Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное

Образовательное учреждение высшего образования

«Уфимский университет науки и технологий»

Факультет информатики и робототехники

Кафедра Информатики

Отчет по лабораторной работе № 3

на тему: «Управление камерой и освещение в OpenGL»

Выполнил:

Студент группы ПРО-233Б Ермолов И О

Проверил:

канд. техн. наук В.А. Котельников

Уфа - 2023

**Цель работы**

Изучение управления камерой и освещения в OpenGL.

**Задание**

Выполнить 14-21 уроки по OpenGL <https://triplepointfive.github.io/ogltutor/>

**Ход работы**

1. В уроках 14-15 я научился управлять перемещением камеры с помощью клавиатуры и мыши.

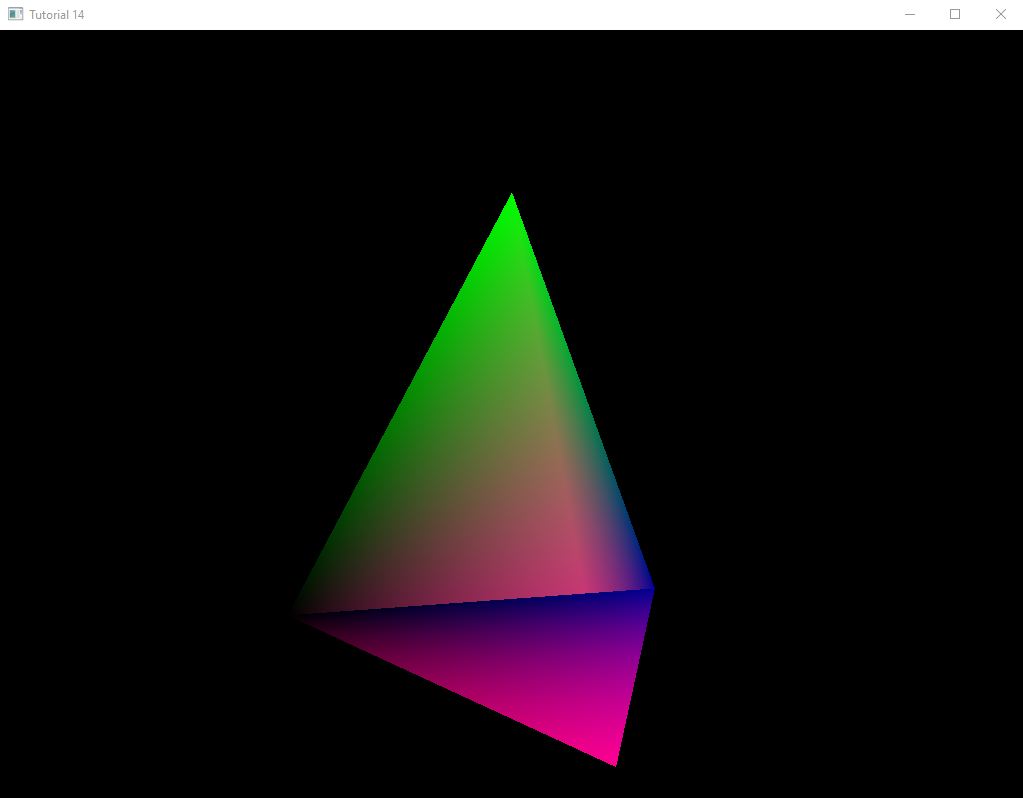


Рисунок 1. Движения камеры стрелками и мышью.

1. В 16 уроке я изучил способ наложения текстур на 3-D объекты в OpenGL с помощью библиотеки Magick.

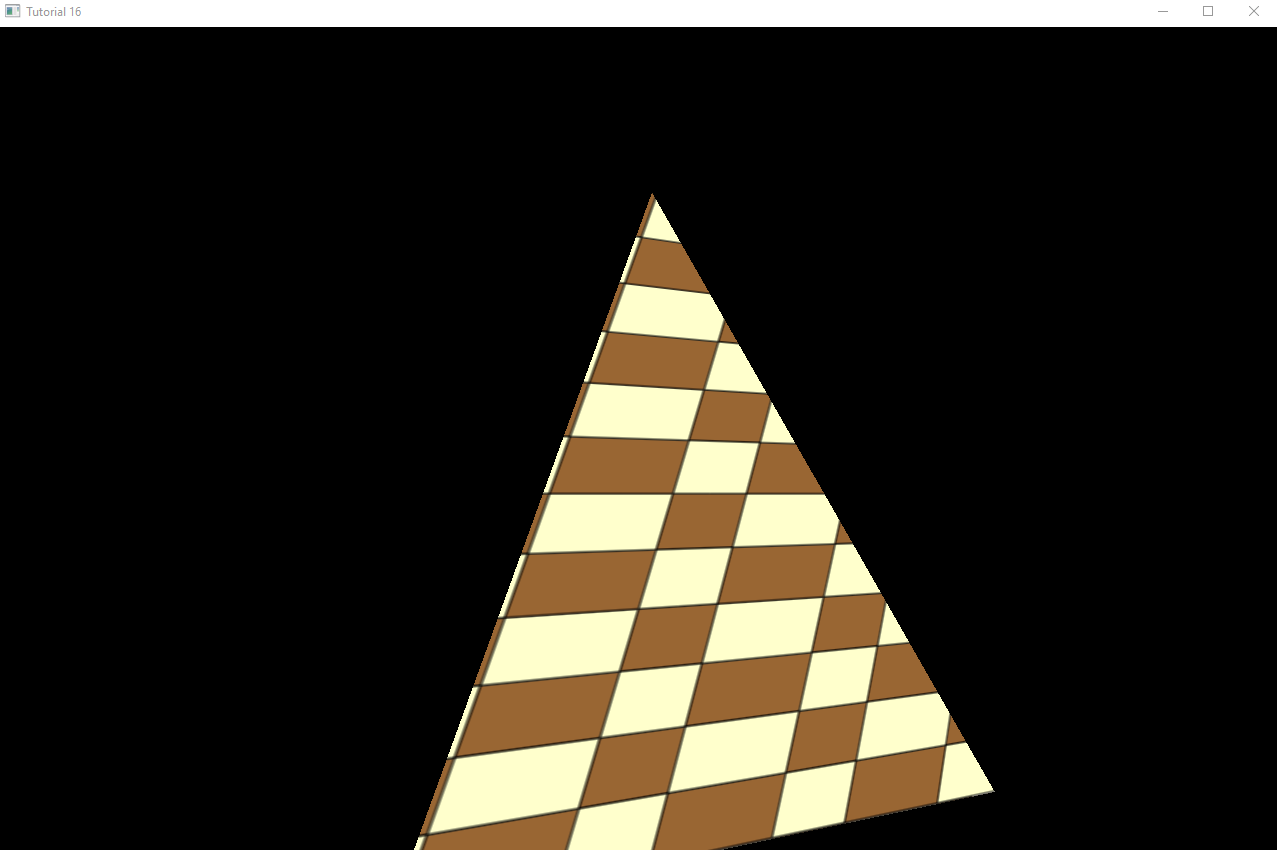


Рисунок 2. Наложение текстур.

1. В уроках 17-19 я изучил основные виды моделей освещения

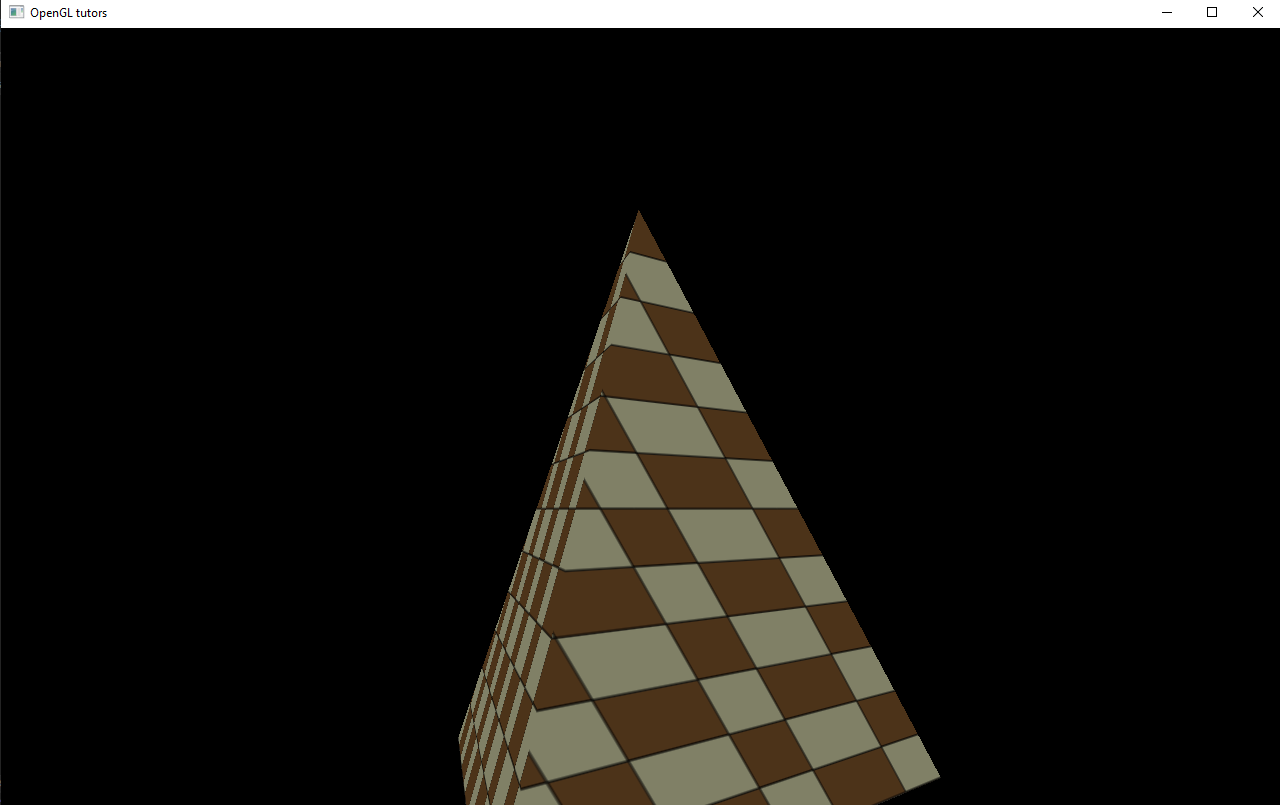


Рисунок 3. Фоновое освещение.

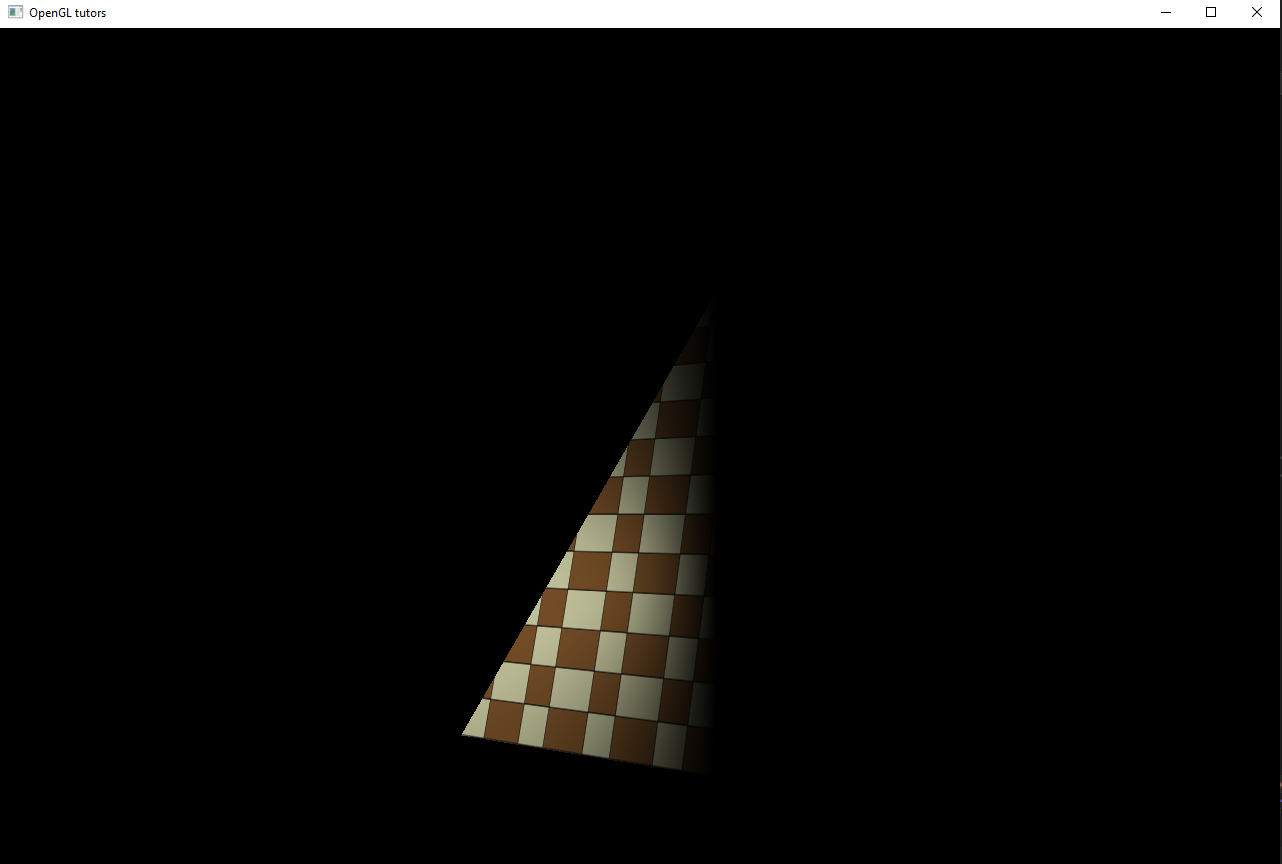


Рисунок 4. Рассеяное освещение.

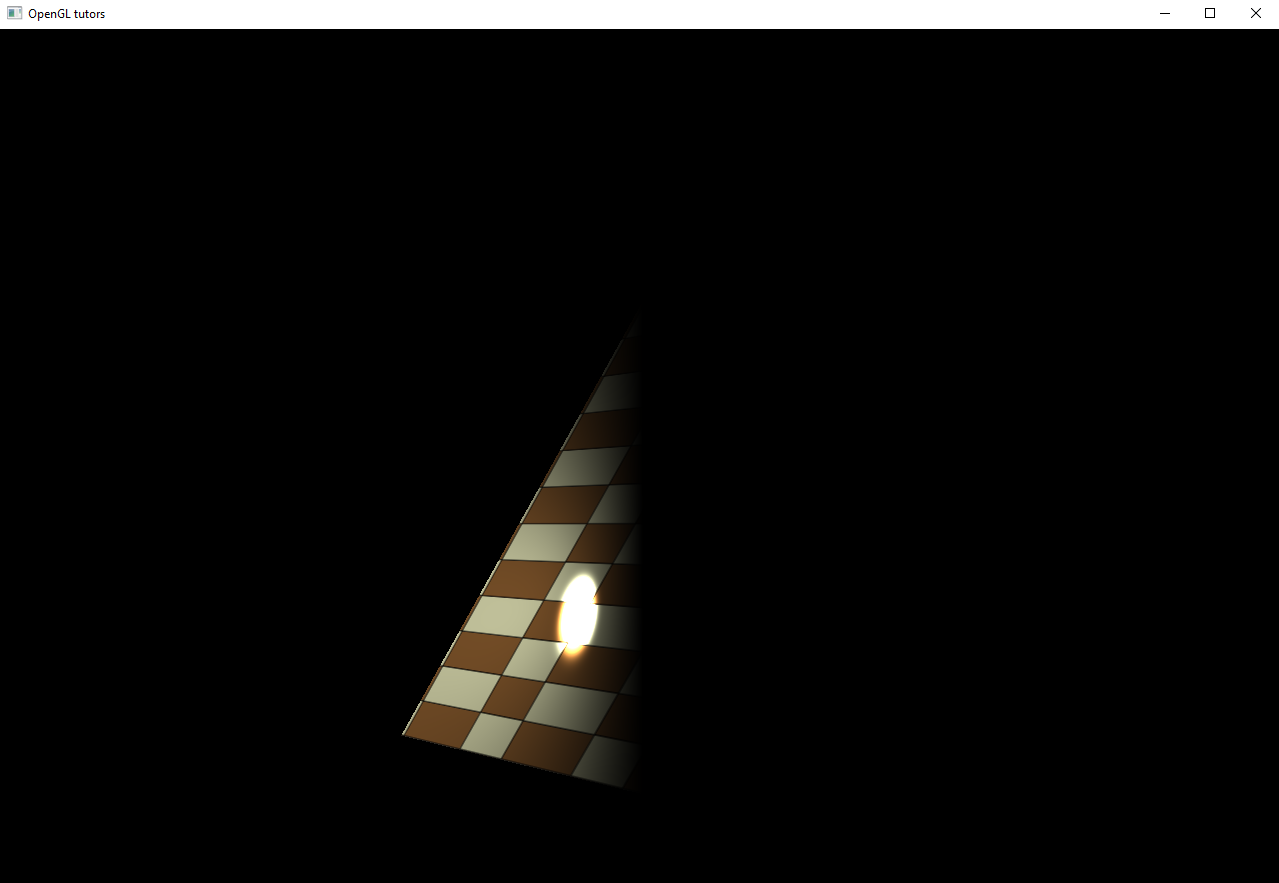


Рисунок 5. Отраженный свет.

1. В уроках 21-22 я изучил способ создания точечных источников света и прожектора.

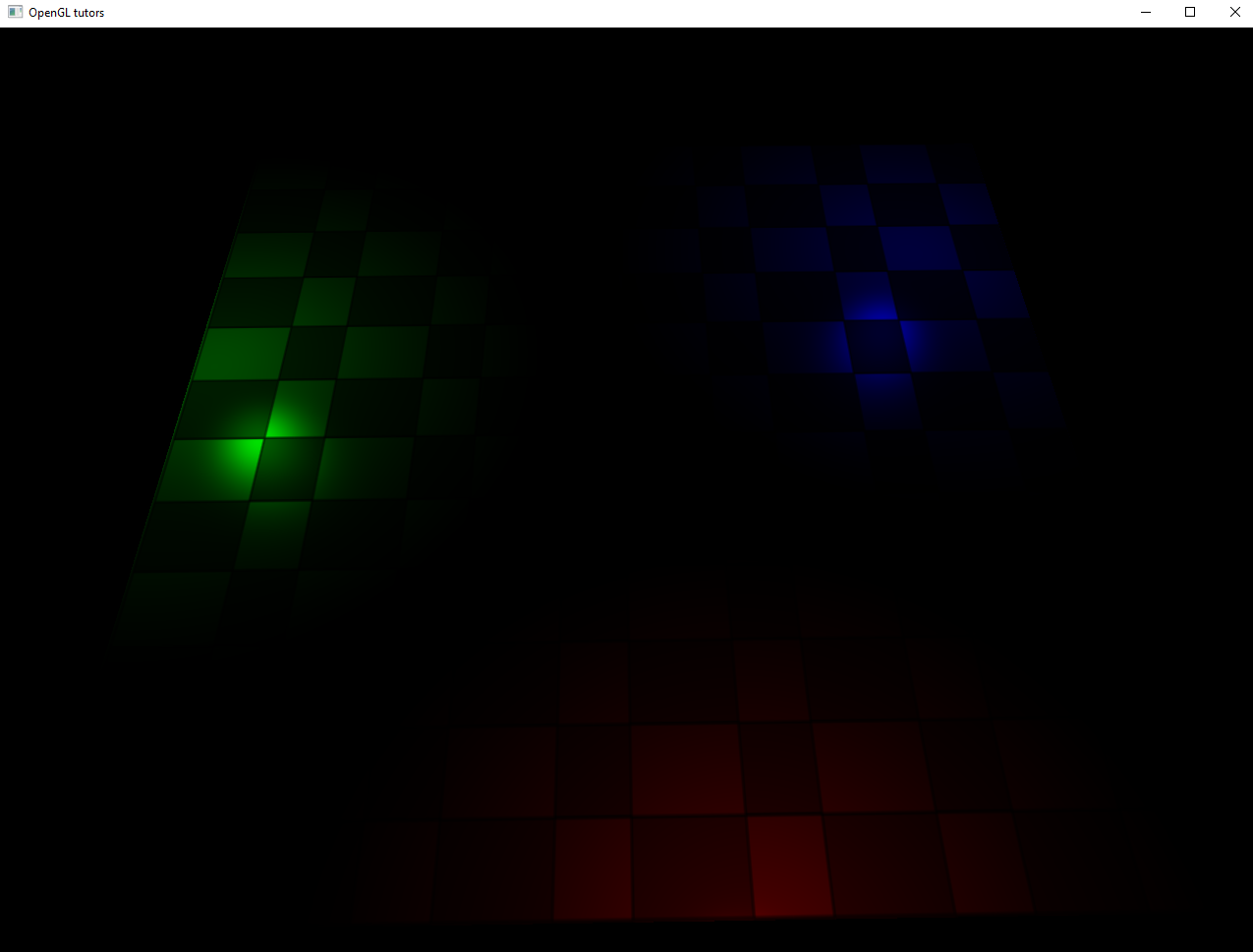


Рисунок 6. Точечные источники света.

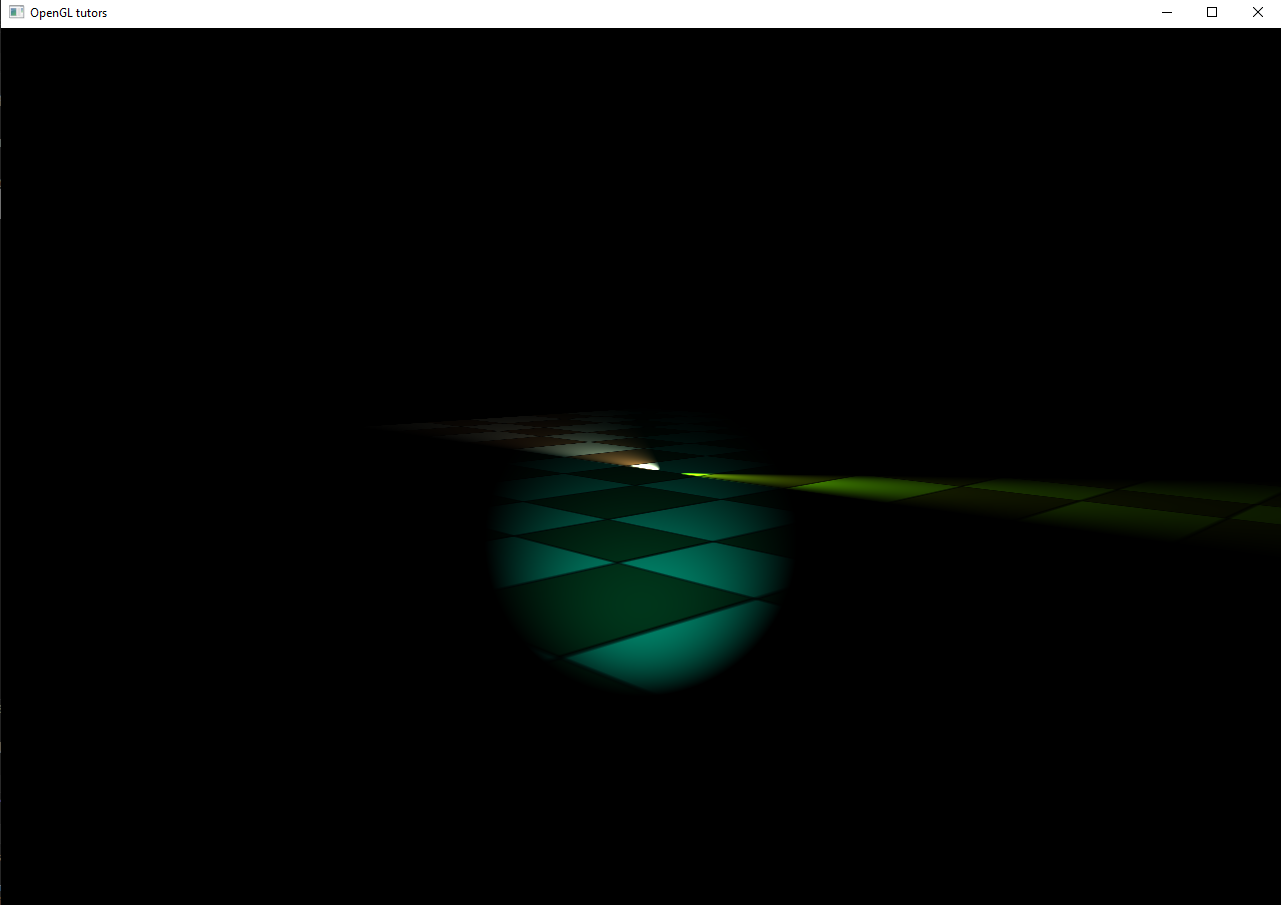


Рисунок 7. Прожекторы.

**Выводы к работе**

При выполнении данной лабораторной работы я познакомился со способами управления камерой и способами освещения. Полный код программ по ссылке: [https://github.com/Ivan122727/EKG/tree/master/LW3](https://github.com/VerVyle/LW_ECG/tree/master/LW3)

**Приложение 1**

**callbacks.h**

#ifndef CALLBACKS\_H

#define CALLBACKS\_H

class ICallbacks

{

public:

virtual void SpecialKeyboardCB(int Key, int x, int y) = 0;

virtual void KeyboardCB(unsigned char Key, int x, int y) = 0;

virtual void PassiveMouseCB(int x, int y) = 0;

virtual void RenderSceneCB() = 0;

virtual void IdleCB() = 0;

};

#endif /\* CALLBACKS\_H \*/

**camera.cpp**

#include <GL/freeglut.h>

#include "camera.h"

const static float STEP\_SCALE = 0.1f;

const static int MARGIN = 10;

Camera::Camera(int WindowWidth, int WindowHeight)

{

m\_windowWidth = WindowWidth;

m\_windowHeight = WindowHeight;

m\_pos = Vector3f(0.0f, 0.0f, 0.0f);

m\_target = Vector3f(0.0f, 0.0f, 1.0f);

m\_target.Normalize();

m\_up = Vector3f(0.0f, 1.0f, 0.0f);

Init();

}

Camera::Camera(int WindowWidth, int WindowHeight, const Vector3f& Pos, const Vector3f& Target, const Vector3f& Up)

{

m\_windowWidth = WindowWidth;

m\_windowHeight = WindowHeight;

m\_pos = Pos;

m\_target = Target;

m\_target.Normalize();

m\_up = Up;

m\_up.Normalize();

Init();

}

void Camera::Init()

{

Vector3f HTarget(m\_target.x, 0.0, m\_target.z);

HTarget.Normalize();

if (HTarget.z >= 0.0f)

{

if (HTarget.x >= 0.0f)

{

m\_AngleH = 360.0f - ToDegree(asin(HTarget.z));

}

else

{

m\_AngleH = 180.0f + ToDegree(asin(HTarget.z));

}

}

else

{

if (HTarget.x >= 0.0f)

{

m\_AngleH = ToDegree(asin(-HTarget.z));

}

else

{

m\_AngleH = 90.0f + ToDegree(asin(-HTarget.z));

}

}

m\_AngleV = -ToDegree(asin(m\_target.y));

m\_mousePos.x = m\_windowWidth / 2;

m\_mousePos.y = m\_windowHeight / 2;

glutWarpPointer(m\_mousePos.x, m\_mousePos.y);

}

bool Camera::OnKeyboard(int Key)

{

bool Ret = false;

switch (Key) {

case GLUT\_KEY\_UP:

{

m\_pos += (m\_target \* STEP\_SCALE);

Ret = true;

}

break;

case GLUT\_KEY\_DOWN:

{

m\_pos -= (m\_target \* STEP\_SCALE);

Ret = true;

}

break;

case GLUT\_KEY\_LEFT:

{

Vector3f Left = m\_target.Cross(m\_up);

Left.Normalize();

Left \*= STEP\_SCALE;

m\_pos += Left;

Ret = true;

}

break;

case GLUT\_KEY\_RIGHT:

{

Vector3f Right = m\_up.Cross(m\_target);

Right.Normalize();

Right \*= STEP\_SCALE;

m\_pos += Right;

Ret = true;

}

break;

}

return Ret;

}

void Camera::OnMouse(int x, int y)

{

if (( x == m\_mousePos.x)&&(y == m\_mousePos.y)) return;

const int DeltaX = x - m\_mousePos.x;

const int DeltaY = y - m\_mousePos.y;

m\_AngleH += (float)DeltaX / 20.0f;

m\_AngleV += (float)DeltaY / 20.0f;

Update();

glutWarpPointer(m\_mousePos.x, m\_mousePos.y);

}

void Camera::OnRender()

{

bool ShouldUpdate = false;

if (ShouldUpdate) {

Update();

}

}

void Camera::Update()

{

const Vector3f Vaxis(0.0f, 1.0f, 0.0f);

// Rotate the view vector by the horizontal angle around the vertical axis

Vector3f View(1.0f, 0.0f, 0.0f);

View.Rotate(m\_AngleH, Vaxis);

View.Normalize();

// Rotate the view vector by the vertical angle around the horizontal axis

Vector3f Haxis = Vaxis.Cross(View);

Haxis.Normalize();

View.Rotate(m\_AngleV, Haxis);

m\_target = View;

m\_target.Normalize();

m\_up = m\_target.Cross(Haxis);

m\_up.Normalize();

}

**camera.h**

#ifndef CAMERA\_H

#define CAMERA\_H

#include "math\_3d.h"

class Camera

{

public:

Camera(int WindowWidth, int WindowHeight);

Camera(int WindowWidth, int WindowHeight, const Vector3f& Pos, const Vector3f& Target, const Vector3f& Up);

bool OnKeyboard(int Key);

void OnMouse(int x, int y);

void OnRender();

const Vector3f& GetPos() const

{

return m\_pos;

}

const Vector3f& GetTarget() const

{

return m\_target;

}

const Vector3f& GetUp() const

{

return m\_up;

}

private:

void Init();

void Update();

Vector3f m\_pos;

Vector3f m\_target;

Vector3f m\_up;

int m\_windowWidth;

int m\_windowHeight;

float m\_AngleH;

float m\_AngleV;

Vector2i m\_mousePos;

};

#endif /\* CAMERA\_H \*/

**glut\_backend.cpp**

#include <stdio.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "glut\_backend.h"

static ICallbacks\* s\_pCallbacks = NULL;

static void SpecialKeyboardCB(int Key, int x, int y){

s\_pCallbacks->SpecialKeyboardCB(Key, x, y);

}

static void KeyboardCB(unsigned char Key, int x, int y){

s\_pCallbacks->KeyboardCB(Key, x, y);

}

static void PassiveMouseCB(int x, int y){

s\_pCallbacks->PassiveMouseCB(x, y);

}

static void RenderSceneCB(){

s\_pCallbacks->RenderSceneCB();

}

static void IdleCB(){

s\_pCallbacks->IdleCB();

}

static void InitCallbacks(){

glutDisplayFunc(RenderSceneCB);

glutIdleFunc(IdleCB);

glutSpecialFunc(SpecialKeyboardCB);

glutPassiveMotionFunc(PassiveMouseCB);

glutKeyboardFunc(KeyboardCB);

}

void GLUTBackendInit(int argc, char\*\* argv){

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE|GLUT\_RGBA);

glutSetOption(GLUT\_ACTION\_ON\_WINDOW\_CLOSE, GLUT\_ACTION\_GLUTMAINLOOP\_RETURNS);

}

bool GLUTBackendCreateWindow(unsigned int Width, unsigned int Height, unsigned int bpp, bool isFullScreen, const char\* pTitle){

if (isFullScreen){

char ModeString[64] = {0};

snprintf(ModeString, sizeof(ModeString), "%dx%d@%d", Width, Height, bpp);

glutGameModeString(ModeString);

glutEnterGameMode();

}

else {

glutInitWindowSize(Width, Height);

glutCreateWindow(pTitle);

}

GLenum res = glewInit();

if (res != GLEW\_OK){

fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));

return false;

}

return true;

}

void GLUTBackendRun(ICallbacks\* pCallbacks){

if (!pCallbacks){

fprintf(stderr, "%s : callbacks not specified!\n", \_\_FUNCTION\_\_);

return;

}

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

glFrontFace(GL\_CW);

glCullFace(GL\_BACK);

glEnable(GL\_CULL\_FACE);

s\_pCallbacks = pCallbacks;

InitCallbacks();

glutMainLoop();

}

**glut\_backend.h**

#ifndef GLUT\_BACKEND\_H

#define GLUT\_BACKEND\_H

#include "callbacks.h"

void GLUTBackendInit(int argc, char\*\* argv);

bool GLUTBackendCreateWindow(unsigned int Width, unsigned int Height, unsigned int bpp, bool isFullScreen, const char\* pTitle);

void GLUTBackendRun(ICallbacks\* pCallbacks);

#endif /\* GLUT\_BACKEND\_H \*/

**lighting\_technique.cpp**

#include <limits.h>

#include <string.h>

#include "lighting\_technique.h"

#include "util.h"

static const char\* pVS = " \n\

#version 330 \n\

\n\

layout (location = 0) in vec3 Position; \n\

layout (location = 1) in vec2 TexCoord; \n\

layout (location = 2) in vec3 Normal; \n\

\n\

uniform mat4 gWVP; \n\

uniform mat4 gWorld; \n\

\n\

out vec2 TexCoord0; \n\

out vec3 Normal0; \n\

out vec3 WorldPos0; \n\

\n\

void main() \n\

{ \n\

gl\_Position = gWVP \* vec4(Position, 1.0); \n\

TexCoord0 = TexCoord; \n\

Normal0 = (gWorld \* vec4(Normal, 0.0)).xyz; \n\

WorldPos0 = (gWorld \* vec4(Position, 1.0)).xyz; \n\

}";

static const char\* pFS = " \n\

#version 330 \n\

\n\

const int MAX\_POINT\_LIGHTS = 2; \n\

const int MAX\_SPOT\_LIGHTS = 2; \n\

\n\

in vec2 TexCoord0; \n\

in vec3 Normal0; \n\

in vec3 WorldPos0; \n\

\n\

out vec4 FragColor; \n\

\n\

struct BaseLight \n\

{ \n\

vec3 Color; \n\

float AmbientIntensity; \n\

float DiffuseIntensity; \n\

}; \n\

\n\

struct DirectionalLight \n\

{ \n\

BaseLight Base; \n\

vec3 Direction; \n\

}; \n\

\n\

struct Attenuation \n\

{ \n\

float Constant; \n\

float Linear; \n\

float Exp; \n\

}; \n\

\n\

struct PointLight \n\

{ \n\

BaseLight Base; \n\

vec3 Position; \n\

Attenuation Atten; \n\

}; \n\

\n\

struct SpotLight \n\

{ \n\

PointLight Base; \n\

vec3 Direction; \n\

float Cutoff; \n\

}; \n\

\n\

uniform int gNumPointLights; \n\

uniform int gNumSpotLights; \n\

uniform DirectionalLight gDirectionalLight; \n\

uniform PointLight gPointLights[MAX\_POINT\_LIGHTS]; \n\

uniform SpotLight gSpotLights[MAX\_SPOT\_LIGHTS]; \n\

uniform sampler2D gSampler; \n\

uniform vec3 gEyeWorldPos; \n\

uniform float gMatSpecularIntensity; \n\

uniform float gSpecularPower; \n\

\n\

vec4 CalcLightInternal(BaseLight Light, vec3 LightDirection, vec3 Normal) \n\

{ \n\

vec4 AmbientColor = vec4(Light.Color, 1.0f) \* Light.AmbientIntensity; \n\

float DiffuseFactor = dot(Normal, -LightDirection); \n\

\n\

vec4 DiffuseColor = vec4(0, 0, 0, 0); \n\

vec4 SpecularColor = vec4(0, 0, 0, 0); \n\

\n\

if (DiffuseFactor > 0) { \n\

DiffuseColor = vec4(Light.Color, 1.0f) \* Light.DiffuseIntensity \* DiffuseFactor; \n\

\n\

vec3 VertexToEye = normalize(gEyeWorldPos - WorldPos0); \n\

vec3 LightReflect = normalize(reflect(LightDirection, Normal)); \n\

float SpecularFactor = dot(VertexToEye, LightReflect); \n\

SpecularFactor = pow(SpecularFactor, gSpecularPower); \n\

if (SpecularFactor > 0) { \n\

SpecularColor = vec4(Light.Color, 1.0f) \* \n\

gMatSpecularIntensity \* SpecularFactor; \n\

} \n\

} \n\

\n\

return (AmbientColor + DiffuseColor + SpecularColor); \n\

} \n\

\n\

vec4 CalcDirectionalLight(vec3 Normal) \n\

{ \n\

return CalcLightInternal(gDirectionalLight.Base, gDirectionalLight.Direction, Normal); \n\

} \n\

\n\

vec4 CalcPointLight( PointLight l, vec3 Normal) \n\

{ \n\

vec3 LightDirection = WorldPos0 - l.Position; \n\

float Distance = length(LightDirection); \n\

LightDirection = normalize(LightDirection); \n\

\n\

vec4 Color = CalcLightInternal(l.Base, LightDirection, Normal); \n\

float Attenuation = l.Atten.Constant + \n\

l.Atten.Linear \* Distance + \n\

l.Atten.Exp \* Distance \* Distance; \n\

\n\

return Color / Attenuation; \n\

} \n\

\n\

vec4 CalcSpotLight( SpotLight l, vec3 Normal) \n\

{ \n\

vec3 LightToPixel = normalize(WorldPos0 - l.Base.Position); \n\

float SpotFactor = dot(LightToPixel, l.Direction); \n\

\n\

if (SpotFactor > l.Cutoff) { \n\

vec4 Color = CalcPointLight(l.Base, Normal); \n\

return Color \* (1.0 - (1.0 - SpotFactor) \* 1.0/(1.0 - l.Cutoff)); \n\

} \n\

else { \n\

return vec4(0,0,0,0); \n\

} \n\

} \n\

\n\

void main() \n\

{ \n\

vec3 Normal = normalize(Normal0); \n\

vec4 TotalLight = CalcDirectionalLight(Normal); \n\

\n\

for (int i = 0 ; i < gNumPointLights ; i++) { \n\

TotalLight += CalcPointLight(gPointLights[i], Normal); \n\

} \n\

\n\

for (int i = 0 ; i < gNumSpotLights ; i++) { \n\

TotalLight += CalcSpotLight(gSpotLights[i], Normal); \n\

} \n\

\n\

FragColor = texture2D(gSampler, TexCoord0.xy) \* TotalLight; \n\

}";

LightingTechnique::LightingTechnique()

{

}

bool LightingTechnique::Init()

{

if (!Technique::Init()) {

return false;

}

if (!AddShader(GL\_VERTEX\_SHADER, pVS)) {

return false;

}

if (!AddShader(GL\_FRAGMENT\_SHADER, pFS)) {

return false;

}

if (!Finalize()) {

return false;

}

m\_WVPLocation = GetUniformLocation("gWVP");

m\_WorldMatrixLocation = GetUniformLocation("gWorld");

m\_samplerLocation = GetUniformLocation("gSampler");

m\_eyeWorldPosLocation = GetUniformLocation("gEyeWorldPos");

m\_dirLightLocation.Color = GetUniformLocation("gDirectionalLight.Base.Color");

m\_dirLightLocation.AmbientIntensity = GetUniformLocation("gDirectionalLight.Base.AmbientIntensity");

m\_dirLightLocation.Direction = GetUniformLocation("gDirectionalLight.Direction");

m\_dirLightLocation.DiffuseIntensity = GetUniformLocation("gDirectionalLight.Base.DiffuseIntensity");

m\_matSpecularIntensityLocation = GetUniformLocation("gMatSpecularIntensity");

m\_matSpecularPowerLocation = GetUniformLocation("gSpecularPower");

m\_numPointLightsLocation = GetUniformLocation("gNumPointLights");

m\_numSpotLightsLocation = GetUniformLocation("gNumSpotLights");

if (m\_dirLightLocation.AmbientIntensity == INVALID\_UNIFORM\_LOCATION ||

m\_WVPLocation == INVALID\_UNIFORM\_LOCATION ||

m\_WorldMatrixLocation == INVALID\_UNIFORM\_LOCATION ||

m\_samplerLocation == INVALID\_UNIFORM\_LOCATION ||

m\_eyeWorldPosLocation == INVALID\_UNIFORM\_LOCATION ||

m\_dirLightLocation.Color == INVALID\_UNIFORM\_LOCATION ||

m\_dirLightLocation.DiffuseIntensity == INVALID\_UNIFORM\_LOCATION ||

m\_dirLightLocation.Direction == INVALID\_UNIFORM\_LOCATION ||

m\_matSpecularIntensityLocation == INVALID\_UNIFORM\_LOCATION ||

m\_matSpecularPowerLocation == INVALID\_UNIFORM\_LOCATION ||

m\_numPointLightsLocation == INVALID\_UNIFORM\_LOCATION ||

m\_numSpotLightsLocation == INVALID\_UNIFORM\_LOCATION) {

return false;

}

for (unsigned int i = 0 ; i < ARRAY\_SIZE\_IN\_ELEMENTS(m\_pointLightsLocation) ; i++) {

char Name[128];

memset(Name, 0, sizeof(Name));

snprintf(Name, sizeof(Name), "gPointLights[%d].Base.Color", i);

m\_pointLightsLocation[i].Color = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gPointLights[%d].Base.AmbientIntensity", i);

m\_pointLightsLocation[i].AmbientIntensity = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gPointLights[%d].Position", i);

m\_pointLightsLocation[i].Position = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gPointLights[%d].Base.DiffuseIntensity", i);

m\_pointLightsLocation[i].DiffuseIntensity = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gPointLights[%d].Atten.Constant", i);

m\_pointLightsLocation[i].Atten.Constant = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gPointLights[%d].Atten.Linear", i);

m\_pointLightsLocation[i].Atten.Linear = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gPointLights[%d].Atten.Exp", i);

m\_pointLightsLocation[i].Atten.Exp = GetUniformLocation(Name);

if (m\_pointLightsLocation[i].Color == INVALID\_UNIFORM\_LOCATION ||

m\_pointLightsLocation[i].AmbientIntensity == INVALID\_UNIFORM\_LOCATION ||

m\_pointLightsLocation[i].Position == INVALID\_UNIFORM\_LOCATION ||

m\_pointLightsLocation[i].DiffuseIntensity == INVALID\_UNIFORM\_LOCATION ||

m\_pointLightsLocation[i].Atten.Constant == INVALID\_UNIFORM\_LOCATION ||

m\_pointLightsLocation[i].Atten.Linear == INVALID\_UNIFORM\_LOCATION ||

m\_pointLightsLocation[i].Atten.Exp == INVALID\_UNIFORM\_LOCATION) {

return false;

}

}

for (unsigned int i = 0 ; i < ARRAY\_SIZE\_IN\_ELEMENTS(m\_spotLightsLocation) ; i++) {

char Name[128];

memset(Name, 0, sizeof(Name));

snprintf(Name, sizeof(Name), "gSpotLights[%d].Base.Base.Color", i);

m\_spotLightsLocation[i].Color = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gSpotLights[%d].Base.Base.AmbientIntensity", i);

m\_spotLightsLocation[i].AmbientIntensity = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gSpotLights[%d].Base.Position", i);

m\_spotLightsLocation[i].Position = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gSpotLights[%d].Direction", i);

m\_spotLightsLocation[i].Direction = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gSpotLights[%d].Cutoff", i);

m\_spotLightsLocation[i].Cutoff = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gSpotLights[%d].Base.Base.DiffuseIntensity", i);

m\_spotLightsLocation[i].DiffuseIntensity = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gSpotLights[%d].Base.Atten.Constant", i);

m\_spotLightsLocation[i].Atten.Constant = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gSpotLights[%d].Base.Atten.Linear", i);

m\_spotLightsLocation[i].Atten.Linear = GetUniformLocation(Name);

snprintf(Name, sizeof(Name), "gSpotLights[%d].Base.Atten.Exp", i);

m\_spotLightsLocation[i].Atten.Exp = GetUniformLocation(Name);

if (m\_spotLightsLocation[i].Color == INVALID\_UNIFORM\_LOCATION ||

m\_spotLightsLocation[i].AmbientIntensity == INVALID\_UNIFORM\_LOCATION ||

m\_spotLightsLocation[i].Position == INVALID\_UNIFORM\_LOCATION ||

m\_spotLightsLocation[i].Direction == INVALID\_UNIFORM\_LOCATION ||

m\_spotLightsLocation[i].Cutoff == INVALID\_UNIFORM\_LOCATION ||

m\_spotLightsLocation[i].DiffuseIntensity == INVALID\_UNIFORM\_LOCATION ||

m\_spotLightsLocation[i].Atten.Constant == INVALID\_UNIFORM\_LOCATION ||

m\_spotLightsLocation[i].Atten.Linear == INVALID\_UNIFORM\_LOCATION ||

m\_spotLightsLocation[i].Atten.Exp == INVALID\_UNIFORM\_LOCATION) {

return false;

}

}

return true;

}

void LightingTechnique::SetWVP(const Matrix4f& WVP)

{

glUniformMatrix4fv(m\_WVPLocation, 1, GL\_TRUE, (const GLfloat\*)WVP.m);

}

void LightingTechnique::SetWorldMatrix(const Matrix4f& WorldInverse)

{

glUniformMatrix4fv(m\_WorldMatrixLocation, 1, GL\_TRUE, (const GLfloat\*)WorldInverse.m);

}

void LightingTechnique::SetTextureUnit(unsigned int TextureUnit)

{

glUniform1i(m\_samplerLocation, TextureUnit);

}

void LightingTechnique::SetDirectionalLight(const DirectionalLight& Light)

{

glUniform3f(m\_dirLightLocation.Color, Light.Color.x, Light.Color.y, Light.Color.z);

glUniform1f(m\_dirLightLocation.AmbientIntensity, Light.AmbientIntensity);

Vector3f Direction = Light.Direction;

Direction.Normalize();

glUniform3f(m\_dirLightLocation.Direction, Direction.x, Direction.y, Direction.z);

glUniform1f(m\_dirLightLocation.DiffuseIntensity, Light.DiffuseIntensity);

}

void LightingTechnique::SetEyeWorldPos(const Vector3f& EyeWorldPos)

{

glUniform3f(m\_eyeWorldPosLocation, EyeWorldPos.x, EyeWorldPos.y, EyeWorldPos.z);

}

void LightingTechnique::SetMatSpecularIntensity(float Intensity)

{

glUniform1f(m\_matSpecularIntensityLocation, Intensity);

}

void LightingTechnique::SetMatSpecularPower(float Power)

{

glUniform1f(m\_matSpecularPowerLocation, Power);

}

void LightingTechnique::SetPointLights(unsigned int NumLights, const PointLight\* pLights)

{

glUniform1i(m\_numPointLightsLocation, NumLights);

for (unsigned int i = 0 ; i < NumLights ; i++) {

glUniform3f(m\_pointLightsLocation[i].Color, pLights[i].Color.x, pLights[i].Color.y, pLights[i].Color.z);

glUniform1f(m\_pointLightsLocation[i].AmbientIntensity, pLights[i].AmbientIntensity);

glUniform1f(m\_pointLightsLocation[i].DiffuseIntensity, pLights[i].DiffuseIntensity);

glUniform3f(m\_pointLightsLocation[i].Position, pLights[i].Position.x, pLights[i].Position.y, pLights[i].Position.z);

glUniform1f(m\_pointLightsLocation[i].Atten.Constant, pLights[i].Attenuation.Constant);

glUniform1f(m\_pointLightsLocation[i].Atten.Linear, pLights[i].Attenuation.Linear);

glUniform1f(m\_pointLightsLocation[i].Atten.Exp, pLights[i].Attenuation.Exp);

}

}

void LightingTechnique::SetSpotLights(unsigned int NumLights, const SpotLight\* pLights)

{

glUniform1i(m\_numSpotLightsLocation, NumLights);

for (unsigned int i = 0 ; i < NumLights ; i++) {

glUniform3f(m\_spotLightsLocation[i].Color, pLights[i].Color.x, pLights[i].Color.y, pLights[i].Color.z);

glUniform1f(m\_spotLightsLocation[i].AmbientIntensity, pLights[i].AmbientIntensity);

glUniform1f(m\_spotLightsLocation[i].DiffuseIntensity, pLights[i].DiffuseIntensity);

glUniform3f(m\_spotLightsLocation[i].Position, pLights[i].Position.x, pLights[i].Position.y, pLights[i].Position.z);

Vector3f Direction = pLights[i].Direction;

Direction.Normalize();

glUniform3f(m\_spotLightsLocation[i].Direction, Direction.x, Direction.y, Direction.z);

glUniform1f(m\_spotLightsLocation[i].Cutoff, cosf(ToRadian(pLights[i].Cutoff)));

glUniform1f(m\_spotLightsLocation[i].Atten.Constant, pLights[i].Attenuation.Constant);

glUniform1f(m\_spotLightsLocation[i].Atten.Linear, pLights[i].Attenuation.Linear);

glUniform1f(m\_spotLightsLocation[i].Atten.Exp, pLights[i].Attenuation.Exp);

}

}

**lighting\_technique.h**

#ifndef LIGHTING\_TECHNIQUE\_H

#define LIGHTING\_TECHNIQUE\_H

#include "technique.h"

#include "math\_3d.h"

struct BaseLight

{

Vector3f Color;

float AmbientIntensity;

float DiffuseIntensity;

BaseLight()

{

Color = Vector3f(0.0f, 0.0f, 0.0f);

AmbientIntensity = 0.0f;

DiffuseIntensity = 0.0f;

}

};

struct DirectionalLight : public BaseLight

{

Vector3f Direction;

DirectionalLight()

{

Direction = Vector3f(0.0f, 0.0f, 0.0f);

}

};

struct PointLight : public BaseLight

{

Vector3f Position;

struct

{

float Constant;

float Linear;

float Exp;

} Attenuation;

PointLight()

{

Position = Vector3f(0.0f, 0.0f, 0.0f);

Attenuation.Constant = 1.0f;

Attenuation.Linear = 0.0f;

Attenuation.Exp = 0.0f;

}

};

struct SpotLight : public PointLight

{

Vector3f Direction;

float Cutoff;

SpotLight()

{

Direction = Vector3f(0.0f, 0.0f, 0.0f);

Cutoff = 0.0f;

}

};

class LightingTechnique : public Technique {

public:

static const unsigned int MAX\_POINT\_LIGHTS = 2;

static const unsigned int MAX\_SPOT\_LIGHTS = 2;

LightingTechnique();

virtual bool Init();

void SetWVP(const Matrix4f& WVP);

void SetWorldMatrix(const Matrix4f& WVP);

void SetTextureUnit(unsigned int TextureUnit);

void SetDirectionalLight(const DirectionalLight& Light);

void SetPointLights(unsigned int NumLights, const PointLight\* pLights);

void SetSpotLights(unsigned int NumLights, const SpotLight\* pLights);

void SetEyeWorldPos(const Vector3f& EyeWorldPos);

void SetMatSpecularIntensity(float Intensity);

void SetMatSpecularPower(float Power);

private:

GLuint m\_WVPLocation;

GLuint m\_WorldMatrixLocation;

GLuint m\_samplerLocation;

GLuint m\_eyeWorldPosLocation;

GLuint m\_matSpecularIntensityLocation;

GLuint m\_matSpecularPowerLocation;

GLuint m\_numPointLightsLocation;

GLuint m\_numSpotLightsLocation;

struct {

GLuint Color;

GLuint AmbientIntensity;

GLuint DiffuseIntensity;

GLuint Direction;

} m\_dirLightLocation;

struct {

GLuint Color;

GLuint AmbientIntensity;

GLuint DiffuseIntensity;

GLuint Position;

struct {

GLuint Constant;

GLuint Linear;

GLuint Exp;

} Atten;

} m\_pointLightsLocation[MAX\_POINT\_LIGHTS];

struct {

GLuint Color;

GLuint AmbientIntensity;

GLuint DiffuseIntensity;

GLuint Position;

GLuint Direction;

GLuint Cutoff;

struct {

GLuint Constant;

GLuint Linear;

GLuint Exp;

} Atten;

} m\_spotLightsLocation[MAX\_SPOT\_LIGHTS];

};

#endif /\* LIGHTING\_TECHNIQUE\_H \*/

#ifndef LIGHTING\_TECHNIQUE\_H

#define LIGHTING\_TECHNIQUE\_H

#include "technique.h"

#include "math\_3d.h"

struct BaseLight

{

Vector3f Color;

float AmbientIntensity;

float DiffuseIntensity;

BaseLight()

{

Color = Vector3f(0.0f, 0.0f, 0.0f);

AmbientIntensity = 0.0f;

DiffuseIntensity = 0.0f;

}

};

struct DirectionalLight : public BaseLight

{

Vector3f Direction;

DirectionalLight()

{

Direction = Vector3f(0.0f, 0.0f, 0.0f);

}

};

struct PointLight : public BaseLight

{

Vector3f Position;

struct

{

float Constant;

float Linear;

float Exp;

} Attenuation;

PointLight()

{

Position = Vector3f(0.0f, 0.0f, 0.0f);

Attenuation.Constant = 1.0f;

Attenuation.Linear = 0.0f;

Attenuation.Exp = 0.0f;

}

};

struct SpotLight : public PointLight

{

Vector3f Direction;

float Cutoff;

SpotLight()

{

Direction = Vector3f(0.0f, 0.0f, 0.0f);

Cutoff = 0.0f;

}

};

class LightingTechnique : public Technique {

public:

static const unsigned int MAX\_POINT\_LIGHTS = 2;

static const unsigned int MAX\_SPOT\_LIGHTS = 2;

LightingTechnique();

virtual bool Init();

void SetWVP(const Matrix4f& WVP);

void SetWorldMatrix(const Matrix4f& WVP);

void SetTextureUnit(unsigned int TextureUnit);

void SetDirectionalLight(const DirectionalLight& Light);

void SetPointLights(unsigned int NumLights, const PointLight\* pLights);

void SetSpotLights(unsigned int NumLights, const SpotLight\* pLights);

void SetEyeWorldPos(const Vector3f& EyeWorldPos);

void SetMatSpecularIntensity(float Intensity);

void SetMatSpecularPower(float Power);

private:

GLuint m\_WVPLocation;

GLuint m\_WorldMatrixLocation;

GLuint m\_samplerLocation;

GLuint m\_eyeWorldPosLocation;

GLuint m\_matSpecularIntensityLocation;

GLuint m\_matSpecularPowerLocation;

GLuint m\_numPointLightsLocation;

GLuint m\_numSpotLightsLocation;

struct {

GLuint Color;

GLuint AmbientIntensity;

GLuint DiffuseIntensity;

GLuint Direction;

} m\_dirLightLocation;

struct {

GLuint Color;

GLuint AmbientIntensity;

GLuint DiffuseIntensity;

GLuint Position;

struct {

GLuint Constant;

GLuint Linear;

GLuint Exp;

} Atten;

} m\_pointLightsLocation[MAX\_POINT\_LIGHTS];

struct {

GLuint Color;

GLuint AmbientIntensity;

GLuint DiffuseIntensity;

GLuint Position;

GLuint Direction;

GLuint Cutoff;

struct {

GLuint Constant;

GLuint Linear;

GLuint Exp;

} Atten;

} m\_spotLightsLocation[MAX\_SPOT\_LIGHTS];

};

#endif /\* LIGHTING\_TECHNIQUE\_H \*/

**main.cpp**

#include <math.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "pipeline.h"

#include "camera.h"

#include "texture.h"

#include "lighting\_technique.h"

#include "glut\_backend.h"

#include "util.h"

#define WINDOW\_WIDTH 1280

#define WINDOW\_HEIGHT 1024

struct Vertex

{

Vector3f m\_pos;

Vector2f m\_tex;

Vector3f m\_normal;

Vertex() {}

Vertex(Vector3f pos, Vector2f tex)

{

m\_pos = pos;

m\_tex = tex;

m\_normal = Vector3f(0.0f, 0.0f, 0.0f);

}

};

class Main : public ICallbacks

{

public:

Main()

{

m\_pGameCamera = NULL;

m\_pTexture = NULL;

m\_pEffect = NULL;

m\_scale = 0.0f;

m\_directionalLight.Color = Vector3f(1.0f, 1.0f, 1.0f);

m\_directionalLight.AmbientIntensity = 0.0f;

m\_directionalLight.DiffuseIntensity = 0.0f;

m\_directionalLight.Direction = Vector3f(1.0f, 0.0f, 0.0f);

}

~Main()

{

delete m\_pEffect;

delete m\_pGameCamera;

delete m\_pTexture;

}

bool Init()

{

Vector3f Pos(-10.0f, 0.0f, -10.0f);

Vector3f Target(1.0f, 0.0f, 1.0f);

Vector3f Up(0.0, 1.0f, 0.0f);

m\_pGameCamera = new Camera(WINDOW\_WIDTH, WINDOW\_HEIGHT, Pos, Target, Up);

unsigned int Indices[] = { 0, 2, 1,

0, 3, 2};

CreateIndexBuffer(Indices, sizeof(Indices));

CreateVertexBuffer(Indices, ARRAY\_SIZE\_IN\_ELEMENTS(Indices));

m\_pEffect = new LightingTechnique();

if (!m\_pEffect->Init())

{

printf("Error initializing the lighting technique\n");

return false;

}

m\_pEffect->Enable();

m\_pEffect->SetTextureUnit(0);

m\_pTexture = new Texture(GL\_TEXTURE\_2D, "../Content/test.png");

if (!m\_pTexture->Load()) {

return false;

}

return true;

}

void Run()

{

GLUTBackendRun(this);

}

virtual void RenderSceneCB()

{

m\_pGameCamera->OnRender();

glClear(GL\_COLOR\_BUFFER\_BIT);

m\_scale += 0.001f;

SpotLight sl[2];

sl[0].DiffuseIntensity = 15.0f;

sl[0].Color = Vector3f(0.5f, 1.0f, 0.0f);

sl[0].Position = Vector3f(-0.0f, -1.9f, -0.0f);

sl[0].Direction = Vector3f(sinf(m\_scale), 0.0f, cosf(m\_scale));

sl[0].Attenuation.Linear = 0.1f;

sl[0].Cutoff = 20.0f;

sl[1].DiffuseIntensity = 10.0f;

sl[1].Color = Vector3f(0.0f, 1.0f, 1.0f);

sl[1].Position = m\_pGameCamera->GetPos();

sl[1].Direction = m\_pGameCamera->GetTarget();

sl[1].Attenuation.Linear = 0.5f;

sl[1].Cutoff = 10.0f;

m\_pEffect->SetSpotLights(2, sl);

Pipeline p;

p.Rotate(0.0f, 0.0f, 0.0f);

p.WorldPos(0.0f, 0.0f, 1.0f);

p.SetCamera(m\_pGameCamera->GetPos(), m\_pGameCamera->GetTarget(), m\_pGameCamera->GetUp());

p.SetPerspectiveProj(60.0f, WINDOW\_WIDTH, WINDOW\_HEIGHT, 0.1f, 100.0f);

m\_pEffect->SetWVP(p.GetWVPTrans());

const Matrix4f& WorldTransformation = p.GetWorldTrans();

m\_pEffect->SetWorldMatrix(WorldTransformation);

m\_pEffect->SetDirectionalLight(m\_directionalLight);

m\_pEffect->SetEyeWorldPos(m\_pGameCamera->GetPos());

m\_pEffect->SetMatSpecularIntensity(1.0f);

m\_pEffect->SetMatSpecularPower(32);

glEnableVertexAttribArray(0);

glEnableVertexAttribArray(1);

glEnableVertexAttribArray(2);

glBindBuffer(GL\_ARRAY\_BUFFER, m\_VBO);

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

glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (const GLvoid\*)12);

glVertexAttribPointer(2, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (const GLvoid\*)20);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, m\_IBO);

m\_pTexture->Bind(GL\_TEXTURE0);

glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, 0);

glDisableVertexAttribArray(0);

glDisableVertexAttribArray(1);

glDisableVertexAttribArray(2);

glutSwapBuffers();

}

virtual void IdleCB()

{

RenderSceneCB();

}

virtual void SpecialKeyboardCB(int Key, int x, int y)

{

m\_pGameCamera->OnKeyboard(Key);

}

virtual void KeyboardCB(unsigned char Key, int x, int y)

{

switch (Key) {

case 'q':

glutLeaveMainLoop();

break;

case 'a':

m\_directionalLight.AmbientIntensity += 0.05f;

break;

case 's':

m\_directionalLight.AmbientIntensity -= 0.05f;

break;

case 'z':

m\_directionalLight.DiffuseIntensity += 0.05f;

break;

case 'x':

m\_directionalLight.DiffuseIntensity -= 0.05f;

break;

}

}

virtual void PassiveMouseCB(int x, int y)

{

m\_pGameCamera->OnMouse(x, y);

}

private:

void CalcNormals(const unsigned int\* pIndices, unsigned int IndexCount,

Vertex\* pVertices, unsigned int VertexCount){

for (unsigned int i = 0; i < IndexCount; i +=3 ){

unsigned int Index0 = pIndices[i];

unsigned int Index1 = pIndices[i + 1];

unsigned int Index2 = pIndices[i + 2];

Vector3f v1 = pVertices[Index1].m\_pos - pVertices[Index0].m\_pos;

Vector3f v2 = pVertices[Index2].m\_pos - pVertices[Index0].m\_pos;

Vector3f Normal = v1.Cross(v2);

Normal.Normalize();

pVertices[Index0].m\_normal += Normal;

pVertices[Index1].m\_normal += Normal;

pVertices[Index2].m\_normal += Normal;

}

for (unsigned int i = 0; i < VertexCount; i++){

pVertices[i].m\_normal.Normalize();

}

}

void CreateVertexBuffer(const unsigned int\* pIndices, unsigned int IndexCount)

{

Vertex Vertices[4] = { Vertex(Vector3f(-10.0f, -2.0f, -10.0f), Vector2f(0.0f, 0.0f)),

Vertex(Vector3f(10.0f, -2.0f, -10.0f), Vector2f(1.0f, 0.0f)),

Vertex(Vector3f(10.0f, -2.0f, 10.0f), Vector2f(1.0f, 1.0f)),

Vertex(Vector3f(-10.0f, -2.0f, 10.0f), Vector2f(0.0f, 1.0f))};

unsigned int VertexCount = ARRAY\_SIZE\_IN\_ELEMENTS(Vertices);

CalcNormals(pIndices, IndexCount, Vertices, VertexCount);

glGenBuffers(1, &m\_VBO);

glBindBuffer(GL\_ARRAY\_BUFFER, m\_VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(Vertices), Vertices, GL\_STATIC\_DRAW);

}

void CreateIndexBuffer(const unsigned int\* pIndices, unsigned int SizeInBytes)

{

glGenBuffers(1, &m\_IBO);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, m\_IBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, SizeInBytes, pIndices, GL\_STATIC\_DRAW);

}

GLuint m\_VBO;

GLuint m\_IBO;

LightingTechnique\* m\_pEffect;

Texture\* m\_pTexture;

Camera\* m\_pGameCamera;

float m\_scale;

DirectionalLight m\_directionalLight;

};

int main(int argc, char\*\* argv) {

Magick::InitializeMagick(\*argv);

GLUTBackendInit(argc, argv);

if (!GLUTBackendCreateWindow(WINDOW\_WIDTH, WINDOW\_HEIGHT, 32, false, "OpenGL tutors")) {

return 1;

}

Main\* pApp = new Main();

if (!pApp->Init()) {

return 1;

}

pApp->Run();

delete pApp;

return 0;

}

**math\_3d.h**

#ifndef MATH\_3D\_H

#define MATH\_3D\_H

#include <stdio.h>

#include <math.h>

#include <corecrt\_math\_defines.h>

#define ToRadian(x) ((x) \* M\_PI / 180.0f)

#define ToDegree(x) ((x) \* 180.0f / M\_PI)

struct Vector2i

{

int x;

int y;

};

struct Vector2f

{

float x;

float y;

Vector2f()

{

}

Vector2f(float \_x, float \_y)

{

x = \_x;

y = \_y;

}

};

struct Vector3f

{

float x;

float y;

float z;

Vector3f()

{

}

Vector3f(float \_x, float \_y, float \_z)

{

x = \_x;

y = \_y;

z = \_z;

}

Vector3f& operator+=(const Vector3f& r)

{

x += r.x;

y += r.y;

z += r.z;

return \*this;

}

Vector3f& operator-=(const Vector3f& r)

{

x -= r.x;

y -= r.y;

z -= r.z;

return \*this;

}

Vector3f& operator\*=(float f)

{

x \*= f;

y \*= f;

z \*= f;

return \*this;

}

Vector3f Cross(const Vector3f& v) const;

Vector3f& Normalize();

void Rotate(float Angle, const Vector3f& Axis);

void Print() const

{

printf("(%.02f, %.02f, %.02f", x, y, z);

}

};

inline Vector3f operator+(const Vector3f& l, const Vector3f& r)

{

Vector3f Ret(l.x + r.x,

l.y + r.y,

l.z + r.z);

return Ret;

}

inline Vector3f operator-(const Vector3f& l, const Vector3f& r)

{

Vector3f Ret(l.x - r.x,

l.y - r.y,

l.z - r.z);

return Ret;

}

inline Vector3f operator\*(const Vector3f& l, float f)

{

Vector3f Ret(l.x \* f,

l.y \* f,

l.z \* f);

return Ret;

}

class Matrix4f

{

public:

float m[4][4];

Matrix4f()

{

}

inline void InitIdentity()

{

m[0][0] = 1.0f; m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = 0.0f;

m[1][0] = 0.0f; m[1][1] = 1.0f; m[1][2] = 0.0f; m[1][3] = 0.0f;

m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = 1.0f; m[2][3] = 0.0f;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;

}

inline Matrix4f operator\*(const Matrix4f& Right) const

{

Matrix4f Ret;

for (unsigned int i = 0 ; i < 4 ; i++) {

for (unsigned int j = 0 ; j < 4 ; j++) {

Ret.m[i][j] = m[i][0] \* Right.m[0][j] +

m[i][1] \* Right.m[1][j] +

m[i][2] \* Right.m[2][j] +

m[i][3] \* Right.m[3][j];

}

}

return Ret;

}

void InitScaleTransform(float ScaleX, float ScaleY, float ScaleZ);

void InitRotateTransform(float RotateX, float RotateY, float RotateZ);

void InitTranslationTransform(float x, float y, float z);

void InitCameraTransform(const Vector3f& Target, const Vector3f& Up);

void InitPersProjTransform(float FOV, float Width, float Height, float zNear, float zFar);

};

struct Quaternion

{

float x, y, z, w;

Quaternion(float \_x, float \_y, float \_z, float \_w);

void Normalize();

Quaternion Conjugate();

};

Quaternion operator\*(const Quaternion& l, const Quaternion& r);

Quaternion operator\*(const Quaternion& q, const Vector3f& v);

#endif /\* MATH\_3D\_H \*/

**math\_3d.cpp**

#include "math\_3d.h"

Vector3f Vector3f::Cross(const Vector3f& v) const

{

const float \_x = y \* v.z - z \* v.y;

const float \_y = z \* v.x - x \* v.z;

const float \_z = x \* v.y - y \* v.x;

return Vector3f(\_x, \_y, \_z);

}

Vector3f& Vector3f::Normalize()

{

const float Length = sqrtf(x \* x + y \* y + z \* z);

x /= Length;

y /= Length;

z /= Length;

return \*this;

}

void Vector3f::Rotate(float Angle, const Vector3f& Axe)

{

const float SinHalfAngle = sinf(ToRadian(Angle/2));

const float CosHalfAngle = cosf(ToRadian(Angle/2));

const float Rx = Axe.x \* SinHalfAngle;

const float Ry = Axe.y \* SinHalfAngle;

const float Rz = Axe.z \* SinHalfAngle;

const float Rw = CosHalfAngle;

Quaternion RotationQ(Rx, Ry, Rz, Rw);

Quaternion ConjugateQ = RotationQ.Conjugate();

// ConjugateQ.Normalize();

Quaternion W = RotationQ \* (\*this) \* ConjugateQ;

x = W.x;

y = W.y;

z = W.z;

}

void Matrix4f::InitScaleTransform(float ScaleX, float ScaleY, float ScaleZ)

{

m[0][0] = ScaleX; m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = 0.0f;

m[1][0] = 0.0f; m[1][1] = ScaleY; m[1][2] = 0.0f; m[1][3] = 0.0f;

m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = ScaleZ; m[2][3] = 0.0f;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;

}

void Matrix4f::InitRotateTransform(float RotateX, float RotateY, float RotateZ)

{

Matrix4f rx, ry, rz;

const float x = ToRadian(RotateX);

const float y = ToRadian(RotateY);

const float z = ToRadian(RotateZ);

rx.m[0][0] = 1.0f; rx.m[0][1] = 0.0f ; rx.m[0][2] = 0.0f ; rx.m[0][3] = 0.0f;

rx.m[1][0] = 0.0f; rx.m[1][1] = cosf(x); rx.m[1][2] = -sinf(x); rx.m[1][3] = 0.0f;

rx.m[2][0] = 0.0f; rx.m[2][1] = sinf(x); rx.m[2][2] = cosf(x) ; rx.m[2][3] = 0.0f;

rx.m[3][0] = 0.0f; rx.m[3][1] = 0.0f ; rx.m[3][2] = 0.0f ; rx.m[3][3] = 1.0f;

ry.m[0][0] = cosf(y); ry.m[0][1] = 0.0f; ry.m[0][2] = -sinf(y); ry.m[0][3] = 0.0f;

ry.m[1][0] = 0.0f ; ry.m[1][1] = 1.0f; ry.m[1][2] = 0.0f ; ry.m[1][3] = 0.0f;

ry.m[2][0] = sinf(y); ry.m[2][1] = 0.0f; ry.m[2][2] = cosf(y) ; ry.m[2][3] = 0.0f;

ry.m[3][0] = 0.0f ; ry.m[3][1] = 0.0f; ry.m[3][2] = 0.0f ; ry.m[3][3] = 1.0f;

rz.m[0][0] = cosf(z); rz.m[0][1] = -sinf(z); rz.m[0][2] = 0.0f; rz.m[0][3] = 0.0f;

rz.m[1][0] = sinf(z); rz.m[1][1] = cosf(z) ; rz.m[1][2] = 0.0f; rz.m[1][3] = 0.0f;

rz.m[2][0] = 0.0f ; rz.m[2][1] = 0.0f ; rz.m[2][2] = 1.0f; rz.m[2][3] = 0.0f;

rz.m[3][0] = 0.0f ; rz.m[3][1] = 0.0f ; rz.m[3][2] = 0.0f; rz.m[3][3] = 1.0f;

\*this = rz \* ry \* rx;

}

void Matrix4f::InitTranslationTransform(float x, float y, float z)

{

m[0][0] = 1.0f; m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = x;

m[1][0] = 0.0f; m[1][1] = 1.0f; m[1][2] = 0.0f; m[1][3] = y;

m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = 1.0f; m[2][3] = z;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;

}

void Matrix4f::InitCameraTransform(const Vector3f& Target, const Vector3f& Up)

{

Vector3f N = Target;

N.Normalize();

Vector3f U = Up;

U.Normalize();

U = U.Cross(N);

Vector3f V = N.Cross(U);

m[0][0] = U.x; m[0][1] = U.y; m[0][2] = U.z; m[0][3] = 0.0f;

m[1][0] = V.x; m[1][1] = V.y; m[1][2] = V.z; m[1][3] = 0.0f;

m[2][0] = N.x; m[2][1] = N.y; m[2][2] = N.z; m[2][3] = 0.0f;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;

}

void Matrix4f::InitPersProjTransform(float FOV, float Width, float Height, float zNear, float zFar)

{

const float ar = Width / Height;

const float zRange = zNear - zFar;

const float tanHalfFOV = tanf(ToRadian(FOV / 2.0f));

m[0][0] = 1.0f/(tanHalfFOV \* ar); m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = 0.0;

m[1][0] = 0.0f; m[1][1] = 1.0f/tanHalfFOV; m[1][2] = 0.0f; m[1][3] = 0.0;

m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = (-zNear -zFar)/zRange ; m[2][3] = 2.0f \* zFar\*zNear/zRange;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 1.0f; m[3][3] = 0.0;

}

Quaternion::Quaternion(float \_x, float \_y, float \_z, float \_w)

{

x = \_x;

y = \_y;

z = \_z;

w = \_w;

}

void Quaternion::Normalize()

{

float Length = sqrtf(x \* x + y \* y + z \* z + w \* w);

x /= Length;

y /= Length;

z /= Length;

w /= Length;

}

Quaternion Quaternion::Conjugate()

{

Quaternion ret(-x, -y, -z, w);

return ret;

}

Quaternion operator\*(const Quaternion& l, const Quaternion& r)

{

const float w = (l.w \* r.w) - (l.x \* r.x) - (l.y \* r.y) - (l.z \* r.z);

const float x = (l.x \* r.w) + (l.w \* r.x) + (l.y \* r.z) - (l.z \* r.y);

const float y = (l.y \* r.w) + (l.w \* r.y) + (l.z \* r.x) - (l.x \* r.z);

const float z = (l.z \* r.w) + (l.w \* r.z) + (l.x \* r.y) - (l.y \* r.x);

Quaternion ret(x, y, z, w);

return ret;

}

Quaternion operator\*(const Quaternion& q, const Vector3f& v)

{

const float w = - (q.x \* v.x) - (q.y \* v.y) - (q.z \* v.z);

const float x = (q.w \* v.x) + (q.y \* v.z) - (q.z \* v.y);

const float y = (q.w \* v.y) + (q.z \* v.x) - (q.x \* v.z);

const float z = (q.w \* v.z) + (q.x \* v.y) - (q.y \* v.x);

Quaternion ret(x, y, z, w);

return ret;

}

**pipeline.cpp**

#include "pipeline.h"

const Matrix4f& Pipeline::GetWorldTrans()

{

Matrix4f ScaleTrans, RotateTrans, TranslationTrans;

ScaleTrans.InitScaleTransform(m\_scale.x, m\_scale.y, m\_scale.z);

RotateTrans.InitRotateTransform(m\_rotateInfo.x, m\_rotateInfo.y, m\_rotateInfo.z);

TranslationTrans.InitTranslationTransform(m\_worldPos.x, m\_worldPos.y, m\_worldPos.z);

m\_WorldTransformation = TranslationTrans \* RotateTrans \* ScaleTrans;

return m\_WorldTransformation;

}

const Matrix4f& Pipeline::GetWVPTrans()

{

GetWorldTrans();

Matrix4f CameraTranslationTrans, CameraRotateTrans, PersProjTrans;

CameraTranslationTrans.InitTranslationTransform(-m\_camera.Pos.x, -m\_camera.Pos.y, -m\_camera.Pos.z);

CameraRotateTrans.InitCameraTransform(m\_camera.Target, m\_camera.Up);

PersProjTrans.InitPersProjTransform(m\_persProj.FOV, m\_persProj.Width, m\_persProj.Height, m\_persProj.zNear, m\_persProj.zFar);

m\_WVPtransformation = PersProjTrans \* CameraRotateTrans \* CameraTranslationTrans \* m\_WorldTransformation;

return m\_WVPtransformation;

}

**pipeline.h**

#ifndef PIPELINE\_H

#define PIPELINE\_H

#include "math\_3d.h"

class Pipeline

{

public:

Pipeline()

{

m\_scale = Vector3f(1.0f, 1.0f, 1.0f);

m\_worldPos = Vector3f(0.0f, 0.0f, 0.0f);

m\_rotateInfo = Vector3f(0.0f, 0.0f, 0.0f);

}

void Scale(float ScaleX, float ScaleY, float ScaleZ)

{

m\_scale.x = ScaleX;

m\_scale.y = ScaleY;

m\_scale.z = ScaleZ;

}

void WorldPos(float x, float y, float z)

{

m\_worldPos.x = x;

m\_worldPos.y = y;

m\_worldPos.z = z;

}

void Rotate(float RotateX, float RotateY, float RotateZ)

{

m\_rotateInfo.x = RotateX;

m\_rotateInfo.y = RotateY;

m\_rotateInfo.z = RotateZ;

}

void SetPerspectiveProj(float FOV, float Width, float Height, float zNear, float zFar)

{

m\_persProj.FOV = FOV;

m\_persProj.Width = Width;

m\_persProj.Height = Height;

m\_persProj.zNear = zNear;

m\_persProj.zFar = zFar;

}

void SetCamera(const Vector3f& Pos, const Vector3f& Target, const Vector3f& Up)

{

m\_camera.Pos = Pos;

m\_camera.Target = Target;

m\_camera.Up = Up;

}

const Matrix4f& GetWVPTrans();

const Matrix4f& GetWorldTrans();

private:

Vector3f m\_scale;

Vector3f m\_worldPos;

Vector3f m\_rotateInfo;

struct {

float FOV;

float Width;

float Height;

float zNear;

float zFar;

} m\_persProj;

struct {

Vector3f Pos;

Vector3f Target;

Vector3f Up;

} m\_camera;

Matrix4f m\_WVPtransformation;

Matrix4f m\_WorldTransformation;

};

#endif /\* PIPELINE\_H \*/

**technique.h**

#ifndef TEXHNIQUE\_H

#define TEXHNIQUE\_H

#include <GL/glew.h>

#include <list>

#define INVALID\_UNIFORM\_LOCATION 0xFFFFFFFF

class Technique

{

public:

Technique();

~Technique();

virtual bool Init();

void Enable();

protected:

bool AddShader(GLenum ShaderType, const char\* pShaderText);

bool Finalize();

GLint GetUniformLocation(const char\* pUniformName);

private:

GLuint m\_shaderProg;

typedef std::list<GLuint> ShaderObjList;

ShaderObjList m\_shaderObjList;

};

#endif /\* TEXHNIQUE\_H \*/

**texture.cpp**

#include <iostream>

#include "texture.h"

Texture::Texture(GLenum TextureTarget, const std::string& FileName)

{

m\_textureTarget = TextureTarget;

m\_fileName = FileName;

m\_pImage = NULL;

}

bool Texture::Load()

{

try {

m\_pImage = new Magick::Image(m\_fileName);

m\_pImage->write(&m\_blob, "RGBA");

}

catch (Magick::Error& Error) {

std::cout << "Error loading texture '" << m\_fileName << "': " << Error.what() << std::endl;

return false;

}

glGenTextures(1, &m\_textureObj);

glBindTexture(m\_textureTarget, m\_textureObj);

glTexImage2D(m\_textureTarget, 0, GL\_RGB, m\_pImage->columns(), m\_pImage->rows(), -0.5, GL\_RGBA, GL\_UNSIGNED\_BYTE, m\_blob.data());

glTexParameterf(m\_textureTarget, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(m\_textureTarget, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

return true;

}

void Texture::Bind(GLenum TextureUnit)

{

glActiveTexture(TextureUnit);

glBindTexture(m\_textureTarget, m\_textureObj);

}

**technique.cpp**

#include <stdio.h>

#include <string.h>

#include "technique.h"

Technique::Technique(){

m\_shaderProg = 0;

}

Technique::~Technique(){

for (ShaderObjList::iterator it = m\_shaderObjList.begin(); it != m\_shaderObjList.end(); it++){

glDeleteShader(\*it);

}

if (m\_shaderProg != 0){

glDeleteProgram(m\_shaderProg);

m\_shaderProg = 0;

}

}

bool Technique::Init(){

m\_shaderProg = glCreateProgram();

if (m\_shaderProg == 0){

fprintf(stderr, "Error creating shader program\n");

return false;

}

return true;

}

//Используем этот метод для добавления шейдеров в программу. Когда заканчиваем - вызываем finalize()

bool Technique::AddShader(GLenum ShaderType, const char\* pShaderText){

GLuint ShaderObj = glCreateShader(ShaderType);

if (ShaderObj == 0){

fprintf(stderr, "Error creating shader type %d\n", ShaderType);

return false;

}

// Сохраним объект шейдера - он будет удален в декструкторе

m\_shaderObjList.push\_back(ShaderObj);

const GLchar\* p[1];

p[0] = pShaderText;

GLint Lengths[1];

Lengths[0] = strlen(pShaderText);

glShaderSource(ShaderObj, 1, p, Lengths);

glCompileShader(ShaderObj);

GLint success;

glGetShaderiv(ShaderObj, GL\_COMPILE\_STATUS, &success);

if (!success){

GLchar InfoLog[1024];

glGetShaderInfoLog(ShaderObj, 1024, NULL, InfoLog);

fprintf(stderr, "Error compiling shader type %d: '%s'\n", ShaderType, InfoLog);

return false;

}

glAttachShader(m\_shaderProg, ShaderObj);

return true;

}

// После добавления всех шейдеров в программу вызываем эту функцию

// для линковки и проверки программу на ошибки

bool Technique::Finalize(){

GLint Success = 0;

GLchar ErrorLog[1024] = {0};

glLinkProgram(m\_shaderProg);

glGetProgramiv(m\_shaderProg, GL\_LINK\_STATUS, &Success);

if (Success == 0){

glGetProgramInfoLog(m\_shaderProg, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Error linking shader program: '%s'\n", ErrorLog);

return false;

}

glValidateProgram(m\_shaderProg);

glGetProgramiv(m\_shaderProg, GL\_VALIDATE\_STATUS, &Success);

if (Success == 0){

glGetProgramInfoLog(m\_shaderProg, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Invalid shader program: '%s'\n", ErrorLog);

return false;

}

// Удаляем промежуточные объекты шейдеров, которые были добавлены в программу

for (ShaderObjList::iterator it = m\_shaderObjList.begin(); it != m\_shaderObjList.end(); it++){

glDeleteShader(\*it);

}

m\_shaderObjList.clear();

return true;

}

void Technique::Enable(){

glUseProgram(m\_shaderProg);

}

GLint Technique::GetUniformLocation(const char\* pUniformName){

GLint Location = glGetUniformLocation(m\_shaderProg, pUniformName);

if (Location == 0xFFFFFFFF){

fprintf(stderr, "Warning! Unable to get the location of uniform '%s'\n", pUniformName);

}

return Location;

}

**texture.h**

#ifndef TEXTURE\_H

#define TEXTURE\_H

#include <string>

#include <GL/glew.h>

#include <Magick++.h>

class Texture

{

public:

Texture(GLenum TextureTarget, const std::string& FileName);

bool Load();

void Bind(GLenum TextureUnit);

private:

std::string m\_fileName;

GLenum m\_textureTarget;

GLuint m\_textureObj;

Magick::Image\* m\_pImage;

Magick::Blob m\_blob;

};

#endif /\* TEXTURE\_H \*/

**util.h**

#ifndef UTIL\_H

#define UTIL\_H

#define ARRAY\_SIZE\_IN\_ELEMENTS(a) (sizeof(a)/sizeof(a[0]))

#endif /\* UTIL\_H \*/