GL\_TEXTURE\_2D;

if (m\_texture)

glDeleteTextures(1, &m\_texture);

glGenTextures(1, &m\_texture);

glBindTexture(m\_target, m\_texture);

glTexParameteri(m\_target, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(m\_target, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(m\_target, GL\_TEXTURE\_WRAP\_S, GL\_MIRRORED\_REPEAT);

glTexParameteri(m\_target, GL\_TEXTURE\_WRAP\_T, GL\_MIRRORED\_REPEAT);

glTexImage2D(m\_target, 0, m\_internationalformat, width, height, 0, m\_format, m\_type, data);

glGenerateMipmap(m\_target);

glBindTexture(m\_target, 0);

}

void Texture::setBorderColor(glm::vec4 color)

{

glBindTexture(m\_target, m\_texture);

glTexParameterfv(m\_target, GL\_TEXTURE\_BORDER\_COLOR, &(color[0]));

glBindTexture(m\_target, 0);

}

void Texture::setWrapMode(GLenum mode)

{

glBindTexture(m\_target, m\_texture);

glTexParameteri(m\_target, GL\_TEXTURE\_WRAP\_S, mode);

glTexParameteri(m\_target, GL\_TEXTURE\_WRAP\_T, mode);

glBindTexture(m\_target, 0);

}

void Texture::genMipmaps()

{

glBindTexture(m\_target, m\_texture);

glGenerateMipmap(m\_target);

glBindTexture(m\_target, 0);

}

**Описание класса Mesh**

Mеш – класс, хранящий данные о полигональной сетке. Класс содержит vertex buffer object и vertex array object из OpenGL, а также указатель на материал, используемый для отрисовки меша. Меш можно создать из списка вершин. Вершина хранит свою позицию, нормаль и текстурные координаты. В vertex array object хранится привязка атрибутов вершин для передачи в вершинный шейдер. Отрисовка шейдера происходит методом draw, который принимает шейдер для отрисовки материала, матрицу Проекции-вида, и преобразование меша в глобальном пространстве.

**Файл Mesh.h**

#pragma once

#include <GL/glew.h>

#include <glm/glm.hpp>

#include <vector>

#include <utility>

#include "Transform.h"

#include "Material.h"

class IDrawable

{

public:

virtual void draw(Shader\* shader = nullptr, const glm::mat4& ProjectionView = glm::mat4(1), const Transform& transform = Transform()) = 0;

};

struct Vertex {

glm::vec3 pos;

glm::vec3 normal;

glm::vec2 texCoords;

Vertex(glm::vec3 position, glm::vec3 normals, glm::vec2 TexCoords) {

pos = position;

normal = normals;

texCoords = TexCoords;

}

};

class Mesh : public IDrawable

{

public:

Mesh(Material\* material = nullptr);

~Mesh();

void draw(Shader\* shader = nullptr, const glm::mat4& ProjectionView = glm::mat4(1), const Transform& transform = Transform()) override;

void init(std::vector<Vertex>& vertices, std::vector<GLuint>& indices);

void setMaterial(Material\* material) {

if (m\_material)

delete m\_material;

m\_material = material;

}

private:

Material\* m\_material;

std::vector<Vertex> m\_vertices;

std::vector<GLuint> m\_indices;

unsigned int m\_indicesCount;

GLuint VAO;

GLuint EBO;

GLuint VBO;

};

Mesh\* createBox(glm::vec3 size = glm::vec3(1));

**Файл Mesh.cpp**

#include "Mesh.h"

Mesh::Mesh(Material\* material)

{

m\_material = material;

m\_indicesCount = 0;

EBO = 0;

VBO = 0;

VAO = 0;

}

Mesh::~Mesh()

{

if (m\_material)

delete m\_material;

}

void Mesh::draw(Shader\* shader, const glm::mat4& ProjectionView, const Transform& transform)

{

if (shader == nullptr) {

if (m\_material == nullptr) return;

m\_material->mat4Param["ProjectionView"] = ProjectionView;

m\_material->mat4Param["Model"] = transform.matrix();

m\_material->use();

}

else {

glUseProgram(shader->getProgram());

shader->setUniform("ProjectionView", (glm::mat4&)ProjectionView);

glm::mat4 model = transform.matrix();

shader->setUniform("Model", model);

}

glBindVertexArray(VAO);

glDrawElements(GL\_TRIANGLES, m\_indicesCount, GL\_UNSIGNED\_INT, 0);

glBindVertexArray(0);

}

void Mesh::init(std::vector<Vertex>& vertices, std::vector<GLuint>& indices)

{

m\_vertices.swap(vertices);

m\_indices.swap(indices);

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

glGenBuffers(1, &EBO);

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, m\_vertices.size() \* sizeof(Vertex), &(m\_vertices[0]), GL\_STATIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, m\_indices.size() \* sizeof(GLuint), &(m\_indices[0]), GL\_STATIC\_DRAW);

//vertex position

glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)0);

//vertex normals

glEnableVertexAttribArray(1);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)offsetof(Vertex, normal));

// vertex texture coords

glEnableVertexAttribArray(2);

glVertexAttribPointer(2, 2, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)offsetof(Vertex, texCoords));

glBindVertexArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

m\_indicesCount = m\_indices.size();

}

Mesh\* createBox(glm::vec3 size) {

std::vector<Vertex> vertices = {

Vertex(glm::vec3(size.x, size.y, size.z),glm::vec3(1), glm::vec2(0)),

Vertex(glm::vec3(-size.x, size.y, size.z),glm::vec3(1), glm::vec2(0)),

Vertex(glm::vec3(size.x, -size.y, size.z),glm::vec3(1), glm::vec2(0)),

Vertex(glm::vec3(-size.x, -size.y, size.z),glm::vec3(1), glm::vec2(0)),

Vertex(glm::vec3(size.x, size.y, -size.z),glm::vec3(1), glm::vec2(0)),

Vertex(glm::vec3(-size.x, size.y, -size.z),glm::vec3(1), glm::vec2(0)),

Vertex(glm::vec3(size.x, -size.y, -size.z),glm::vec3(1), glm::vec2(0)),

Vertex(glm::vec3(-size.x, -size.y, -size.z),glm::vec3(1), glm::vec2(0)),

};

std::vector<GLuint> indices = {

0, 1, 2,

1, 2, 3,

0, 4, 1,

4, 1, 5,

0, 2, 4,

2, 4, 6,

1, 5, 3,

5, 3, 7,

2, 7, 3,

2, 7, 6,

4, 5, 6,

5, 6, 7,

};

Mesh\* mesh = new Mesh;

mesh->init(vertices, indices);

vertices.clear();

indices.clear();

return mesh;

}

**Описание класса Model**

Модель – список мешей, которые соединены в общий объект. Класс помимо мешей хранит и все текстуры, которые используются для отрисовки модели. Модель можно загрузить из стороннего файл методом load.

**Файл Model.h**

#pragma once

#include "Mesh.h"

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

class Model :

public IAsset, public IDrawable

{

public:

Model();

~Model() override;

void load(const std::string& filename) override;

void draw(Shader\* shader, const glm::mat4& ProjectionView = glm::mat4(1), const Transform& transform = Transform()) override;

Mesh\* getMesh(int index) {

return m\_meshes[index];

}

private:

std::string m\_directory;

std::vector<Mesh\*> m\_meshes;

std::vector<Texture\*> m\_textures;

void processNode(aiNode\* node, const aiScene\* scene);

void processMesh(aiMesh\* mesh, const aiScene\* scene);

bool loadMaterialTexture(Material\* material, aiMaterial\* aiMat, aiTextureType texType, const std::string& uniformName);

};

**Файл Model.cpp**

#include "Model.h"

#include "Application.h"

Model::Model()

{

m\_directory = "";

}

Model::~Model()

{

if (!m\_meshes.empty()) {

for (auto& elem : m\_meshes)

delete elem;

m\_meshes.clear();

}

if (!m\_textures.empty()) {

for (auto& elem : m\_textures)

delete elem;

m\_textures.clear();

}

}

void Model::load(const std::string& filename)

{

m\_directory = filename.substr(0, filename.find\_last\_of('/'));

Assimp::Importer importer;

const aiScene\* scene = importer.ReadFile(filename, aiProcess\_Triangulate | aiProcess\_FlipUVs | aiProcess\_GenNormals);

if (!scene || scene->mFlags & AI\_SCENE\_FLAGS\_INCOMPLETE || !scene->mRootNode) {

std::cout << "Error: " << importer.GetErrorString() << std::endl;

return;

}

processNode(scene->mRootNode, scene);

}

void Model::draw(Shader\* shader, const glm::mat4& ProjectionView, const Transform& transform)

{

for (auto& mesh : m\_meshes) {

mesh->draw(shader, ProjectionView, transform);

}

}

void Model::processNode(aiNode\* node, const aiScene\* scene)

{

for (int i = 0; i < node->mNumMeshes; i++) {

aiMesh\* mesh = scene->mMeshes[node->mMeshes[i]];

processMesh(mesh, scene);

}

for (int i = 0; i < node->mNumChildren; i++) {

processNode(node->mChildren[i], scene);

}

}

void Model::processMesh(aiMesh\* aimesh, const aiScene\* scene)

{

Mesh\* mesh = new Mesh();

std::vector<Vertex> vertices;

std::vector<GLuint> indices;

for (int i = 0; i < aimesh->mNumVertices; i++) {

glm::vec3 pos(aimesh->mVertices[i].x, aimesh->mVertices[i].y, aimesh->mVertices[i].z);

glm::vec3 normal(aimesh->mNormals[i].x, aimesh->mNormals[i].y, aimesh->mNormals[i].z);

glm::vec2 texcoord(0, 0);

if (aimesh->mTextureCoords[0]) {

texcoord = glm::vec2(aimesh->mTextureCoords[0][i].x, aimesh->mTextureCoords[0][i].y);

}

vertices.push\_back(Vertex(pos, normal, texcoord));

}

for (int i = 0; i < aimesh->mNumFaces; i++) {

aiFace face = aimesh->mFaces[i];

for (int j = 0; j < face.mNumIndices; j++)

indices.push\_back(face.mIndices[j]);

}

mesh->init(vertices, indices);

vertices.clear();

indices.clear();

m\_meshes.push\_back(mesh);

if (aimesh->mMaterialIndex >= 0) {

Material\* material = new Material(Loader().getAsset<Shader>("DefaultLit.shader"));

aiMaterial\* mat = scene->mMaterials[aimesh->mMaterialIndex];

if (!loadMaterialTexture(material, mat, aiTextureType\_DIFFUSE, "material.Albedo")) {

aiColor4D albedo;

mat->Get(AI\_MATKEY\_COLOR\_DIFFUSE, albedo);

Texture\* texture = new Texture();

const unsigned char data[4] = {albedo.r \* 255, albedo.g \* 255, albedo.b \* 255, albedo.a \* 255};

texture->generateTextureFromData(1, 1, data);

material->texParam["material.Albedo0"] = texture;

m\_textures.push\_back(texture);

}

if (!loadMaterialTexture(material, mat, aiTextureType\_SPECULAR, "material.Specular")) {

aiColor4D spec;

mat->Get(AI\_MATKEY\_COLOR\_SPECULAR, spec);

Texture\* texture = new Texture();

const unsigned char data[4] = { spec.r \* 255, spec.g \* 255, spec.b \* 255, spec.a \* 255 };

texture->generateTextureFromData(1, 1, data);

material->texParam["material.Specular0"] = texture;

m\_textures.push\_back(texture);

}

float shininess;

mat->Get(AI\_MATKEY\_SHININESS, shininess);

material->floatParam["material.shininess"] = shininess;

std::cout << "Shininess : " << shininess << std::endl;

mesh->setMaterial(material);

}

}

bool Model::loadMaterialTexture(Material\* material, aiMaterial\* aiMat, aiTextureType texType, const std::string& uniformName)

{

if (aiMat->GetTextureCount(texType) == 0) return false;

for (unsigned int i = 0; i < aiMat->GetTextureCount(texType); i++)

{

aiString str;

aiMat->GetTexture(texType, i, &str);

material->texParam[uniformName + std::to\_string(i)] =

Loader().getAsset<Texture>(m\_directory.substr(m\_directory.find\_first\_of('/') + 1, m\_directory.find\_last\_of('/')) + "/" + str.C\_Str());

}

return true;

}

**Описание классов кадровых буферов**

Framebuffer – класс-обертка над framebuffer object в OpenGL. К кадровому буферу можно привязать текстуру для назначение color-, depth- или stencil- компонентов. Также можно привязать Renderbuffer для прикрепления компонента глубины или трафарета. Кадровый буфер можно назначить для отрисовки, тогда все вызовы отрисовки в OpenGL будут производиться не на экран, а в кадровый буфер.

Renderbuffer – класс-обертка над renderbuffer object в OpenGL. Renderbuffer можно привязать к framebuffer для хранения данных о глубине или трафарету. Класс RenderbufferMultiSample поддерживает multisampling.

**Файл Framebuffers.h**

#pragma once

#include <GL/glew.h>

#include <glm/glm.hpp>

#include <vector>

#include "Texture.h"

class Renderbuffer;

class Framebuffer

{

public:

Framebuffer(GLenum read = GL\_FRONT, GLenum draw = GL\_FRONT);

virtual ~Framebuffer();

void clear();

void bind();

void unbind();

void copy(Framebuffer& copyTarget, GLbitfield bitmask);

GLuint GL() {

return m\_buffer;

}

void setClearColor(glm::vec4 color);

//sdfgsdfhg

virtual void attachTexture(Texture\* texture, GLenum attachmentType);

virtual void attachRenderbuffer(Renderbuffer\* renderbuffer);

Texture\* getTextureByAttachment(GLenum attachment) const;

protected:

GLuint m\_buffer = -1;

std::vector<Texture\*> m\_bindTextures;

std::vector<GLuint> m\_attachmentType;

glm::vec4 m\_clearColor = glm::vec4(0);

Renderbuffer\* m\_attachedRenderBuffer;

};

class Renderbuffer {

public:

Renderbuffer(int width, int height);

virtual ~Renderbuffer();

GLuint GL();

glm::vec2 size();

protected:

GLuint m\_buffer;

glm::vec2 m\_size;

};

class RenderbufferMultisample : public Renderbuffer {

public:

RenderbufferMultisample(int width, int height);

~RenderbufferMultisample() override;

};

**Файл Framebuffers.сpp**

#include "Framebuffers.h"

#include <glm/gtc/type\_ptr.hpp>

#include <iostream>

Framebuffer::Framebuffer(GLenum read, GLenum draw)

{

glGenFramebuffers(1, &m\_buffer);

glBindFramebuffer(GL\_FRAMEBUFFER, m\_buffer);

//glReadBuffer(read);

//glDrawBuffer(draw);

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

}

Framebuffer::~Framebuffer()

{

}

void Framebuffer::clear()

{

glBindFramebuffer(GL\_FRAMEBUFFER, m\_buffer);

for (int i = 0; i < m\_bindTextures.size(); i++) {

if (m\_attachmentType[i] >= GL\_COLOR\_ATTACHMENT0)

glClearBufferfv(GL\_COLOR, i, glm::value\_ptr(m\_clearColor));

}

glClear(GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

}

void Framebuffer::bind()

{

if (m\_bindTextures.empty()) return;

glViewport(0, 0, (int)m\_bindTextures[0]->size().x, (int)m\_bindTextures[0]->size().y);

glBindFramebuffer(GL\_FRAMEBUFFER, m\_buffer);

glEnable(GL\_DEPTH\_TEST);

//glEnable(GL\_STENCIL\_TEST);

}

void Framebuffer::unbind()

{

if (m\_bindTextures.empty()) return;

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

glDisable(GL\_DEPTH\_TEST);

//glDisable(GL\_STENCIL\_TEST);

}

void Framebuffer::copy(Framebuffer& copyTarget, GLbitfield bitmask)

{

glBindFramebuffer(GL\_READ\_FRAMEBUFFER, m\_buffer);

glBindFramebuffer(GL\_DRAW\_FRAMEBUFFER, copyTarget.GL());

glBlitFramebuffer(0, 0, (int)m\_bindTextures[0]->size().x, (int)m\_bindTextures[0]->size().y,

0, 0, (int)copyTarget.m\_bindTextures[0]->size().x, (int)copyTarget.m\_bindTextures[0]->size().y,

bitmask,

GL\_NEAREST);

glBindFramebuffer(GL\_READ\_FRAMEBUFFER, 0);

glBindFramebuffer(GL\_DRAW\_FRAMEBUFFER, 0);

}

void Framebuffer::setClearColor(glm::vec4 color)

{

m\_clearColor = color;

clear();

}

void Framebuffer::attachTexture(Texture\* texture, GLenum attachmentType)

{

glBindFramebuffer(GL\_FRAMEBUFFER, m\_buffer);

int i;

for (i = 0; i < m\_bindTextures.size(); i++) {

if (m\_attachmentType[i] == attachmentType) {

m\_bindTextures.erase(m\_bindTextures.begin() + i);

m\_attachmentType.erase(m\_attachmentType.begin() + i);

break;

}

}

m\_bindTextures.push\_back(texture);

m\_attachmentType.push\_back(attachmentType);

glFramebufferTexture(GL\_FRAMEBUFFER, attachmentType, texture->GL(), 0);

glDrawBuffers(m\_attachmentType.size(), &(m\_attachmentType[0]));

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

}

void Framebuffer::attachRenderbuffer(Renderbuffer\* renderbuffer)

{

m\_attachedRenderBuffer = renderbuffer;

glBindFramebuffer(GL\_FRAMEBUFFER, m\_buffer);

glFramebufferRenderbuffer(GL\_FRAMEBUFFER, GL\_DEPTH\_STENCIL\_ATTACHMENT, GL\_RENDERBUFFER, renderbuffer->GL());

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

}

Texture\* Framebuffer::getTextureByAttachment(GLenum attachment) const

{

int i;

for (i = 0; i < m\_attachmentType.size(); i++) {

if (m\_attachmentType[i] == attachment) break;

}

if (i == m\_attachmentType.size()) return nullptr;

return m\_bindTextures[i];

}

Renderbuffer::Renderbuffer(int width, int height)

{

glGenRenderbuffers(1, &m\_buffer);

//std::cout << glCheckError() << std::endl;

glBindRenderbuffer(GL\_RENDERBUFFER, m\_buffer);

//std::cout << glCheckError() << std::endl;

glRenderbufferStorage(GL\_RENDERBUFFER, GL\_DEPTH24\_STENCIL8, width, height);

//std::cout << glCheckError() << std::endl;

glBindRenderbuffer(GL\_RENDERBUFFER, 0);

m\_size = glm::vec2(width, height);

}

Renderbuffer::~Renderbuffer()

{

glDeleteRenderbuffers(1, &m\_buffer);

}

GLuint Renderbuffer::GL()

{

return m\_buffer;

}

glm::vec2 Renderbuffer::size() {

return m\_size;

}

void FramebufferMultiSample::attachTexture(Texture\* texture, GLenum attachmentType)

{

glBindFramebuffer(GL\_FRAMEBUFFER, m\_buffer);

int i;

for (i = 0; i < m\_bindTextures.size(); i++) {

if (m\_attachmentType[i] == attachmentType) {

m\_bindTextures.erase(m\_bindTextures.begin() + i);

m\_attachmentType.erase(m\_attachmentType.begin() + i);

break;

}

}

m\_bindTextures.push\_back(texture);

m\_attachmentType.push\_back(attachmentType);

glBindTexture(texture->target(), texture->GL());

glFramebufferTexture2D(GL\_FRAMEBUFFER, attachmentType, texture->target(), texture->GL(), 0);

glDrawBuffers(m\_attachmentType.size(), &(m\_attachmentType[0]));

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

glBindTexture(texture->target(), 0);

}

RenderbufferMultisample::RenderbufferMultisample(int width, int height) : Renderbuffer(width, height)

{

glDeleteRenderbuffers(1, &m\_buffer);

glGenRenderbuffers(1, &m\_buffer);

//std::cout << glCheckError() << std::endl;

glBindRenderbuffer(GL\_RENDERBUFFER, m\_buffer);

//std::cout << glCheckError() << std::endl;

glRenderbufferStorageMultisample(GL\_RENDERBUFFER, 4, GL\_DEPTH24\_STENCIL8, width, height);

//std::cout << glCheckError() << std::endl;

glBindRenderbuffer(GL\_RENDERBUFFER, 0);

m\_size = glm::vec2(width, height);

}

RenderbufferMultisample::~RenderbufferMultisample()

{

}

**Генерация G-буфера**

К G-буферу подключены 4 текстуры:

1. Позиция – 3-компонентная текстура. Хранится в формате float. Хранит x,y,z координаты поверхности в R, G, B каналах соответственно.
2. Нормаль – 3-компонентная текстура. Хранится в формате float. Хранит направления вектора нормали к поверхности в R, G, B, каналах соответственно
3. Цвет – 4-компонентная текстура. Хранится в формате unsigned byte. Хранит цвет поверхности
4. Отражение – 4-компонентная текстура. Хранится в формате float. Хранит отражающий цвет в R, G, B каналах. Alpha канал используется для хранения степени отражающей способности.

Для генерации G-буфера вся непрозрачная сцена отрисовывается с точки зрения камеры. Используются программируемый вершинный и фрагментный шейдеры (см. Листинг).

**Листинг DefaultVertex.glsl:**

#version 450 core

layout(location = 0) in vec3 position;

layout(location = 1) in vec3 normal;

layout(location = 2) in vec2 texCoords;

uniform mat4 ProjectionView;

uniform mat4 Model;

out vec3 Normal;

out vec2 TexCoords;

out vec3 FragPos;

void main(){

Normal = normalize(mat3(transpose(inverse(Model))) \* normal);

TexCoords = texCoords;

FragPos = vec3(Model \* vec4(position, 1));

gl\_Position = ProjectionView \* Model \* vec4(position, 1);

}

**Листинг DefaultFragment.glsl:**

#version 450 core

layout (location = 0) out vec3 gPosition;

layout (location = 1) out vec3 gNormal;

layout (location = 2) out vec4 gAlbedo;

layout (location = 3) out vec4 gSpec;

struct Material{

sampler2D Albedo0;

sampler2D Specular0;

float shininess;

};

uniform Material material;

in vec2 TexCoords;

in vec3 FragPos;

in vec3 Normal;

void main() {

gPosition = FragPos;

gNormal = Normal;

gAlbedo = texture(material.Albedo0, TexCoords);

gSpec = texture(material.Specular0, TexCoords);

gSpec.a = material.shininess;

}

Ниже приведен отрывок кода, отвечающий за генерацию G-буфера

glm::mat4 pv = camera->getProjectionViewMatrix();

m\_Gbuffer.clear();

m\_Gbuffer.bind();

for (StaticMeshComponent\* mesh : m\_drawables) {

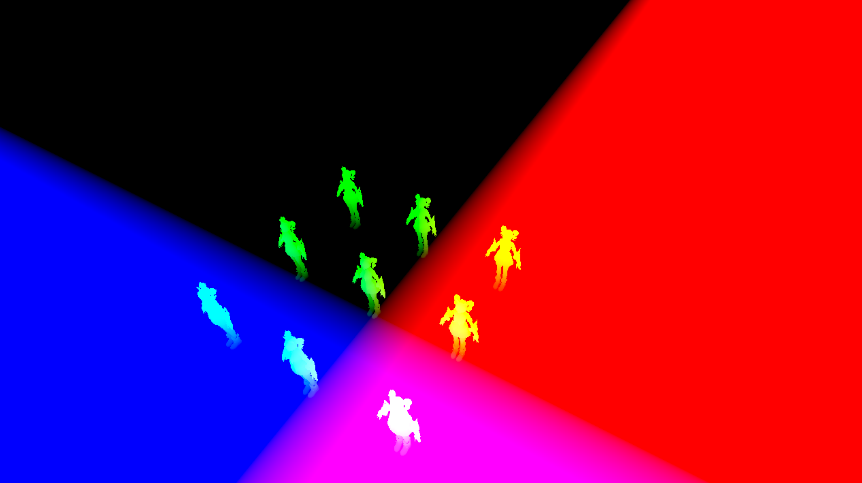
mesh->drawCall(pv);

}

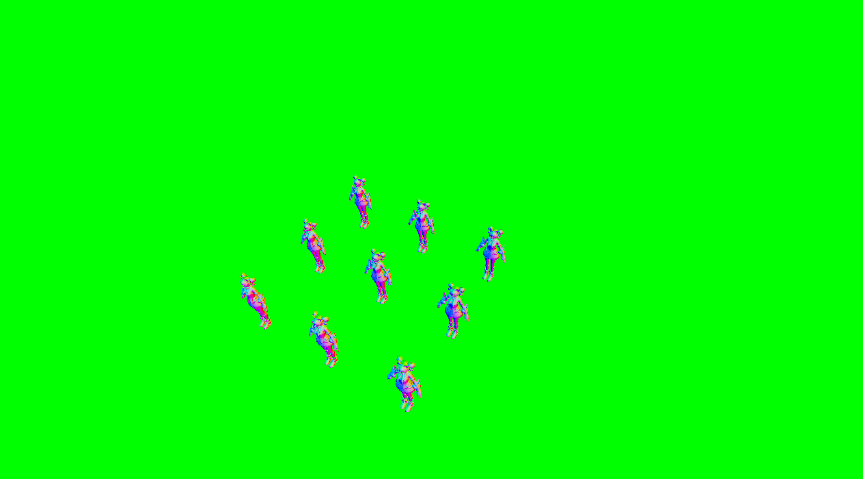
m\_Gbuffer.unbind();

Ниже приведен пример генерации G-буфера (4 текстуры):

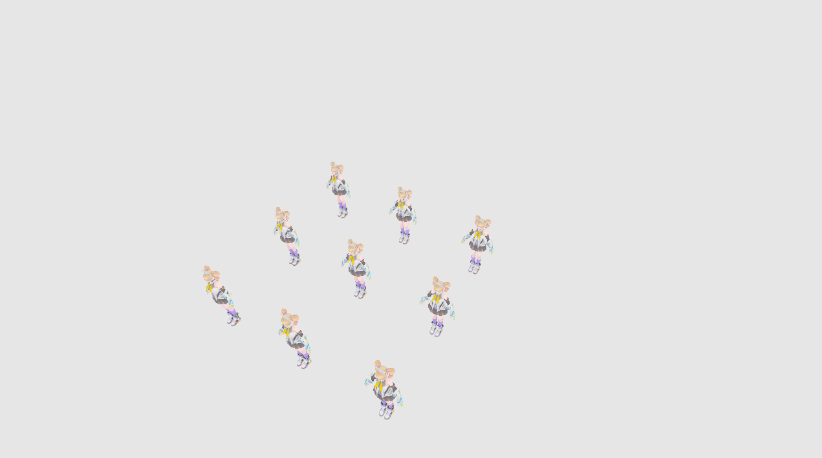
1. Позиция:



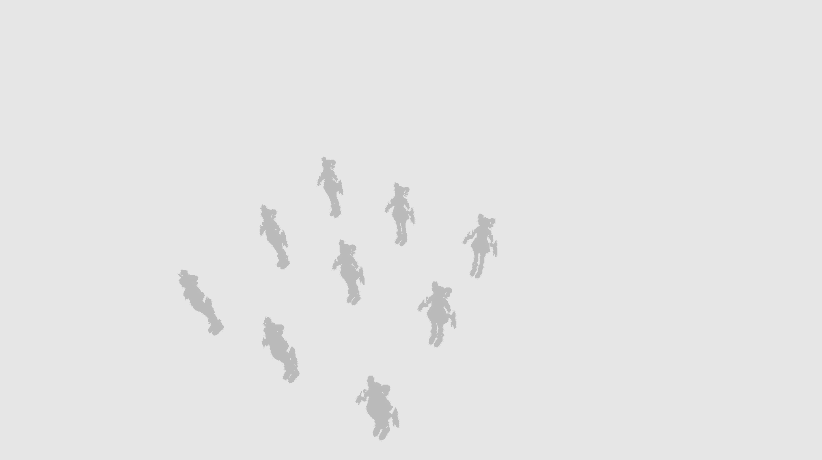
1. Нормали:



1. Цвет



1. Отражение



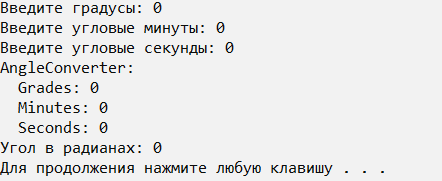
К G-буферу также подключен Renderbuffer, который хранит глубину сцены и используется для теста глубины.

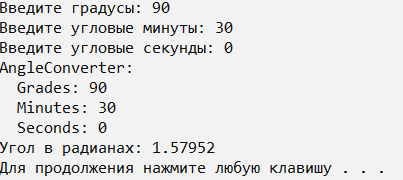
**Генерация карты теней**

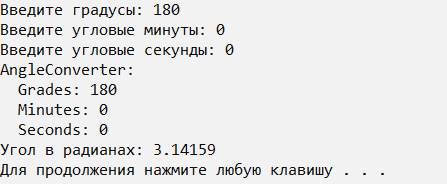
Затенение применяется с использованием технологии Variance Shadow Mapping (Подробнее об этой технике можно прочитать [здесь](https://developer.nvidia.com/gpugems/gpugems3/part-ii-light-and-shadows/chapter-8-summed-area-variance-shadow-maps)).

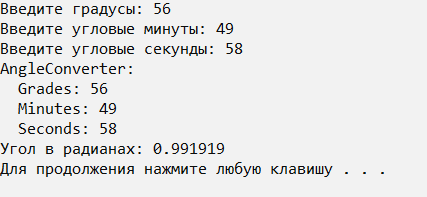
Карта теней хранится в float формате. В R-канал карты теней записывается M1 = d, где d – глубина поверхности. В G-канал карты записывается M2 = d2 + 1/4

**Скриншоты работы программы**









**Выводы**

В ходе лабораторной работы были изучены принципы создания классов, ограничения прав доступа к полям и методам класса, а также применены на практике знания базовых синтаксических конструкций языка C++ и объектно-ориентированного программирования. Закреплены знания по теме: Классы, определение методов, права доступа.