/////阴影 fast\_realistic\_soft\_penumbra\_shadows//////////////

1>penumbra\_shadow\_mapping.vs

#version 120

uniform vec3 LightPosition, CameraPosition;

uniform mat4x4 LightTextureMatrices[5];

varying vec3 LightDirection, LightDirectionReflected, CameraDirection, Normal;

varying vec4 ShadowTexCoord[5];

void main()

{

LightDirection = LightPosition - gl\_Vertex.xyz;

LightDirectionReflected = reflect(-LightDirection, gl\_Normal);

CameraDirection = CameraPosition - gl\_Vertex.xyz;

Normal = gl\_Normal;

gl\_TexCoord[0] = gl\_MultiTexCoord0;

for(int i = 0; i < 5; i++) ShadowTexCoord[i] = LightTextureMatrices[i] \* gl\_Vertex;

gl\_Position = gl\_ModelViewProjectionMatrix \* gl\_Vertex;

}

2>penumbra\_shadow\_mapping.fs

#version 120

#extension GL\_EXT\_texture\_array : enable

uniform sampler2D Texture;

uniform sampler2DArrayShadow ShadowMap;

varying vec3 LightDirection, LightDirectionReflected, CameraDirection, Normal;

varying vec4 ShadowTexCoord[5];

void main()

{

float Shadow = 0.0;

for(int i = 0; i < 5; i++)

{

Shadow += shadow2DArray(ShadowMap, vec4(ShadowTexCoord[i].xy / ShadowTexCoord[i].w, i, ShadowTexCoord[i].z / ShadowTexCoord[i].w)).r;

}

Shadow /= 5.0;

float NdotLD = max(dot(normalize(LightDirection), Normal), 0.0) \* Shadow;

float Spec = pow(max(dot(normalize(LightDirectionReflected), normalize(CameraDirection)), 0.0), 32.0) \* Shadow;

gl\_FragColor = vec4(texture2D(Texture, gl\_TexCoord[0].st).rgb \* (0.25 + NdotLD \* 0.75 + Spec), 1.0);

}

3>Source

#include "opengl\_21\_tutorials\_win32\_framework.h"

// ----------------------------------------------------------------------------------------------------------------------------

CBuffer::CBuffer()

{

SetDefaults();

}

CBuffer::~CBuffer()

{

Empty();

}

void CBuffer::AddData(void \*Data, int DataSize)

{

int Remaining = BufferSize - Position;

if(DataSize > Remaining)

{

BYTE \*OldBuffer = Buffer;

int OldBufferSize = BufferSize;

int Needed = DataSize - Remaining;

BufferSize += Needed > BUFFER\_SIZE\_INCREMENT ? Needed : BUFFER\_SIZE\_INCREMENT;

Buffer = new BYTE[BufferSize];

memcpy(Buffer, OldBuffer, OldBufferSize);

delete [] OldBuffer;

}

memcpy(Buffer + Position, Data, DataSize);

Position += DataSize;

}

void CBuffer::Empty()

{

delete [] Buffer;

SetDefaults();

}

void \*CBuffer::GetData()

{

return Buffer;

}

int CBuffer::GetDataSize()

{

return Position;

}

void CBuffer::SetDefaults()

{

Buffer = NULL;

BufferSize = 0;

Position = 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int gl\_max\_texture\_size = 0, gl\_max\_texture\_max\_anisotropy\_ext = 0;

// ----------------------------------------------------------------------------------------------------------------------------

CTexture::CTexture()

{

Texture = 0;

}

CTexture::~CTexture()

{

}

CTexture::operator GLuint ()

{

return Texture;

}

bool CTexture::LoadTexture2D(char \*FileName)

{

CString DirectoryFileName = ModuleDirectory + FileName;

int Width, Height, BPP;

FIBITMAP \*dib = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileName);

FreeImage\_Unload(dib);

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_2D, Texture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_2D, GL\_GENERATE\_MIPMAP, GL\_TRUE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib));

glBindTexture(GL\_TEXTURE\_2D, 0);

FreeImage\_Unload(dib);

return true;

}

bool CTexture::LoadTextureCubeMap(char \*\*FileNames)

{

int Width, Height, BPP;

FIBITMAP \*dib[6];

bool Error = false;

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

{

CString DirectoryFileName = ModuleDirectory + FileNames[i];

dib[i] = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib[i] == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

Error = true;

}

}

if(Error)

{

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileNames[5]);

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, Texture);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_GENERATE\_MIPMAP, GL\_TRUE);

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

{

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib[i]));

}

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

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

{

FreeImage\_Unload(dib[i]);

}

return true;

}

void CTexture::Destroy()

{

glDeleteTextures(1, &Texture);

Texture = 0;

}

FIBITMAP \*CTexture::GetBitmap(char \*FileName, int &Width, int &Height, int &BPP)

{

FREE\_IMAGE\_FORMAT fif = FreeImage\_GetFileType(FileName);

if(fif == FIF\_UNKNOWN)

{

fif = FreeImage\_GetFIFFromFilename(FileName);

}

if(fif == FIF\_UNKNOWN)

{

return NULL;

}

FIBITMAP \*dib = NULL;

if(FreeImage\_FIFSupportsReading(fif))

{

dib = FreeImage\_Load(fif, FileName);

}

if(dib != NULL)

{

int OriginalWidth = FreeImage\_GetWidth(dib);

int OriginalHeight = FreeImage\_GetHeight(dib);

Width = OriginalWidth;

Height = OriginalHeight;

if(Width == 0 || Height == 0)

{

FreeImage\_Unload(dib);

return NULL;

}

BPP = FreeImage\_GetBPP(dib);

if(Width > gl\_max\_texture\_size) Width = gl\_max\_texture\_size;

if(Height > gl\_max\_texture\_size) Height = gl\_max\_texture\_size;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

Width = 1 << (int)floor((log((float)Width) / log(2.0f)) + 0.5f);

Height = 1 << (int)floor((log((float)Height) / log(2.0f)) + 0.5f);

}

if(Width != OriginalWidth || Height != OriginalHeight)

{

FIBITMAP \*rdib = FreeImage\_Rescale(dib, Width, Height, FILTER\_BICUBIC);

FreeImage\_Unload(dib);

dib = rdib;

}

}

return dib;

}

// ----------------------------------------------------------------------------------------------------------------------------

CShaderProgram::CShaderProgram()

{

SetDefaults();

}

CShaderProgram::~CShaderProgram()

{

}

CShaderProgram::operator GLuint ()

{

return Program;

}

bool CShaderProgram::Load(char \*VertexShaderFileName, char \*FragmentShaderFileName)

{

bool Error = false;

Destroy();

Error |= ((VertexShader = LoadShader(VertexShaderFileName, GL\_VERTEX\_SHADER)) == 0);

Error |= ((FragmentShader = LoadShader(FragmentShaderFileName, GL\_FRAGMENT\_SHADER)) == 0);

if(Error)

{

Destroy();

return false;

}

Program = glCreateProgram();

glAttachShader(Program, VertexShader);

glAttachShader(Program, FragmentShader);

glLinkProgram(Program);

int LinkStatus;

glGetProgramiv(Program, GL\_LINK\_STATUS, &LinkStatus);

if(LinkStatus == GL\_FALSE)

{

ErrorLog.Append("Error linking program (%s, %s)!\r\n", VertexShaderFileName, FragmentShaderFileName);

int InfoLogLength = 0;

glGetProgramiv(Program, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetProgramInfoLog(Program, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

Destroy();

return false;

}

return true;

}

void CShaderProgram::Destroy()

{

glDetachShader(Program, VertexShader);

glDetachShader(Program, FragmentShader);

glDeleteShader(VertexShader);

glDeleteShader(FragmentShader);

glDeleteProgram(Program);

delete [] UniformLocations;

delete [] AttribLocations;

SetDefaults();

}

GLuint CShaderProgram::LoadShader(char \*FileName, GLenum Type)

{

CString DirectoryFileName = ModuleDirectory + FileName;

FILE \*File;

if(fopen\_s(&File, DirectoryFileName, "rb") != 0)

{

ErrorLog.Append("Error loading file " + DirectoryFileName + "!\r\n");

return 0;

}

fseek(File, 0, SEEK\_END);

long Size = ftell(File);

fseek(File, 0, SEEK\_SET);

char \*Source = new char[Size + 1];

fread(Source, 1, Size, File);

fclose(File);

Source[Size] = 0;

GLuint Shader = glCreateShader(Type);

glShaderSource(Shader, 1, (const char\*\*)&Source, NULL);

delete [] Source;

glCompileShader(Shader);

int CompileStatus;

glGetShaderiv(Shader, GL\_COMPILE\_STATUS, &CompileStatus);

if(CompileStatus == GL\_FALSE)

{

ErrorLog.Append("Error compiling shader %s!\r\n", FileName);

int InfoLogLength = 0;

glGetShaderiv(Shader, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetShaderInfoLog(Shader, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

glDeleteShader(Shader);

return 0;

}

return Shader;

}

void CShaderProgram::SetDefaults()

{

VertexShader = 0;

FragmentShader = 0;

Program = 0;

UniformLocations = NULL;

AttribLocations = NULL;

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera::CCamera()

{

ViewMatrix = NULL;

ViewMatrixInverse = NULL;

X = vec3(1.0f, 0.0f, 0.0f);

Y = vec3(0.0f, 1.0f, 0.0f);

Z = vec3(0.0f, 0.0f, 1.0f);

Position = vec3(0.0f, 0.0f, 5.0f);

Reference = vec3(0.0f, 0.0f, 0.0f);

}

CCamera::~CCamera()

{

}

void CCamera::Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference)

{

this->Position = Position;

this->Reference = Reference;

Z = normalize(Position - Reference);

X = normalize(cross(vec3(0.0f, 1.0f, 0.0f), Z));

Y = cross(Z, X);

if(!RotateAroundReference)

{

this->Reference = this->Position;

this->Position += Z \* 0.05f;

}

CalculateViewMatrix();

}

void CCamera::Move(const vec3 &Movement)

{

Position += Movement;

Reference += Movement;

CalculateViewMatrix();

}

vec3 CCamera::OnKeys(BYTE Keys, float FrameTime)

{

float Speed = 5.0f;

if(Keys & 0x40) Speed \*= 2.0f;

if(Keys & 0x80) Speed \*= 0.5f;

float Distance = Speed \* FrameTime;

vec3 Up(0.0f, 1.0f, 0.0f);

vec3 Right = X;

vec3 Forward = cross(Up, Right);

Up \*= Distance;

Right \*= Distance;

Forward \*= Distance;

vec3 Movement;

if(Keys & 0x01) Movement += Forward;

if(Keys & 0x02) Movement -= Forward;

if(Keys & 0x04) Movement -= Right;

if(Keys & 0x08) Movement += Right;

if(Keys & 0x10) Movement += Up;

if(Keys & 0x20) Movement -= Up;

return Movement;

}

void CCamera::OnMouseMove(int dx, int dy)

{

float Sensitivity = 0.25f;

Position -= Reference;

if(dx != 0)

{

float DeltaX = (float)dx \* Sensitivity;

X = rotate(X, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Y = rotate(Y, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Z = rotate(Z, DeltaX, vec3(0.0f, 1.0f, 0.0f));

}

if(dy != 0)

{

float DeltaY = (float)dy \* Sensitivity;

Y = rotate(Y, DeltaY, X);

Z = rotate(Z, DeltaY, X);

if(Y.y < 0.0f)

{

Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);

Y = cross(Z, X);

}

}

Position = Reference + Z \* length(Position);

CalculateViewMatrix();

}

void CCamera::OnMouseWheel(float zDelta)

{

Position -= Reference;

if(zDelta < 0 && length(Position) < 500.0f)

{

Position += Position \* 0.1f;

}

if(zDelta > 0 && length(Position) > 0.05f)

{

Position -= Position \* 0.1f;

}

Position += Reference;

CalculateViewMatrix();

}

void CCamera::SetViewMatrixPointer(float \*ViewMatrix, float \*ViewMatrixInverse)

{

this->ViewMatrix = (mat4x4\*)ViewMatrix;

this->ViewMatrixInverse = (mat4x4\*)ViewMatrixInverse;

CalculateViewMatrix();

}

void CCamera::CalculateViewMatrix()

{

if(ViewMatrix != NULL)

{

\*ViewMatrix = mat4x4(X.x, Y.x, Z.x, 0.0f, X.y, Y.y, Z.y, 0.0f, X.z, Y.z, Z.z, 0.0f, -dot(X, Position), -dot(Y, Position), -dot(Z, Position), 1.0f);

if(ViewMatrixInverse != NULL)

{

\*ViewMatrixInverse = inverse(\*ViewMatrix);

}

}

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera Camera;

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer::COpenGLRenderer()

{

Pause = false;

Radius = 0.05f;

Camera.SetViewMatrixPointer(&ViewMatrix);

}

COpenGLRenderer::~COpenGLRenderer()

{

}

bool COpenGLRenderer::Init()

{

// ------------------------------------------------------------------------------------------------------------------------

bool Error = false;

// check OpenGL extensions ------------------------------------------------------------------------------------------------

if(!GLEW\_ARB\_depth\_texture)

{

ErrorLog.Append("GL\_ARB\_depth\_texture not supported!\r\n");

Error = true;

}

if(!GLEW\_EXT\_texture\_array)

{

ErrorLog.Append("GL\_EXT\_texture\_array not supported!\r\n");

Error = true;

}

if(!GLEW\_EXT\_framebuffer\_object)

{

ErrorLog.Append("GL\_EXT\_framebuffer\_object not supported!\r\n");

Error = true;

}

// load textures and shaders ----------------------------------------------------------------------------------------------

Error |= !Texture.LoadTexture2D("texture.jpg");

Error |= !PenumbraShadowMapping.Load("penumbra\_shadow\_mapping.vs", "penumbra\_shadow\_mapping.fs");

// if an error occurred, return false -------------------------------------------------------------------------------------

if(Error)

{

return false;

}

// get uniform locations --------------------------------------------------------------------------------------------------

PenumbraShadowMapping.UniformLocations = new GLuint[3];

PenumbraShadowMapping.UniformLocations[0] = glGetUniformLocation(PenumbraShadowMapping, "LightPosition");

PenumbraShadowMapping.UniformLocations[1] = glGetUniformLocation(PenumbraShadowMapping, "CameraPosition");

PenumbraShadowMapping.UniformLocations[2] = glGetUniformLocation(PenumbraShadowMapping, "LightTextureMatrices");

// set constant uniforms --------------------------------------------------------------------------------------------------

glUseProgram(PenumbraShadowMapping);

glUniform1i(glGetUniformLocation(PenumbraShadowMapping, "Texture"), 0);

glUniform1i(glGetUniformLocation(PenumbraShadowMapping, "ShadowMap"), 1);

glUseProgram(0);

// init array buffers -----------------------------------------------------------------------------------------------------

float Data[] =

{ // s, t, nx, ny, nz, x, y, z

0.0f, 0.0f, 0.0f, 1.0f, 0.0f, -10.0f, -0.5f, 10.0f,

20.0f, 0.0f, 0.0f, 1.0f, 0.0f, 10.0f, -0.5f, 10.0f,

20.0f, 20.0f, 0.0f, 1.0f, 0.0f, 10.0f, -0.5f, -10.0f,

0.0f, 20.0f, 0.0f, 1.0f, 0.0f, -10.0f, -0.5f, -10.0f,

0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.5f, -0.5f, 0.5f,

1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.5f, -0.5f, -0.5f,

1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.5f, 0.5f, -0.5f,

0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.5f, 0.5f, 0.5f,

0.0f, 0.0f, -1.0f, 0.0f, 0.0f, 0.5f, -0.5f, -0.5f,

1.0f, 0.0f, -1.0f, 0.0f, 0.0f, 0.5f, -0.5f, 0.5f,

1.0f, 1.0f, -1.0f, 0.0f, 0.0f, 0.5f, 0.5f, 0.5f,

0.0f, 1.0f, -1.0f, 0.0f, 0.0f, 0.5f, 0.5f, -0.5f,

0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 0.5f, 0.5f,

1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.5f, 0.5f, 0.5f,

1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.5f, 0.5f, -0.5f,

0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.5f, 0.5f, -0.5f,

0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.5f, -0.5f, -0.5f,

1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.5f, -0.5f, -0.5f,

1.0f, 1.0f, 0.0f, -1.0f, 0.0f, 1.5f, -0.5f, 0.5f,

0.0f, 1.0f, 0.0f, -1.0f, 0.0f, 0.5f, -0.5f, 0.5f,

0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.5f, -0.5f, 0.5f,

1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.5f, -0.5f, 0.5f,

1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.5f, 0.5f, 0.5f,

0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.5f, 0.5f, 0.5f,

0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 1.5f, -0.5f, -0.5f,

1.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.5f, -0.5f, -0.5f,

1.0f, 1.0f, 0.0f, 0.0f, -1.0f, 0.5f, 0.5f, -0.5f,

0.0f, 1.0f, 0.0f, 0.0f, -1.0f, 1.5f, 0.5f, -0.5f,

0.0f, 0.0f, 1.0f, 0.0f, 0.0f, -0.5f, -0.5f, 0.5f,

1.0f, 0.0f, 1.0f, 0.0f, 0.0f, -0.5f, -0.5f, -0.5f,

1.0f, 1.0f, 1.0f, 0.0f, 0.0f, -0.5f, 0.5f, -0.5f,

0.0f, 1.0f, 1.0f, 0.0f, 0.0f, -0.5f, 0.5f, 0.5f,

0.0f, 0.0f, -1.0f, 0.0f, 0.0f, -1.5f, -0.5f, -0.5f,

1.0f, 0.0f, -1.0f, 0.0f, 0.0f, -1.5f, -0.5f, 0.5f,

1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.5f, 0.5f, 0.5f,

0.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.5f, 0.5f, -0.5f,

0.0f, 0.0f, 0.0f, 1.0f, 0.0f, -1.5f, 0.5f, 0.5f,

1.0f, 0.0f, 0.0f, 1.0f, 0.0f, -0.5f, 0.5f, 0.5f,

1.0f, 1.0f, 0.0f, 1.0f, 0.0f, -0.5f, 0.5f, -0.5f,

0.0f, 1.0f, 0.0f, 1.0f, 0.0f, -1.5f, 0.5f, -0.5f,

0.0f, 0.0f, 0.0f, -1.0f, 0.0f, -1.5f, -0.5f, -0.5f,

1.0f, 0.0f, 0.0f, -1.0f, 0.0f, -0.5f, -0.5f, -0.5f,

1.0f, 1.0f, 0.0f, -1.0f, 0.0f, -0.5f, -0.5f, 0.5f,

0.0f, 1.0f, 0.0f, -1.0f, 0.0f, -1.5f, -0.5f, 0.5f,

0.0f, 0.0f, 0.0f, 0.0f, 1.0f, -1.5f, -0.5f, 0.5f,

1.0f, 0.0f, 0.0f, 0.0f, 1.0f, -0.5f, -0.5f, 0.5f,

1.0f, 1.0f, 0.0f, 0.0f, 1.0f, -0.5f, 0.5f, 0.5f,

0.0f, 1.0f, 0.0f, 0.0f, 1.0f, -1.5f, 0.5f, 0.5f,

0.0f, 0.0f, 0.0f, 0.0f, -1.0f, -0.5f, -0.5f, -0.5f,

1.0f, 0.0f, 0.0f, 0.0f, -1.0f, -1.5f, -0.5f, -0.5f,

1.0f, 1.0f, 0.0f, 0.0f, -1.0f, -1.5f, 0.5f, -0.5f,

0.0f, 1.0f, 0.0f, 0.0f, -1.0f, -0.5f, 0.5f, -0.5f

};

glGenBuffers(1, &VBO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, 1664, Data, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

// generate shadow map texture --------------------------------------------------------------------------------------------

glGenTextures(1, &ShadowMap);

glBindTexture(GL\_TEXTURE\_2D\_ARRAY, ShadowMap);

glTexParameteri(GL\_TEXTURE\_2D\_ARRAY, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D\_ARRAY, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D\_ARRAY, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D\_ARRAY, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D\_ARRAY, GL\_TEXTURE\_COMPARE\_MODE, GL\_COMPARE\_R\_TO\_TEXTURE);

glTexParameteri(GL\_TEXTURE\_2D\_ARRAY, GL\_TEXTURE\_COMPARE\_FUNC, GL\_LEQUAL);

glTexParameteri(GL\_TEXTURE\_2D\_ARRAY, GL\_DEPTH\_TEXTURE\_MODE, GL\_INTENSITY);

glTexImage3D(GL\_TEXTURE\_2D\_ARRAY, 0, GL\_DEPTH\_COMPONENT24, SHADOW\_MAP\_SIZE, SHADOW\_MAP\_SIZE, 5, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL);

glBindTexture(GL\_TEXTURE\_2D\_ARRAY, 0);

// generate FBO -----------------------------------------------------------------------------------------------------------

glGenFramebuffersEXT(1, &FBO);

// set light position and projection matrix -------------------------------------------------------------------------------

LightPosition = vec3(0.0f, 2.5f, -5.0f);

LightProjectionMatrix = perspective(45.0f, 1.0f, 0.125f, 512.0f);

// set camera -------------------------------------------------------------------------------------------------------------

Camera.Look(vec3(0.0f, 1.25f, 5.0f), vec3(0.0f, -0.5f, 0.0f));

// ------------------------------------------------------------------------------------------------------------------------

return true;

}

void COpenGLRenderer::RenderShadowMap()

{

// calculate light matrices -----------------------------------------------------------------------------------------------

LightViewMatrices[0] = look(LightPosition, vec3(0.0f), vec3(0.0f, 1.0f, 0.0f));

LightTextureMatrices[0] = BiasMatrix \* LightProjectionMatrix \* LightViewMatrices[0];

vec3 lp = vec3(0.0f);

lp += vec3(LightViewMatrices[0][0], LightViewMatrices[0][4], LightViewMatrices[0][8]);

lp += vec3(LightViewMatrices[0][1], LightViewMatrices[0][5], LightViewMatrices[0][9]);

lp = normalize(lp);

lp \*= Radius;

lp += LightPosition;

for(int i = 1; i < 5; i++)

{

LightViewMatrices[i] = look(lp, vec3(0.0f), vec3(0.0f, 1.0f, 0.0f));

LightTextureMatrices[i] = BiasMatrix \* LightProjectionMatrix \* LightViewMatrices[i];

lp = rotate(lp, 90.0f, vec3(LightViewMatrices[0][2], LightViewMatrices[0][6], LightViewMatrices[0][10]));

}

// render shadow map ------------------------------------------------------------------------------------------------------

glViewport(0, 0, SHADOW\_MAP\_SIZE, SHADOW\_MAP\_SIZE);

for(int i = 0; i < 5; i++)

{

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glDrawBuffers(0, NULL); glReadBuffer(GL\_NONE);

glFramebufferTextureLayerEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, ShadowMap, 0, i);

glClear(GL\_DEPTH\_BUFFER\_BIT);

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&LightProjectionMatrix);

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&LightViewMatrices[i]);

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

glCullFace(GL\_FRONT);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(3, GL\_FLOAT, 32, (void\*)20);

glDrawArrays(GL\_QUADS, 0, 52);

glDisableClientState(GL\_VERTEX\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glCullFace(GL\_BACK);

glDisable(GL\_CULL\_FACE);

glDisable(GL\_DEPTH\_TEST);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

}

}

void COpenGLRenderer::Render(float FrameTime)

{

// render shadow map ------------------------------------------------------------------------------------------------------

if(!Pause)

{

RenderShadowMap();

}

// render scene -----------------------------------------------------------------------------------------------------------

glViewport(0, 0, Width, Height);

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

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&ProjectionMatrix);

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, Texture);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D\_ARRAY, ShadowMap);

glUseProgram(PenumbraShadowMapping);

glUniform3fv(PenumbraShadowMapping.UniformLocations[0], 1, &LightPosition);

glUniform3fv(PenumbraShadowMapping.UniformLocations[1], 1, &Camera.Position);

glUniformMatrix4fv(PenumbraShadowMapping.UniformLocations[2], 5, GL\_FALSE, (GLfloat\*)LightTextureMatrices);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glTexCoordPointer(2, GL\_FLOAT, 32, (void\*)0);

glEnableClientState(GL\_NORMAL\_ARRAY);

glNormalPointer(GL\_FLOAT, 32, (void\*)8);

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(3, GL\_FLOAT, 32, (void\*)20);

glDrawArrays(GL\_QUADS, 0, 52);

glDisableClientState(GL\_VERTEX\_ARRAY);

glDisableClientState(GL\_NORMAL\_ARRAY);

glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glUseProgram(0);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D\_ARRAY, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, 0);

glDisable(GL\_CULL\_FACE);

glDisable(GL\_DEPTH\_TEST);

// move light -------------------------------------------------------------------------------------------------------------

if(!Pause)

{

LightPosition = rotate(LightPosition, 2.8125f \* FrameTime, vec3(0.0f, 1.0f, 0.0f));

}

// ------------------------------------------------------------------------------------------------------------------------

Text.Set("Radius: %.03f", Radius);

}

void COpenGLRenderer::Resize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

ProjectionMatrix = perspective(45.0f, (float)Width / (float)Height, 0.125f, 512.0f);

}

void COpenGLRenderer::Destroy()

{

Texture.Destroy();

PenumbraShadowMapping.Destroy();

glDeleteBuffers(1, &VBO);

glDeleteTextures(1, &ShadowMap);

if(GLEW\_EXT\_framebuffer\_object)

{

glDeleteFramebuffersEXT(1, &FBO);

}

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer OpenGLRenderer;

// ----------------------------------------------------------------------------------------------------------------------------

CString ModuleDirectory, ErrorLog;

// ----------------------------------------------------------------------------------------------------------------------------

void GetModuleDirectory()

{

char \*moduledirectory = new char[256];

GetModuleFileName(GetModuleHandle(NULL), moduledirectory, 256);

\*(strrchr(moduledirectory, '\\') + 1) = 0;

ModuleDirectory = moduledirectory;

delete [] moduledirectory;

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView::COpenGLView()

{

}

COpenGLView::~COpenGLView()

{

}

bool COpenGLView::Init(HINSTANCE hInstance, char \*Title, int Width, int Height, int Samples)

{

this->Title = Title;

this->Width = Width;

this->Height = Height;

WNDCLASSEX WndClassEx;

memset(&WndClassEx, 0, sizeof(WNDCLASSEX));

WndClassEx.cbSize = sizeof(WNDCLASSEX);

WndClassEx.style = CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

WndClassEx.lpfnWndProc = WndProc;

WndClassEx.hInstance = hInstance;

WndClassEx.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hCursor = LoadCursor(NULL, IDC\_ARROW);

WndClassEx.lpszClassName = "Win32OpenGLWindowClass";

if(RegisterClassEx(&WndClassEx) == 0)

{

ErrorLog.Set("RegisterClassEx failed!");

return false;

}

DWORD Style = WS\_OVERLAPPEDWINDOW | WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN;

hWnd = CreateWindowEx(WS\_EX\_APPWINDOW, WndClassEx.lpszClassName, Title, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL);

if(hWnd == NULL)

{

ErrorLog.Set("CreateWindowEx failed!");

return false;

}

HDC hDC = GetDC(hWnd);

if(hDC == NULL)

{

ErrorLog.Set("GetDC failed!");

return false;

}

PIXELFORMATDESCRIPTOR pfd;

memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));

pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);

pfd.nVersion = 1;

pfd.dwFlags = PFD\_DRAW\_TO\_WINDOW | PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER;

pfd.iPixelType = PFD\_TYPE\_RGBA;

pfd.cColorBits = 32;

pfd.cDepthBits = 24;

pfd.iLayerType = PFD\_MAIN\_PLANE;

int PixelFormat = ChoosePixelFormat(hDC, &pfd);

if(PixelFormat == 0)

{

ErrorLog.Set("ChoosePixelFormat failed!");

return false;

}

static int MSAAPixelFormat = 0;

if(SetPixelFormat(hDC, MSAAPixelFormat == 0 ? PixelFormat : MSAAPixelFormat, &pfd) == FALSE)

{

ErrorLog.Set("SetPixelFormat failed!");

return false;

}

hGLRC = wglCreateContext(hDC);

if(hGLRC == NULL)

{

ErrorLog.Set("wglCreateContext failed!");

return false;

}

if(wglMakeCurrent(hDC, hGLRC) == FALSE)

{

ErrorLog.Set("wglMakeCurrent failed!");

return false;

}

if(glewInit() != GLEW\_OK)

{

ErrorLog.Set("glewInit failed!");

return false;

}

if(!GLEW\_VERSION\_2\_1)

{

ErrorLog.Set("OpenGL 2.1 not supported!");

return false;

}

if(MSAAPixelFormat == 0 && Samples > 0)

{

if(GLEW\_ARB\_multisample && WGLEW\_ARB\_pixel\_format)

{

while(Samples > 0)

{

UINT NumFormats = 0;

int PFAttribs[] =

{

WGL\_DRAW\_TO\_WINDOW\_ARB, GL\_TRUE,

WGL\_SUPPORT\_OPENGL\_ARB, GL\_TRUE,

WGL\_DOUBLE\_BUFFER\_ARB, GL\_TRUE,

WGL\_PIXEL\_TYPE\_ARB, WGL\_TYPE\_RGBA\_ARB,

WGL\_COLOR\_BITS\_ARB, 32,

WGL\_DEPTH\_BITS\_ARB, 24,

WGL\_ACCELERATION\_ARB, WGL\_FULL\_ACCELERATION\_ARB,

WGL\_SAMPLE\_BUFFERS\_ARB, GL\_TRUE,

WGL\_SAMPLES\_ARB, Samples,

0

};

if(wglChoosePixelFormatARB(hDC, PFAttribs, NULL, 1, &MSAAPixelFormat, &NumFormats) == TRUE && NumFormats > 0) break;

Samples--;

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

UnregisterClass(WndClassEx.lpszClassName, hInstance);

return Init(hInstance, Title, Width, Height, Samples);

}

else

{

Samples = 0;

}

}

this->Samples = Samples;

GetModuleDirectory();

glGetIntegerv(GL\_MAX\_TEXTURE\_SIZE, &gl\_max\_texture\_size);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glGetIntegerv(GL\_MAX\_TEXTURE\_MAX\_ANISOTROPY\_EXT, &gl\_max\_texture\_max\_anisotropy\_ext);

}

if(WGLEW\_EXT\_swap\_control)

{

wglSwapIntervalEXT(0);

}

return OpenGLRenderer.Init();

}

void COpenGLView::Show(bool Maximized)

{

RECT dRect, wRect, cRect;

GetWindowRect(GetDesktopWindow(), &dRect);

GetWindowRect(hWnd, &wRect);

GetClientRect(hWnd, &cRect);

wRect.right += Width - cRect.right;

wRect.bottom += Height - cRect.bottom;

wRect.right -= wRect.left;

wRect.bottom -= wRect.top;

wRect.left = dRect.right / 2 - wRect.right / 2;

wRect.top = dRect.bottom / 2 - wRect.bottom / 2;

MoveWindow(hWnd, wRect.left, wRect.top, wRect.right, wRect.bottom, FALSE);

ShowWindow(hWnd, Maximized ? SW\_SHOWMAXIMIZED : SW\_SHOWNORMAL);

}

void COpenGLView::MessageLoop()

{

MSG Msg;

while(GetMessage(&Msg, NULL, 0, 0) > 0)

{

TranslateMessage(&Msg);

DispatchMessage(&Msg);

}

}

void COpenGLView::Destroy()

{

if(GLEW\_VERSION\_2\_1)

{

OpenGLRenderer.Destroy();

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

}

void COpenGLView::OnKeyDown(UINT Key)

{

switch(Key)

{

case VK\_SPACE:

OpenGLRenderer.Pause = !OpenGLRenderer.Pause;

break;

case VK\_ADD:

OpenGLRenderer.Radius += 0.001f;

if(OpenGLRenderer.Pause) OpenGLRenderer.RenderShadowMap();

break;

case VK\_SUBTRACT:

OpenGLRenderer.Radius -= 0.001f;

if(OpenGLRenderer.Radius < 0.0f) OpenGLRenderer.Radius = 0.0f;

if(OpenGLRenderer.Pause) OpenGLRenderer.RenderShadowMap();

break;

}

}

void COpenGLView::OnMouseMove(int X, int Y)

{

if(GetKeyState(VK\_RBUTTON) & 0x80)

{

Camera.OnMouseMove(LastX - X, LastY - Y);

LastX = X;

LastY = Y;

}

}

void COpenGLView::OnMouseWheel(short zDelta)

{

Camera.OnMouseWheel(zDelta);

}

void COpenGLView::OnPaint()

{

static DWORD LastFPSTime = GetTickCount(), LastFrameTime = LastFPSTime, FPS = 0;

PAINTSTRUCT ps;

HDC hDC = BeginPaint(hWnd, &ps);

DWORD Time = GetTickCount();

float FrameTime = (Time - LastFrameTime) \* 0.001f;

LastFrameTime = Time;

if(Time - LastFPSTime > 1000)

{

CString Text = Title;

if(OpenGLRenderer.Text[0] != 0)

{

Text.Append(" - " + OpenGLRenderer.Text);

}

Text.Append(" - %dx%d", Width, Height);

Text.Append(", ATF %dx", gl\_max\_texture\_max\_anisotropy\_ext);

Text.Append(", MSAA %dx", Samples);

Text.Append(", FPS: %d", FPS);

Text.Append(" - %s", glGetString(GL\_RENDERER));

SetWindowText(hWnd, Text);

LastFPSTime = Time;

FPS = 0;

}

else

{

FPS++;

}

BYTE Keys = 0x00;

if(GetKeyState('W') & 0x80) Keys |= 0x01;

if(GetKeyState('S') & 0x80) Keys |= 0x02;

if(GetKeyState('A') & 0x80) Keys |= 0x04;

if(GetKeyState('D') & 0x80) Keys |= 0x08;

if(GetKeyState('R') & 0x80) Keys |= 0x10;

if(GetKeyState('F') & 0x80) Keys |= 0x20;

if(GetKeyState(VK\_SHIFT) & 0x80) Keys |= 0x40;

if(GetKeyState(VK\_CONTROL) & 0x80) Keys |= 0x80;

if(Keys & 0x3F)

{

Camera.Move(Camera.OnKeys(Keys, FrameTime));

}

OpenGLRenderer.Render(FrameTime);

SwapBuffers(hDC);

EndPaint(hWnd, &ps);

InvalidateRect(hWnd, NULL, FALSE);

}

void COpenGLView::OnRButtonDown(int X, int Y)

{

LastX = X;

LastY = Y;

}

void COpenGLView::OnSize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

OpenGLRenderer.Resize(Width, Height);

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView OpenGLView;

// ----------------------------------------------------------------------------------------------------------------------------

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)

{

switch(uiMsg)

{

case WM\_CLOSE:

PostQuitMessage(0);

break;

case WM\_MOUSEMOVE:

OpenGLView.OnMouseMove(LOWORD(lParam), HIWORD(lParam));

break;

case 0x020A: // WM\_MOUSWHEEL

OpenGLView.OnMouseWheel(HIWORD(wParam));

break;

case WM\_KEYDOWN:

OpenGLView.OnKeyDown((UINT)wParam);

break;

case WM\_PAINT:

OpenGLView.OnPaint();

break;

case WM\_RBUTTONDOWN:

OpenGLView.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_SIZE:

OpenGLView.OnSize(LOWORD(lParam), HIWORD(lParam));

break;

default:

return DefWindowProc(hWnd, uiMsg, wParam, lParam);

}

return 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine, int iShow)

{

char \*AppName = "Fast realistic soft penumbra shadows";

if(OpenGLView.Init(hInstance, AppName, 800, 600, 4))

{

OpenGLView.Show();

OpenGLView.MessageLoop();

}

else

{

MessageBox(NULL, ErrorLog, AppName, MB\_OK | MB\_ICONERROR);

}

OpenGLView.Destroy();

return 0;

}

//////////////////////////////////////////////////////插图////////////////////////////////////////////////

/////////////////////////////////////////////////////sun\_rays\_lens\_flare\_halo///////////////////////////////////////////////

1. sunrayslensflarehalo.vs
2. #version 120
3. void main(void)
4. {
5. gl\_TexCoord[0] = gl\_Vertex;
6. gl\_Position = gl\_Vertex \* 2.0 - 1.0;
7. }
8. 2>sunrayslensflarehalo.fs

#version 120

uniform sampler2D LowBlurredSunTexture, HighBlurredSunTexture, DirtTexture;

uniform float Dispersal, HaloWidth, Intensity;

uniform vec2 SunPosProj;

uniform vec3 Distortion;

vec3 texture2DDistorted(sampler2D Texture, vec2 TexCoord, vec2 Offset)

{

return vec3(

texture2D(Texture, TexCoord + Offset \* Distortion.r).r,

texture2D(Texture, TexCoord + Offset \* Distortion.g).g,

texture2D(Texture, TexCoord + Offset \* Distortion.b).b

);

}

void main()

{

vec3 RadialBlur = vec3(0.0);

vec2 TexCoord = gl\_TexCoord[0].st;

int RadialBlurSamples = 128;

vec2 RadialBlurVector = (SunPosProj - TexCoord) / RadialBlurSamples;

for(int i = 0; i < RadialBlurSamples; i++)

{

RadialBlur += texture2D(LowBlurredSunTexture, TexCoord).rgb;

TexCoord += RadialBlurVector;

}

RadialBlur /= RadialBlurSamples;

vec3 LensFlareHalo = vec3(0.0);

TexCoord = 1.0 - gl\_TexCoord[0].st;

vec2 LensFlareVector = (vec2(0.5) - TexCoord) \* Dispersal;

vec2 LensFlareOffset = vec2(0.0);

for(int i = 0; i < 5; i++)

{

LensFlareHalo += texture2DDistorted(HighBlurredSunTexture, TexCoord, LensFlareOffset).rgb;

LensFlareOffset += LensFlareVector;

}

LensFlareHalo += texture2DDistorted(HighBlurredSunTexture, TexCoord, normalize(LensFlareVector) \* HaloWidth);

LensFlareHalo /= 6.0;

gl\_FragColor = vec4((texture2D(HighBlurredSunTexture, gl\_TexCoord[0].st).rgb + (RadialBlur + LensFlareHalo) \* texture2D(DirtTexture, gl\_TexCoord[0].st).rgb) \* Intensity, 1.0);

}

3>sundepthtest.vs

#version 120

void main()

{

gl\_TexCoord[0] = gl\_Vertex;

gl\_Position = gl\_Vertex \* 2.0 - 1.0;

}

4>sundepthtest.fs

#version 120

uniform sampler2D SunTexture, SunDepthTexture, DepthTexture;

void main()

{

if(texture2D(DepthTexture, gl\_TexCoord[0].st).r < texture2D(SunDepthTexture, gl\_TexCoord[0].st).r)

{

gl\_FragColor = vec4(vec3(0.0), 1.0);

}

else

{

gl\_FragColor = vec4(texture2D(SunTexture, gl\_TexCoord[0].st).rgb, 1.0);

}

}  
5>blur.vs

#version 120

void main(void)

{

gl\_TexCoord[0] = gl\_Vertex;

gl\_Position = gl\_Vertex \* 2.0 - 1.0;

}

6>blurh.fs

#version 120

uniform sampler2D Texture;

uniform int Width;

uniform float odw;

void main()

{

vec3 Color = vec3(0.0);

int wp1 = Width + 1;

float Sum = 0.0;

for(int x = -Width; x <= Width; x++)

{

float width = (wp1 - abs(float(x)));

Color += texture2D(Texture, gl\_TexCoord[0].st + vec2(odw \* x, 0.0)).rgb \* width;

Sum += width;

}

gl\_FragColor = vec4(Color / Sum, 1.0);

}

7>blurv.fs

#version 120

uniform sampler2D Texture;

uniform int Width;

uniform float odh;

void main()

{

vec3 Color = vec3(0.0);

int wp1 = Width + 1;

float Sum = 0.0;

for(int y = -Width; y <= Width; y++)

{

float width = (wp1 - abs(float(y)));

Color += texture2D(Texture, gl\_TexCoord[0].st + vec2(0.0, odh \* y)).rgb \* width;

Sum += width;

}

gl\_FragColor = vec4(Color / Sum, 1.0);

}

///////////////////////////Source////////////////////////////////////////

#include "opengl\_21\_tutorials\_win32\_framework.h"

// ----------------------------------------------------------------------------------------------------------------------------

CBuffer::CBuffer()

{

SetDefaults();

}

CBuffer::~CBuffer()

{

Empty();

}

void CBuffer::AddData(void \*Data, int DataSize)

{

int Remaining = BufferSize - Position;

if(DataSize > Remaining)

{

BYTE \*OldBuffer = Buffer;

int OldBufferSize = BufferSize;

int Needed = DataSize - Remaining;

BufferSize += Needed > BUFFER\_SIZE\_INCREMENT ? Needed : BUFFER\_SIZE\_INCREMENT;

Buffer = new BYTE[BufferSize];

memcpy(Buffer, OldBuffer, OldBufferSize);

delete [] OldBuffer;

}

memcpy(Buffer + Position, Data, DataSize);

Position += DataSize;

}

void CBuffer::Empty()

{

delete [] Buffer;

SetDefaults();

}

void \*CBuffer::GetData()

{

return Buffer;

}

int CBuffer::GetDataSize()

{

return Position;

}

void CBuffer::SetDefaults()

{

Buffer = NULL;

BufferSize = 0;

Position = 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int gl\_max\_texture\_size = 0, gl\_max\_texture\_max\_anisotropy\_ext = 0;

// ----------------------------------------------------------------------------------------------------------------------------

CTexture::CTexture()

{

Texture = 0;

}

CTexture::~CTexture()

{

}

CTexture::operator GLuint ()

{

return Texture;

}

bool CTexture::LoadTexture2D(char \*FileName)

{

CString DirectoryFileName = ModuleDirectory + FileName;

int Width, Height, BPP;

FIBITMAP \*dib = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileName);

FreeImage\_Unload(dib);

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_2D, Texture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_2D, GL\_GENERATE\_MIPMAP, GL\_TRUE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib));

glBindTexture(GL\_TEXTURE\_2D, 0);

FreeImage\_Unload(dib);

return true;

}

bool CTexture::LoadTextureCubeMap(char \*\*FileNames)

{

int Width, Height, BPP;

FIBITMAP \*dib[6];

bool Error = false;

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

{

CString DirectoryFileName = ModuleDirectory + FileNames[i];

dib[i] = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib[i] == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

Error = true;

}

}

if(Error)

{

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileNames[5]);

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, Texture);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_GENERATE\_MIPMAP, GL\_TRUE);

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

{

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib[i]));

}

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

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

{

FreeImage\_Unload(dib[i]);

}

return true;

}

void CTexture::Destroy()

{

glDeleteTextures(1, &Texture);

Texture = 0;

}

FIBITMAP \*CTexture::GetBitmap(char \*FileName, int &Width, int &Height, int &BPP)

{

FREE\_IMAGE\_FORMAT fif = FreeImage\_GetFileType(FileName);

if(fif == FIF\_UNKNOWN)

{

fif = FreeImage\_GetFIFFromFilename(FileName);

}

if(fif == FIF\_UNKNOWN)

{

return NULL;

}

FIBITMAP \*dib = NULL;

if(FreeImage\_FIFSupportsReading(fif))

{

dib = FreeImage\_Load(fif, FileName);

}

if(dib != NULL)

{

int OriginalWidth = FreeImage\_GetWidth(dib);

int OriginalHeight = FreeImage\_GetHeight(dib);

Width = OriginalWidth;

Height = OriginalHeight;

if(Width == 0 || Height == 0)

{

FreeImage\_Unload(dib);

return NULL;

}

BPP = FreeImage\_GetBPP(dib);

if(Width > gl\_max\_texture\_size) Width = gl\_max\_texture\_size;

if(Height > gl\_max\_texture\_size) Height = gl\_max\_texture\_size;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

Width = 1 << (int)floor((log((float)Width) / log(2.0f)) + 0.5f);

Height = 1 << (int)floor((log((float)Height) / log(2.0f)) + 0.5f);

}

if(Width != OriginalWidth || Height != OriginalHeight)

{

FIBITMAP \*rdib = FreeImage\_Rescale(dib, Width, Height, FILTER\_BICUBIC);

FreeImage\_Unload(dib);

dib = rdib;

}

}

return dib;

}

// ----------------------------------------------------------------------------------------------------------------------------

CShaderProgram::CShaderProgram()

{

SetDefaults();

}

CShaderProgram::~CShaderProgram()

{

}

CShaderProgram::operator GLuint ()

{

return Program;

}

bool CShaderProgram::Load(char \*VertexShaderFileName, char \*FragmentShaderFileName)

{

bool Error = false;

Destroy();

Error |= ((VertexShader = LoadShader(VertexShaderFileName, GL\_VERTEX\_SHADER)) == 0);

Error |= ((FragmentShader = LoadShader(FragmentShaderFileName, GL\_FRAGMENT\_SHADER)) == 0);

if(Error)

{

Destroy();

return false;

}

Program = glCreateProgram();

glAttachShader(Program, VertexShader);

glAttachShader(Program, FragmentShader);

glLinkProgram(Program);

int LinkStatus;

glGetProgramiv(Program, GL\_LINK\_STATUS, &LinkStatus);

if(LinkStatus == GL\_FALSE)

{

ErrorLog.Append("Error linking program (%s, %s)!\r\n", VertexShaderFileName, FragmentShaderFileName);

int InfoLogLength = 0;

glGetProgramiv(Program, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetProgramInfoLog(Program, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

Destroy();

return false;

}

return true;

}

void CShaderProgram::Destroy()

{

glDetachShader(Program, VertexShader);

glDetachShader(Program, FragmentShader);

glDeleteShader(VertexShader);

glDeleteShader(FragmentShader);

glDeleteProgram(Program);

delete [] UniformLocations;

delete [] AttribLocations;

SetDefaults();

}

GLuint CShaderProgram::LoadShader(char \*FileName, GLenum Type)

{

CString DirectoryFileName = ModuleDirectory + FileName;

FILE \*File;

if(fopen\_s(&File, DirectoryFileName, "rb") != 0)

{

ErrorLog.Append("Error loading file " + DirectoryFileName + "!\r\n");

return 0;

}

fseek(File, 0, SEEK\_END);

long Size = ftell(File);

fseek(File, 0, SEEK\_SET);

char \*Source = new char[Size + 1];

fread(Source, 1, Size, File);

fclose(File);

Source[Size] = 0;

GLuint Shader = glCreateShader(Type);

glShaderSource(Shader, 1, (const char\*\*)&Source, NULL);

delete [] Source;

glCompileShader(Shader);

int CompileStatus;

glGetShaderiv(Shader, GL\_COMPILE\_STATUS, &CompileStatus);

if(CompileStatus == GL\_FALSE)

{

ErrorLog.Append("Error compiling shader %s!\r\n", FileName);

int InfoLogLength = 0;

glGetShaderiv(Shader, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetShaderInfoLog(Shader, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

glDeleteShader(Shader);

return 0;

}

return Shader;

}

void CShaderProgram::SetDefaults()

{

VertexShader = 0;

FragmentShader = 0;

Program = 0;

UniformLocations = NULL;

AttribLocations = NULL;

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera::CCamera()

{

ViewMatrix = NULL;

ViewMatrixInverse = NULL;

X = vec3(1.0f, 0.0f, 0.0f);

Y = vec3(0.0f, 1.0f, 0.0f);

Z = vec3(0.0f, 0.0f, 1.0f);

Position = vec3(0.0f, 0.0f, 5.0f);

Reference = vec3(0.0f, 0.0f, 0.0f);

}

CCamera::~CCamera()

{

}

void CCamera::Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference)

{

this->Position = Position;

this->Reference = Reference;

Z = normalize(Position - Reference);

X = normalize(cross(vec3(0.0f, 1.0f, 0.0f), Z));

Y = cross(Z, X);

if(!RotateAroundReference)

{

this->Reference = this->Position;

this->Position += Z \* 0.05f;

}

CalculateViewMatrix();

}

void CCamera::Move(const vec3 &Movement)

{

Position += Movement;

Reference += Movement;

CalculateViewMatrix();

}

vec3 CCamera::OnKeys(BYTE Keys, float FrameTime)

{

float Speed = 5.0f;

if(Keys & 0x40) Speed \*= 2.0f;

if(Keys & 0x80) Speed \*= 0.5f;

float Distance = Speed \* FrameTime;

vec3 Up(0.0f, 1.0f, 0.0f);

vec3 Right = X;

vec3 Forward = cross(Up, Right);

Up \*= Distance;

Right \*= Distance;

Forward \*= Distance;

vec3 Movement;

if(Keys & 0x01) Movement += Forward;

if(Keys & 0x02) Movement -= Forward;

if(Keys & 0x04) Movement -= Right;

if(Keys & 0x08) Movement += Right;

if(Keys & 0x10) Movement += Up;

if(Keys & 0x20) Movement -= Up;

return Movement;

}

void CCamera::OnMouseMove(int dx, int dy)

{

float Sensitivity = 0.25f;

Position -= Reference;

if(dx != 0)

{

float DeltaX = (float)dx \* Sensitivity;

X = rotate(X, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Y = rotate(Y, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Z = rotate(Z, DeltaX, vec3(0.0f, 1.0f, 0.0f));

}

if(dy != 0)

{

float DeltaY = (float)dy \* Sensitivity;

Y = rotate(Y, DeltaY, X);

Z = rotate(Z, DeltaY, X);

if(Y.y < 0.0f)

{

Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);

Y = cross(Z, X);

}

}

Position = Reference + Z \* length(Position);

CalculateViewMatrix();

}

void CCamera::OnMouseWheel(float zDelta)

{

Position -= Reference;

if(zDelta < 0 && length(Position) < 500.0f)

{

Position += Position \* 0.1f;

}

if(zDelta > 0 && length(Position) > 0.05f)

{

Position -= Position \* 0.1f;

}

Position += Reference;

CalculateViewMatrix();

}

void CCamera::SetViewMatrixPointer(float \*ViewMatrix, float \*ViewMatrixInverse)

{

this->ViewMatrix = (mat4x4\*)ViewMatrix;

this->ViewMatrixInverse = (mat4x4\*)ViewMatrixInverse;

CalculateViewMatrix();

}

void CCamera::CalculateViewMatrix()

{

if(ViewMatrix != NULL)

{

\*ViewMatrix = mat4x4(X.x, Y.x, Z.x, 0.0f, X.y, Y.y, Z.y, 0.0f, X.z, Y.z, Z.z, 0.0f, -dot(X, Position), -dot(Y, Position), -dot(Z, Position), 1.0f);

if(ViewMatrixInverse != NULL)

{

\*ViewMatrixInverse = inverse(\*ViewMatrix);

}

}

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera Camera;

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer::COpenGLRenderer()

{

Camera.SetViewMatrixPointer(&ViewMatrix);

}

COpenGLRenderer::~COpenGLRenderer()

{

}

bool COpenGLRenderer::Init()

{

bool Error = false;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

ErrorLog.Append("GL\_ARB\_texture\_non\_power\_of\_two not supported!\r\n");

Error = true;

}

if(!GLEW\_ARB\_depth\_texture)

{

ErrorLog.Append("GL\_ARB\_depth\_texture not supported!\r\n");

Error = true;

}

if(!GLEW\_EXT\_framebuffer\_object)

{

ErrorLog.Append("GL\_EXT\_framebuffer\_object not supported!\r\n");

Error = true;

}

Error |= !DirtTexture.LoadTexture2D("lensdirt\_lowc.jpg");

Error |= !SunDepthTest.Load("sundepthtest.vs", "sundepthtest.fs");

Error |= !BlurH.Load("blur.vs", "blurh.fs");

Error |= !BlurV.Load("blur.vs", "blurv.fs");

Error |= !SunRaysLensFlareHalo.Load("sunrayslensflarehalo.vs", "sunrayslensflarehalo.fs");

if(Error)

{

return false;

}

glUseProgram(SunDepthTest);

glUniform1i(glGetUniformLocation(SunDepthTest, "SunTexture"), 0);

glUniform1i(glGetUniformLocation(SunDepthTest, "SunDepthTexture"), 1);

glUniform1i(glGetUniformLocation(SunDepthTest, "DepthTexture"), 2);

glUseProgram(0);

glUseProgram(SunRaysLensFlareHalo);

glUniform1i(glGetUniformLocation(SunRaysLensFlareHalo, "LowBlurredSunTexture"), 0);

glUniform1i(glGetUniformLocation(SunRaysLensFlareHalo, "HighBlurredSunTexture"), 1);

glUniform1i(glGetUniformLocation(SunRaysLensFlareHalo, "DirtTexture"), 2);

glUniform1f(glGetUniformLocation(SunRaysLensFlareHalo, "Dispersal"), 0.1875f);

glUniform1f(glGetUniformLocation(SunRaysLensFlareHalo, "HaloWidth"), 0.45f);

glUniform1f(glGetUniformLocation(SunRaysLensFlareHalo, "Intensity"), 1.5f);

glUniform3f(glGetUniformLocation(SunRaysLensFlareHalo, "Distortion"), 0.94f, 0.97f, 1.00f);

glUseProgram(0);

//glGenTextures(1, &ScreenTexture);

glGenTextures(1, &DepthTexture);

glGenTextures(5, SunTextures);

glGenFramebuffersEXT(1, &FBO);

Camera.Look(vec3(0.5f, 0.25f, 2.5f) \* 2.5f, vec3(0.0f, 0.0f, 0.0f), true);

GLfloat LightModelAmbient[] = {0.0f, 0.0f, 0.0f, 1.0f};

glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, LightModelAmbient);

GLfloat LightAmbient[] = {0.25f, 0.25f, 0.25f, 1.0f};

glLightfv(GL\_LIGHT0, GL\_AMBIENT, LightAmbient);

GLfloat LightDiffuse[] = {0.75f, 0.75f, 0.75f, 1.0f};

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, LightDiffuse);

GLfloat MaterialAmbient[] = {1.0f, 1.0f, 1.0f, 1.0f};

glMaterialfv(GL\_FRONT\_AND\_BACK, GL\_AMBIENT, MaterialAmbient);

GLfloat MaterialDiffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};

glMaterialfv(GL\_FRONT\_AND\_BACK, GL\_DIFFUSE, MaterialDiffuse);

glEnable(GL\_LIGHT0);

glEnable(GL\_COLOR\_MATERIAL);

return true;

}

void COpenGLRenderer::Render(float FrameTime)

{

float SunR = 3.75f;

vec3 SunPos = /\*Camera.Position +\*/ vec3(0.0f, 0.0f, -100.0f);

glViewport(0, 0, Width, Height);

//glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

//glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, ScreenTexture, 0);

//glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, DepthTexture, 0);

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

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&ProjectionMatrix);

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glLightfv(GL\_LIGHT0, GL\_POSITION, &vec4(SunPos, 1.0f));

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

glEnable(GL\_LIGHTING);

glColor3f(1.0f, 1.0f, 1.0f);

GLUquadric \*obj = gluNewQuadric();

for(int z = -2; z <= 2; z += 1)

{

for(int x = -2; x <= 2; x += 1)

{

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glTranslatef((float)x, -0.5f, (float)z);

gluSphere(obj, 0.25f, 32, 32);

}

}

gluDeleteQuadric(obj);

glDisable(GL\_LIGHTING);

glDisable(GL\_CULL\_FACE);

glDisable(GL\_DEPTH\_TEST);

glBindTexture(GL\_TEXTURE\_2D, DepthTexture);

glCopyTexSubImage2D(GL\_TEXTURE\_2D, 0, 0, 0, 0, 0, Width, Height);

glBindTexture(GL\_TEXTURE\_2D, 0);

//glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

bool CalculateSunRaysLensFlareHalo = false;

int Test = 0, Tests = 16;

float Angle = 0.0f, AngleInc = 360.0f / Tests;

mat4x4 VPB = BiasMatrix \* ProjectionMatrix \* ViewMatrix;

while(Test < Tests && !CalculateSunRaysLensFlareHalo)

{

vec4 SunPosProj = VPB \* vec4(SunPos + rotate(Camera.X, Angle, Camera.Z) \* SunR, 1.0f);

SunPosProj /= SunPosProj.w;

CalculateSunRaysLensFlareHalo |= (SunPosProj.x >= 0.0f && SunPosProj.x <= 1.0f && SunPosProj.y >= 0.0f && SunPosProj.y <= 1.0f && SunPosProj.z >= 0.0f && SunPosProj.z <= 1.0f);

Angle += AngleInc;

Test++;

}

if(CalculateSunRaysLensFlareHalo)

{

vec4 SunPosProj = VPB \* vec4(SunPos, 1.0f);

SunPosProj /= SunPosProj.w;

glViewport(0, 0, SunTextureWidth, SunTextureHeight);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[1], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, SunTextures[4], 0);

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

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glTranslatef(SunPos.x, SunPos.y, SunPos.z);

glColor3f(1.0f, 0.90f, 0.80f);

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

GLUquadric \*obj = gluNewQuadric();

gluSphere(obj, SunR, 16, 16);

gluDeleteQuadric(obj);

glDisable(GL\_CULL\_FACE);

glDisable(GL\_DEPTH\_TEST);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[0], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, SunTextures[1]);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, SunTextures[4]);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_2D, DepthTexture);

glUseProgram(SunDepthTest);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[2], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[0]);

glUseProgram(BlurH);

glUniform1i(glGetUniformLocation(BlurH, "Width"), 1);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[1], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[2]);

glUseProgram(BlurV);

glUniform1i(glGetUniformLocation(BlurV, "Width"), 1);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[3], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[0]);

glUseProgram(BlurH);

glUniform1i(glGetUniformLocation(BlurH, "Width"), 10);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[2], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[3]);

glUseProgram(BlurV);

glUniform1i(glGetUniformLocation(BlurV, "Width"), 10);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[3], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, SunTextures[1]);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, SunTextures[2]);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_2D, DirtTexture);

glUseProgram(SunRaysLensFlareHalo);

glUniform2fv(glGetUniformLocation(SunRaysLensFlareHalo, "SunPosProj"), 1, &SunPosProj);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

glViewport(0, 0, Width, Height);

}

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&ortho(0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f));

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

/\*glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, ScreenTexture);

glColor3f(1.0f, 1.0f, 1.0f);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f); glVertex2f(0.0f, 0.0f);

glTexCoord2f(1.0f, 0.0f); glVertex2f(1.0f, 0.0f);

glTexCoord2f(1.0f, 1.0f); glVertex2f(1.0f, 1.0f);

glTexCoord2f(0.0f, 1.0f); glVertex2f(0.0f, 1.0f);

glEnd();

glBindTexture(GL\_TEXTURE\_2D, 0);

glDisable(GL\_TEXTURE\_2D);\*/

if(CalculateSunRaysLensFlareHalo)

{

glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[3]);

glBlendFunc(GL\_ONE, GL\_ONE\_MINUS\_SRC\_COLOR);

glEnable(GL\_BLEND);

glColor3f(1.0f, 1.0f, 1.0f);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f); glVertex2f(0.0f, 0.0f);

glTexCoord2f(1.0f, 0.0f); glVertex2f(1.0f, 0.0f);

glTexCoord2f(1.0f, 1.0f); glVertex2f(1.0f, 1.0f);

glTexCoord2f(0.0f, 1.0f); glVertex2f(0.0f, 1.0f);

glEnd();

glDisable(GL\_BLEND);

glBindTexture(GL\_TEXTURE\_2D, 0);

glDisable(GL\_TEXTURE\_2D);

}

}

void COpenGLRenderer::Resize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

ProjectionMatrix = perspective(45.0f, (float)Width / (float)Height, 0.125f, 512.0f);

/\*glBindTexture(GL\_TEXTURE\_2D, ScreenTexture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);\*/

glBindTexture(GL\_TEXTURE\_2D, DepthTexture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT24, Width, Height, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);

SunTextureWidth = Width / 2;

SunTextureHeight = Height / 2;

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

{

glBindTexture(GL\_TEXTURE\_2D, SunTextures[i]);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, SunTextureWidth, SunTextureHeight, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);

}

glBindTexture(GL\_TEXTURE\_2D, SunTextures[4]);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT24, SunTextureWidth, SunTextureHeight, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);

glUseProgram(BlurH);

glUniform1f(glGetUniformLocation(BlurH, "odw"), 1.0f / (float)SunTextureWidth);

glUseProgram(BlurV);

glUniform1f(glGetUniformLocation(BlurV, "odh"), 1.0f / (float)SunTextureHeight);

glUseProgram(0);

}

void COpenGLRenderer::Destroy()

{

DirtTexture.Destroy();

//glDeleteTextures(1, &ScreenTexture);

glDeleteTextures(1, &DepthTexture);

glDeleteTextures(5, SunTextures);

SunDepthTest.Destroy();

BlurH.Destroy();

BlurV.Destroy();

SunRaysLensFlareHalo.Destroy();

if(GLEW\_EXT\_framebuffer\_object)

{

glDeleteFramebuffersEXT(1, &FBO);

}

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer OpenGLRenderer;

// ----------------------------------------------------------------------------------------------------------------------------

CString ModuleDirectory, ErrorLog;

// ----------------------------------------------------------------------------------------------------------------------------

void GetModuleDirectory()

{

char \*moduledirectory = new char[256];

GetModuleFileName(GetModuleHandle(NULL), moduledirectory, 256);

\*(strrchr(moduledirectory, '\\') + 1) = 0;

ModuleDirectory = moduledirectory;

delete [] moduledirectory;

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView::COpenGLView()

{

}

COpenGLView::~COpenGLView()

{

}

bool COpenGLView::Init(HINSTANCE hInstance, char \*Title, int Width, int Height, int Samples)

{

this->Title = Title;

this->Width = Width;

this->Height = Height;

WNDCLASSEX WndClassEx;

memset(&WndClassEx, 0, sizeof(WNDCLASSEX));

WndClassEx.cbSize = sizeof(WNDCLASSEX);

WndClassEx.style = CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

WndClassEx.lpfnWndProc = WndProc;

WndClassEx.hInstance = hInstance;

WndClassEx.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hCursor = LoadCursor(NULL, IDC\_ARROW);

WndClassEx.lpszClassName = "Win32OpenGLWindowClass";

if(RegisterClassEx(&WndClassEx) == 0)

{

ErrorLog.Set("RegisterClassEx failed!");

return false;

}

DWORD Style = WS\_OVERLAPPEDWINDOW | WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN;

hWnd = CreateWindowEx(WS\_EX\_APPWINDOW, WndClassEx.lpszClassName, Title, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL);

if(hWnd == NULL)

{

ErrorLog.Set("CreateWindowEx failed!");

return false;

}

HDC hDC = GetDC(hWnd);

if(hDC == NULL)

{

ErrorLog.Set("GetDC failed!");

return false;

}

PIXELFORMATDESCRIPTOR pfd;

memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));

pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);

pfd.nVersion = 1;

pfd.dwFlags = PFD\_DRAW\_TO\_WINDOW | PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER;

pfd.iPixelType = PFD\_TYPE\_RGBA;

pfd.cColorBits = 32;

pfd.cDepthBits = 24;

pfd.iLayerType = PFD\_MAIN\_PLANE;

int PixelFormat = ChoosePixelFormat(hDC, &pfd);

if(PixelFormat == 0)

{

ErrorLog.Set("ChoosePixelFormat failed!");

return false;

}

static int MSAAPixelFormat = 0;

if(SetPixelFormat(hDC, MSAAPixelFormat == 0 ? PixelFormat : MSAAPixelFormat, &pfd) == FALSE)

{

ErrorLog.Set("SetPixelFormat failed!");

return false;

}

hGLRC = wglCreateContext(hDC);

if(hGLRC == NULL)

{

ErrorLog.Set("wglCreateContext failed!");

return false;

}

if(wglMakeCurrent(hDC, hGLRC) == FALSE)

{

ErrorLog.Set("wglMakeCurrent failed!");

return false;

}

if(glewInit() != GLEW\_OK)

{

ErrorLog.Set("glewInit failed!");

return false;

}

if(!GLEW\_VERSION\_2\_1)

{

ErrorLog.Set("OpenGL 2.1 not supported!");

return false;

}

if(MSAAPixelFormat == 0 && Samples > 0)

{

if(GLEW\_ARB\_multisample && WGLEW\_ARB\_pixel\_format)

{

while(Samples > 0)

{

UINT NumFormats = 0;

int PFAttribs[] =

{

WGL\_DRAW\_TO\_WINDOW\_ARB, GL\_TRUE,

WGL\_SUPPORT\_OPENGL\_ARB, GL\_TRUE,

WGL\_DOUBLE\_BUFFER\_ARB, GL\_TRUE,

WGL\_PIXEL\_TYPE\_ARB, WGL\_TYPE\_RGBA\_ARB,

WGL\_COLOR\_BITS\_ARB, 32,

WGL\_DEPTH\_BITS\_ARB, 24,

WGL\_ACCELERATION\_ARB, WGL\_FULL\_ACCELERATION\_ARB,

WGL\_SAMPLE\_BUFFERS\_ARB, GL\_TRUE,

WGL\_SAMPLES\_ARB, Samples,

0

};

if(wglChoosePixelFormatARB(hDC, PFAttribs, NULL, 1, &MSAAPixelFormat, &NumFormats) == TRUE && NumFormats > 0) break;

Samples--;

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

UnregisterClass(WndClassEx.lpszClassName, hInstance);

return Init(hInstance, Title, Width, Height, Samples);

}

else

{

Samples = 0;

}

}

this->Samples = Samples;

GetModuleDirectory();

glGetIntegerv(GL\_MAX\_TEXTURE\_SIZE, &gl\_max\_texture\_size);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glGetIntegerv(GL\_MAX\_TEXTURE\_MAX\_ANISOTROPY\_EXT, &gl\_max\_texture\_max\_anisotropy\_ext);

}

if(WGLEW\_EXT\_swap\_control)

{

wglSwapIntervalEXT(0);

}

return OpenGLRenderer.Init();

}

void COpenGLView::Show(bool Maximized)

{

RECT dRect, wRect, cRect;

GetWindowRect(GetDesktopWindow(), &dRect);

GetWindowRect(hWnd, &wRect);

GetClientRect(hWnd, &cRect);

wRect.right += Width - cRect.right;

wRect.bottom += Height - cRect.bottom;

wRect.right -= wRect.left;

wRect.bottom -= wRect.top;

wRect.left = dRect.right / 2 - wRect.right / 2;

wRect.top = dRect.bottom / 2 - wRect.bottom / 2;

MoveWindow(hWnd, wRect.left, wRect.top, wRect.right, wRect.bottom, FALSE);

ShowWindow(hWnd, Maximized ? SW\_SHOWMAXIMIZED : SW\_SHOWNORMAL);

}

void COpenGLView::MessageLoop()

{

MSG Msg;

while(GetMessage(&Msg, NULL, 0, 0) > 0)

{

TranslateMessage(&Msg);

DispatchMessage(&Msg);

}

}

void COpenGLView::Destroy()

{

if(GLEW\_VERSION\_2\_1)

{

OpenGLRenderer.Destroy();

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

}

void COpenGLView::OnKeyDown(UINT Key)

{

/\*switch(Key)

{

case VK\_F1:

break;

case VK\_SPACE:

break;

}\*/

}

void COpenGLView::OnMouseMove(int X, int Y)

{

if(GetKeyState(VK\_RBUTTON) & 0x80)

{

Camera.OnMouseMove(LastX - X, LastY - Y);

LastX = X;

LastY = Y;

}

}

void COpenGLView::OnMouseWheel(short zDelta)

{

Camera.OnMouseWheel(zDelta);

}

void COpenGLView::OnPaint()

{

static DWORD LastFPSTime = GetTickCount(), LastFrameTime = LastFPSTime, FPS = 0;

PAINTSTRUCT ps;

HDC hDC = BeginPaint(hWnd, &ps);

DWORD Time = GetTickCount();

float FrameTime = (Time - LastFrameTime) \* 0.001f;

LastFrameTime = Time;

if(Time - LastFPSTime > 1000)

{

CString Text = Title;

if(OpenGLRenderer.Text[0] != 0)

{

Text.Append(" - " + OpenGLRenderer.Text);

}

Text.Append(" - %dx%d", Width, Height);

Text.Append(", ATF %dx", gl\_max\_texture\_max\_anisotropy\_ext);

Text.Append(", MSAA %dx", Samples);

Text.Append(", FPS: %d", FPS);

Text.Append(" - %s", glGetString(GL\_RENDERER));

SetWindowText(hWnd, Text);

LastFPSTime = Time;

FPS = 0;

}

else

{

FPS++;

}

BYTE Keys = 0x00;

if(GetKeyState('W') & 0x80) Keys |= 0x01;

if(GetKeyState('S') & 0x80) Keys |= 0x02;

if(GetKeyState('A') & 0x80) Keys |= 0x04;

if(GetKeyState('D') & 0x80) Keys |= 0x08;

if(GetKeyState('R') & 0x80) Keys |= 0x10;

if(GetKeyState('F') & 0x80) Keys |= 0x20;

if(GetKeyState(VK\_SHIFT) & 0x80) Keys |= 0x40;

if(GetKeyState(VK\_CONTROL) & 0x80) Keys |= 0x80;

if(Keys & 0x3F)

{

Camera.Move(Camera.OnKeys(Keys, FrameTime));

}

OpenGLRenderer.Render(FrameTime);

SwapBuffers(hDC);

EndPaint(hWnd, &ps);

InvalidateRect(hWnd, NULL, FALSE);

}

void COpenGLView::OnRButtonDown(int X, int Y)

{

LastX = X;

LastY = Y;

}

void COpenGLView::OnSize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

OpenGLRenderer.Resize(Width, Height);

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView OpenGLView;

// ----------------------------------------------------------------------------------------------------------------------------

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)

{

switch(uiMsg)

{

case WM\_CLOSE:

PostQuitMessage(0);

break;

case WM\_MOUSEMOVE:

OpenGLView.OnMouseMove(LOWORD(lParam), HIWORD(lParam));

break;

case 0x020A: // WM\_MOUSWHEEL

OpenGLView.OnMouseWheel(HIWORD(wParam));

break;

case WM\_KEYDOWN:

OpenGLView.OnKeyDown((UINT)wParam);

break;

case WM\_PAINT:

OpenGLView.OnPaint();

break;

case WM\_RBUTTONDOWN:

OpenGLView.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_SIZE:

OpenGLView.OnSize(LOWORD(lParam), HIWORD(lParam));

break;

default:

return DefWindowProc(hWnd, uiMsg, wParam, lParam);

}

return 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine, int iShow)

{

char \*AppName = "Sun rays, lens flare, halo";

if(OpenGLView.Init(hInstance, AppName, 800, 600, 4))

{

OpenGLView.Show();

OpenGLView.MessageLoop();

}

else

{

MessageBox(NULL, ErrorLog, AppName, MB\_OK | MB\_ICONERROR);

}

OpenGLView.Destroy();

return 0;

}

/////////////插图//////////////////////



/////////////////////////////////////大气光散射//////////////////

(1): blur.vs

#version 120

void main(void)

{

gl\_TexCoord[0] = gl\_Vertex;

gl\_Position = gl\_Vertex \* 2.0 - 1.0;

}

(2: blurh.fs

#version 120

uniform sampler2D Texture;

uniform int Width;

uniform float odw;

void main()

{

vec3 Color = vec3(0.0);

int wp1 = Width + 1;

float Sum = 0.0;

for(int x = -Width; x <= Width; x++)

{

float width = (wp1 - abs(float(x)));

Color += texture2D(Texture, gl\_TexCoord[0].st + vec2(odw \* x, 0.0)).rgb \* width;

Sum += width;

}

gl\_FragColor = vec4(Color / Sum, 1.0);

}

(3): blurv.fs

#version 120

uniform sampler2D Texture;

uniform int Width;

uniform float odh;

void main()

{

vec3 Color = vec3(0.0);

int wp1 = Width + 1;

float Sum = 0.0;

for(int y = -Width; y <= Width; y++)

{

float width = (wp1 - abs(float(y)));

Color += texture2D(Texture, gl\_TexCoord[0].st + vec2(0.0, odh \* y)).rgb \* width;

Sum += width;

}

gl\_FragColor = vec4(Color / Sum, 1.0);

}

(4): skyfromatmosphere.vs

// Atmospheric scattering vertex shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform vec3 v3CameraPos; // The camera's current position

uniform vec3 v3LightPos; // The direction vector to the light source

uniform vec3 v3InvWavelength; // 1 / pow(wavelength, 4) for the red, green, and blue channels

uniform float fCameraHeight; // The camera's current height

uniform float fCameraHeight2; // fCameraHeight^2

uniform float fOuterRadius; // The outer (atmosphere) radius

uniform float fOuterRadius2; // fOuterRadius^2

uniform float fInnerRadius; // The inner (planetary) radius

uniform float fInnerRadius2; // fInnerRadius^2

uniform float fKrESun; // Kr \* ESun

uniform float fKmESun; // Km \* ESun

uniform float fKr4PI; // Kr \* 4 \* PI

uniform float fKm4PI; // Km \* 4 \* PI

uniform float fScale; // 1 / (fOuterRadius - fInnerRadius)

uniform float fScaleDepth; // The scale depth (i.e. the altitude at which the atmosphere's average density is found)

uniform float fScaleOverScaleDepth; // fScale / fScaleDepth

uniform int Samples;

varying vec3 v3Direction;

float scale(float fCos)

{

float x = 1.0 - fCos;

return fScaleDepth \* exp(-0.00287 + x \* (0.459 + x \* (3.83 + x \* (-6.80 + x \* 5.25))));

}

void main(void)

{

// Get the ray from the camera to the vertex, and its length (which is the far point of the ray passing through the atmosphere)

vec3 v3Pos = gl\_Vertex.xyz;

vec3 v3Ray = v3Pos - v3CameraPos;

float fFar = length(v3Ray);

v3Ray /= fFar;

// Calculate the ray's starting position, then calculate its scattering offset

vec3 v3Start = v3CameraPos;

float fHeight = length(v3Start);

float fDepth = exp(fScaleOverScaleDepth \* (fInnerRadius - fCameraHeight));

float fStartAngle = dot(v3Ray, v3Start) / fHeight;

float fStartOffset = fDepth \* scale(fStartAngle);

// Initialize the scattering loop variables

float fSampleLength = fFar / Samples;

float fScaledLength = fSampleLength \* fScale;

vec3 v3SampleRay = v3Ray \* fSampleLength;

vec3 v3SamplePoint = v3Start + v3SampleRay \* 0.5;

// Now loop through the sample rays

vec3 v3FrontColor = vec3(0.0);

for(int i = 0; i < Samples; i++)

{

float fHeight = length(v3SamplePoint);

float fDepth = exp(fScaleOverScaleDepth \* (fInnerRadius - fHeight));

float fLightAngle = dot(v3LightPos, v3SamplePoint) / fHeight;

float fCameraAngle = dot(v3Ray, v3SamplePoint) / fHeight;

float fScatter = (fStartOffset + fDepth \* (scale(fLightAngle) - scale(fCameraAngle)));

vec3 v3Attenuate = exp(-fScatter \* (v3InvWavelength \* fKr4PI + fKm4PI));

v3FrontColor += v3Attenuate \* (fDepth \* fScaledLength);

v3SamplePoint += v3SampleRay;

}

// Finally, scale the Mie and Rayleigh colors and set up the varying variables for the pixel shader

gl\_FrontSecondaryColor.rgb = v3FrontColor \* fKmESun;

gl\_FrontColor.rgb = v3FrontColor \* (v3InvWavelength \* fKrESun);

gl\_Position = gl\_ModelViewProjectionMatrix \* gl\_Vertex;

v3Direction = v3CameraPos - v3Pos;

}

(5): skyfromatmosphere.fs

// Atmospheric scattering fragment shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform vec3 v3LightPos;

uniform float g;

uniform float g2;

varying vec3 v3Direction;

void main (void)

{

float fCos = dot(v3LightPos, v3Direction) / length(v3Direction);

float fRayleighPhase = 1.0 + fCos \* fCos;

float fMiePhase = (1.0 - g2) / (2.0 + g2) \* (1.0 + fCos \* fCos) / pow(1.0 + g2 - 2.0 \* g \* fCos, 1.5);

gl\_FragColor = vec4(1.0 - exp(-1.5 \* (fRayleighPhase \* gl\_Color.rgb + fMiePhase \* gl\_SecondaryColor.rgb)), 1.0);

}

(6): sundepthtest.vs

#version 120

void main()

{

gl\_TexCoord[0] = gl\_Vertex;

gl\_Position = gl\_Vertex \* 2.0 - 1.0;

}

(7): sundepthtest.fs

#version 120

uniform sampler2D SunTexture, DepthTexture;

void main()

{

if(texture2D(DepthTexture, gl\_TexCoord[0].st).r < 1.0)

{

gl\_FragColor = vec4(vec3(0.0), 1.0);

}

else

{

gl\_FragColor = vec4(texture2D(SunTexture, gl\_TexCoord[0].st).rgb, 1.0);

}

}

(8): sunrayslensflarehalo.vs

#version 120

void main(void)

{

gl\_TexCoord[0] = gl\_Vertex;

gl\_Position = gl\_Vertex \* 2.0 - 1.0;

}

（9）：sunrayslensflarehalo.fs

#version 120

uniform sampler2D LowBlurredSunTexture, HighBlurredSunTexture, DirtTexture;

uniform float Dispersal, HaloWidth, Intensity;

uniform vec2 SunPosProj;

uniform vec3 Distortion;

vec3 texture2DDistorted(sampler2D Texture, vec2 TexCoord, vec2 Offset)

{

return vec3(

texture2D(Texture, TexCoord + Offset \* Distortion.r).r,

texture2D(Texture, TexCoord + Offset \* Distortion.g).g,

texture2D(Texture, TexCoord + Offset \* Distortion.b).b

);

}

void main()

{

vec3 RadialBlur = vec3(0.0);

vec2 TexCoord = gl\_TexCoord[0].st;

int RadialBlurSamples = 128;

vec2 RadialBlurVector = (SunPosProj - TexCoord) / RadialBlurSamples;

for(int i = 0; i < RadialBlurSamples; i++)

{

RadialBlur += texture2D(LowBlurredSunTexture, TexCoord).rgb;

TexCoord += RadialBlurVector;

}

RadialBlur /= RadialBlurSamples;

vec3 LensFlareHalo = vec3(0.0);

TexCoord = 1.0 - gl\_TexCoord[0].st;

vec2 LensFlareVector = (vec2(0.5) - TexCoord) \* Dispersal;

vec2 LensFlareOffset = vec2(0.0);

for(int i = 0; i < 5; i++)

{

LensFlareHalo += texture2DDistorted(HighBlurredSunTexture, TexCoord, LensFlareOffset).rgb;

LensFlareOffset += LensFlareVector;

}

LensFlareHalo += texture2DDistorted(HighBlurredSunTexture, TexCoord, normalize(LensFlareVector) \* HaloWidth);

LensFlareHalo /= 6.0;

gl\_FragColor = vec4((texture2D(HighBlurredSunTexture, gl\_TexCoord[0].st).rgb + (RadialBlur + LensFlareHalo) \* texture2D(DirtTexture, gl\_TexCoord[0].st).rgb) \* Intensity, 1.0);

}

//////////////////////////////////////////Source//////////////////////////////////////

class COpenGLRenderer

{

protected:

int Width, Height;

mat3x3 NormalMatrix;

mat4x4 ModelMatrix, ViewMatrix, ProjectionMatrix;

protected:

CTexture Texture, DirtTexture;

CShaderProgram Sky, SunDepthTest, BlurH, BlurV, SunRaysLensFlareHalo;

GLuint /\*ScreenTexture,\*/ DepthTexture, SunTextures[4], FBO, VBO;

int SunTextureWidth, SunTextureHeight;

public:

bool Pause;

float SunR;

vec3 SunCPos, SunRotVec;

public:

CString Text;

public:

COpenGLRenderer();

~COpenGLRenderer();

bool Init();

void Render(float FrameTime);

void Resize(int Width, int Height);

void Destroy();

};

#include "opengl\_21\_tutorials\_win32\_framework.h"

// ----------------------------------------------------------------------------------------------------------------------------

CBuffer::CBuffer()

{

SetDefaults();

}

CBuffer::~CBuffer()

{

Empty();

}

void CBuffer::AddData(void \*Data, int DataSize)

{

int Remaining = BufferSize - Position;

if(DataSize > Remaining)

{

BYTE \*OldBuffer = Buffer;

int OldBufferSize = BufferSize;

int Needed = DataSize - Remaining;

BufferSize += Needed > BUFFER\_SIZE\_INCREMENT ? Needed : BUFFER\_SIZE\_INCREMENT;

Buffer = new BYTE[BufferSize];

memcpy(Buffer, OldBuffer, OldBufferSize);

delete [] OldBuffer;

}

memcpy(Buffer + Position, Data, DataSize);

Position += DataSize;

}

void CBuffer::Empty()

{

delete [] Buffer;

SetDefaults();

}

void \*CBuffer::GetData()

{

return Buffer;

}

int CBuffer::GetDataSize()

{

return Position;

}

void CBuffer::SetDefaults()

{

Buffer = NULL;

BufferSize = 0;

Position = 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int gl\_max\_texture\_size = 0, gl\_max\_texture\_max\_anisotropy\_ext = 0;

// ----------------------------------------------------------------------------------------------------------------------------

CTexture::CTexture()

{

Texture = 0;

}

CTexture::~CTexture()

{

}

CTexture::operator GLuint ()

{

return Texture;

}

bool CTexture::LoadTexture2D(char \*FileName)

{

CString DirectoryFileName = ModuleDirectory + FileName;

int Width, Height, BPP;

FIBITMAP \*dib = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileName);

FreeImage\_Unload(dib);

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_2D, Texture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_2D, GL\_GENERATE\_MIPMAP, GL\_TRUE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib));

glBindTexture(GL\_TEXTURE\_2D, 0);

FreeImage\_Unload(dib);

return true;

}

bool CTexture::LoadTextureCubeMap(char \*\*FileNames)

{

int Width, Height, BPP;

FIBITMAP \*dib[6];

bool Error = false;

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

{

CString DirectoryFileName = ModuleDirectory + FileNames[i];

dib[i] = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib[i] == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

Error = true;

}

}

if(Error)

{

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileNames[5]);

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, Texture);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_GENERATE\_MIPMAP, GL\_TRUE);

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

{

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib[i]));

}

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

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

{

FreeImage\_Unload(dib[i]);

}

return true;

}

void CTexture::Destroy()

{

glDeleteTextures(1, &Texture);

Texture = 0;

}

FIBITMAP \*CTexture::GetBitmap(char \*FileName, int &Width, int &Height, int &BPP)

{

FREE\_IMAGE\_FORMAT fif = FreeImage\_GetFileType(FileName);

if(fif == FIF\_UNKNOWN)

{

fif = FreeImage\_GetFIFFromFilename(FileName);

}

if(fif == FIF\_UNKNOWN)

{

return NULL;

}

FIBITMAP \*dib = NULL;

if(FreeImage\_FIFSupportsReading(fif))

{

dib = FreeImage\_Load(fif, FileName);

}

if(dib != NULL)

{

int OriginalWidth = FreeImage\_GetWidth(dib);

int OriginalHeight = FreeImage\_GetHeight(dib);

Width = OriginalWidth;

Height = OriginalHeight;

if(Width == 0 || Height == 0)

{

FreeImage\_Unload(dib);

return NULL;

}

BPP = FreeImage\_GetBPP(dib);

if(Width > gl\_max\_texture\_size) Width = gl\_max\_texture\_size;

if(Height > gl\_max\_texture\_size) Height = gl\_max\_texture\_size;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

Width = 1 << (int)floor((log((float)Width) / log(2.0f)) + 0.5f);

Height = 1 << (int)floor((log((float)Height) / log(2.0f)) + 0.5f);

}

if(Width != OriginalWidth || Height != OriginalHeight)

{

FIBITMAP \*rdib = FreeImage\_Rescale(dib, Width, Height, FILTER\_BICUBIC);

FreeImage\_Unload(dib);

dib = rdib;

}

}

return dib;

}

// ----------------------------------------------------------------------------------------------------------------------------

CShaderProgram::CShaderProgram()

{

SetDefaults();

}

CShaderProgram::~CShaderProgram()

{

}

CShaderProgram::operator GLuint ()

{

return Program;

}

bool CShaderProgram::Load(char \*VertexShaderFileName, char \*FragmentShaderFileName)

{

bool Error = false;

Destroy();

Error |= ((VertexShader = LoadShader(VertexShaderFileName, GL\_VERTEX\_SHADER)) == 0);

Error |= ((FragmentShader = LoadShader(FragmentShaderFileName, GL\_FRAGMENT\_SHADER)) == 0);

if(Error)

{

Destroy();

return false;

}

Program = glCreateProgram();

glAttachShader(Program, VertexShader);

glAttachShader(Program, FragmentShader);

glLinkProgram(Program);

int LinkStatus;

glGetProgramiv(Program, GL\_LINK\_STATUS, &LinkStatus);

if(LinkStatus == GL\_FALSE)

{

ErrorLog.Append("Error linking program (%s, %s)!\r\n", VertexShaderFileName, FragmentShaderFileName);

int InfoLogLength = 0;

glGetProgramiv(Program, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetProgramInfoLog(Program, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

Destroy();

return false;

}

return true;

}

void CShaderProgram::Destroy()

{

glDetachShader(Program, VertexShader);

glDetachShader(Program, FragmentShader);

glDeleteShader(VertexShader);

glDeleteShader(FragmentShader);

glDeleteProgram(Program);

delete [] UniformLocations;

delete [] AttribLocations;

SetDefaults();

}

GLuint CShaderProgram::LoadShader(char \*FileName, GLenum Type)

{

CString DirectoryFileName = ModuleDirectory + FileName;

FILE \*File;

if(fopen\_s(&File, DirectoryFileName, "rb") != 0)

{

ErrorLog.Append("Error loading file " + DirectoryFileName + "!\r\n");

return 0;

}

fseek(File, 0, SEEK\_END);

long Size = ftell(File);

fseek(File, 0, SEEK\_SET);

char \*Source = new char[Size + 1];

fread(Source, 1, Size, File);

fclose(File);

Source[Size] = 0;

GLuint Shader = glCreateShader(Type);

glShaderSource(Shader, 1, (const char\*\*)&Source, NULL);

delete [] Source;

glCompileShader(Shader);

int CompileStatus;

glGetShaderiv(Shader, GL\_COMPILE\_STATUS, &CompileStatus);

if(CompileStatus == GL\_FALSE)

{

ErrorLog.Append("Error compiling shader %s!\r\n", FileName);

int InfoLogLength = 0;

glGetShaderiv(Shader, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetShaderInfoLog(Shader, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

glDeleteShader(Shader);

return 0;

}

return Shader;

}

void CShaderProgram::SetDefaults()

{

VertexShader = 0;

FragmentShader = 0;

Program = 0;

UniformLocations = NULL;

AttribLocations = NULL;

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera::CCamera()

{

ViewMatrix = NULL;

ViewMatrixInverse = NULL;

X = vec3(1.0f, 0.0f, 0.0f);

Y = vec3(0.0f, 1.0f, 0.0f);

Z = vec3(0.0f, 0.0f, 1.0f);

Position = vec3(0.0f, 0.0f, 5.0f);

Reference = vec3(0.0f, 0.0f, 0.0f);

}

CCamera::~CCamera()

{

}

void CCamera::Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference)

{

this->Position = Position;

this->Reference = Reference;

Z = normalize(Position - Reference);

X = normalize(cross(vec3(0.0f, 1.0f, 0.0f), Z));

Y = cross(Z, X);

if(!RotateAroundReference)

{

this->Reference = this->Position;

this->Position += Z \* 0.05f;

}

CalculateViewMatrix();

}

void CCamera::Move(const vec3 &Movement)

{

Position += Movement;

Reference += Movement;

CalculateViewMatrix();

}

vec3 CCamera::OnKeys(BYTE Keys, float FrameTime)

{

float Speed = 5.0f;

if(Keys & 0x40) Speed \*= 2.0f;

if(Keys & 0x80) Speed \*= 0.5f;

float Distance = Speed \* FrameTime;

vec3 Up(0.0f, 1.0f, 0.0f);

vec3 Right = X;

vec3 Forward = cross(Up, Right);

Up \*= Distance;

Right \*= Distance;

Forward \*= Distance;

vec3 Movement;

if(Keys & 0x01) Movement += Forward;

if(Keys & 0x02) Movement -= Forward;

if(Keys & 0x04) Movement -= Right;

if(Keys & 0x08) Movement += Right;

if(Keys & 0x10) Movement += Up;

if(Keys & 0x20) Movement -= Up;

return Movement;

}

void CCamera::OnMouseMove(int dx, int dy)

{

float Sensitivity = 0.25f;

Position -= Reference;

if(dx != 0)

{

float DeltaX = (float)dx \* Sensitivity;

X = rotate(X, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Y = rotate(Y, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Z = rotate(Z, DeltaX, vec3(0.0f, 1.0f, 0.0f));

}

if(dy != 0)

{

float DeltaY = (float)dy \* Sensitivity;

Y = rotate(Y, DeltaY, X);

Z = rotate(Z, DeltaY, X);

if(Y.y < 0.0f)

{

Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);

Y = cross(Z, X);

}

}

Position = Reference + Z \* length(Position);

CalculateViewMatrix();

}

void CCamera::OnMouseWheel(float zDelta)

{

Position -= Reference;

if(zDelta < 0 && length(Position) < 500.0f)

{

Position += Position \* 0.1f;

}

if(zDelta > 0 && length(Position) > 0.05f)

{

Position -= Position \* 0.1f;

}

Position += Reference;

CalculateViewMatrix();

}

void CCamera::SetViewMatrixPointer(float \*ViewMatrix, float \*ViewMatrixInverse)

{

this->ViewMatrix = (mat4x4\*)ViewMatrix;

this->ViewMatrixInverse = (mat4x4\*)ViewMatrixInverse;

CalculateViewMatrix();

}

void CCamera::CalculateViewMatrix()

{

if(ViewMatrix != NULL)

{

\*ViewMatrix = mat4x4(X.x, Y.x, Z.x, 0.0f, X.y, Y.y, Z.y, 0.0f, X.z, Y.z, Z.z, 0.0f, -dot(X, Position), -dot(Y, Position), -dot(Z, Position), 1.0f);

if(ViewMatrixInverse != NULL)

{

\*ViewMatrixInverse = inverse(\*ViewMatrix);

}

}

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera Camera;

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer::COpenGLRenderer()

{

Pause = false;

SunR = 15.0f;

SunRotVec = normalize(vec3(0.0f, -0.66f, -1.0f));

SunCPos = rotate(vec3(-512.0f, 0.0f, 0.0f), -22.5f, SunRotVec);

Camera.SetViewMatrixPointer(&ViewMatrix);

}

COpenGLRenderer::~COpenGLRenderer()

{

}

bool COpenGLRenderer::Init()

{

bool Error = false;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

ErrorLog.Append("GL\_ARB\_texture\_non\_power\_of\_two not supported!\r\n");

Error = true;

}

if(!GLEW\_ARB\_depth\_texture)

{

ErrorLog.Append("GL\_ARB\_depth\_texture not supported!\r\n");

Error = true;

}

if(!GLEW\_EXT\_framebuffer\_object)

{

ErrorLog.Append("GL\_EXT\_framebuffer\_object not supported!\r\n");

Error = true;

}

Error |= !Texture.LoadTexture2D("texture.jpg");

Error |= !DirtTexture.LoadTexture2D("lensdirt\_lowc.jpg");

Error |= !Sky.Load("skyfromatmosphere.vs", "skyfromatmosphere.fs");

Error |= !SunDepthTest.Load("sundepthtest.vs", "sundepthtest.fs");

Error |= !BlurH.Load("blur.vs", "blurh.fs");

Error |= !BlurV.Load("blur.vs", "blurv.fs");

Error |= !SunRaysLensFlareHalo.Load("sunrayslensflarehalo.vs", "sunrayslensflarehalo.fs");

if(Error)

{

return false;

}

glUseProgram(SunDepthTest);

glUniform1i(glGetUniformLocation(SunDepthTest, "SunTexture"), 0);

glUniform1i(glGetUniformLocation(SunDepthTest, "DepthTexture"), 1);

glUseProgram(0);

glUseProgram(SunRaysLensFlareHalo);

glUniform1i(glGetUniformLocation(SunRaysLensFlareHalo, "LowBlurredSunTexture"), 0);

glUniform1i(glGetUniformLocation(SunRaysLensFlareHalo, "HighBlurredSunTexture"), 1);

glUniform1i(glGetUniformLocation(SunRaysLensFlareHalo, "DirtTexture"), 2);

glUniform1f(glGetUniformLocation(SunRaysLensFlareHalo, "Dispersal"), 0.1875f);

glUniform1f(glGetUniformLocation(SunRaysLensFlareHalo, "HaloWidth"), 0.45f);

glUniform1f(glGetUniformLocation(SunRaysLensFlareHalo, "Intensity"), 2.25f);

glUniform3f(glGetUniformLocation(SunRaysLensFlareHalo, "Distortion"), 0.94f, 0.97f, 1.00f);

glUseProgram(0);

float Kr = 0.0030f;

float Km = 0.0015f;

float ESun = 16.0f;

float g = -0.75f;

float InnerRadius = 10.0f;

float OuterRadius = 10.25f;

float Scale = 1.0f / (OuterRadius - InnerRadius);

float ScaleDepth = 0.25f;

float ScaleOverScaleDepth = Scale / ScaleDepth;

glUseProgram(Sky);

glUniform3f(glGetUniformLocation(Sky, "v3CameraPos"), 0.0f, InnerRadius, 0.0f);

glUniform3f(glGetUniformLocation(Sky, "v3InvWavelength"), 1.0f / powf(0.650f, 4.0f), 1.0f / powf(0.570f, 4.0f), 1.0f / powf(0.475f, 4.0f));

glUniform1f(glGetUniformLocation(Sky, "fCameraHeight"), InnerRadius);

glUniform1f(glGetUniformLocation(Sky, "fCameraHeight2"), InnerRadius \* InnerRadius);

glUniform1f(glGetUniformLocation(Sky, "fInnerRadius"), InnerRadius);

glUniform1f(glGetUniformLocation(Sky, "fInnerRadius2"), InnerRadius \* InnerRadius);

glUniform1f(glGetUniformLocation(Sky, "fOuterRadius"), OuterRadius);

glUniform1f(glGetUniformLocation(Sky, "fOuterRadius2"), OuterRadius \* OuterRadius);

glUniform1f(glGetUniformLocation(Sky, "fKrESun"), Kr \* ESun);

glUniform1f(glGetUniformLocation(Sky, "fKmESun"), Km \* ESun);

glUniform1f(glGetUniformLocation(Sky, "fKr4PI"), Kr \* 4.0f \* (float)M\_PI);

glUniform1f(glGetUniformLocation(Sky, "fKm4PI"), Km \* 4.0f \* (float)M\_PI);

glUniform1f(glGetUniformLocation(Sky, "fScale"), Scale);

glUniform1f(glGetUniformLocation(Sky, "fScaleDepth"), ScaleDepth);

glUniform1f(glGetUniformLocation(Sky, "fScaleOverScaleDepth"), ScaleOverScaleDepth);

glUniform1f(glGetUniformLocation(Sky, "g"), g);

glUniform1f(glGetUniformLocation(Sky, "g2"), g \* g);

glUniform1i(glGetUniformLocation(Sky, "Samples"), 4);

glUseProgram(0);

vec3 \*SkyDomeVertices = new vec3[112 \* 3], va, vb, vc, vd;

float stepa = (float)M\_PI \* 2.0f / 16, startb = asin(InnerRadius / OuterRadius), stepb = ((float)M\_PI\_2 - startb) / 4;

int pos = 0;

for(int y = 0; y < 3; y++)

{

float b = startb + stepb \* y;

for(int x = 0; x < 16; x++)

{

float a = stepa \* x;

va = vec3(sin(a) \* cos(b) \* OuterRadius, sin(b) \* OuterRadius, -cos(a) \* cos(b) \* OuterRadius);

vb = vec3(sin(a + stepa) \* cos(b) \* OuterRadius, sin(b) \* OuterRadius, -cos(a + stepa) \* cos(b) \* OuterRadius);

vc = vec3(sin(a + stepa) \* cos(b + stepb) \* OuterRadius, sin(b + stepb) \* OuterRadius, -cos(a + stepa) \* cos(b + stepb) \* OuterRadius);

vd = vec3(sin(a) \* cos(b + stepb) \* OuterRadius, sin(b + stepb) \* OuterRadius, -cos(a) \* cos(b + stepb) \* OuterRadius);

SkyDomeVertices[pos + 0] = va;

SkyDomeVertices[pos + 1] = vb;

SkyDomeVertices[pos + 2] = vc;

pos += 3;

SkyDomeVertices[pos + 0] = vc;

SkyDomeVertices[pos + 1] = vd;

SkyDomeVertices[pos + 2] = va;

pos += 3;

}

}

float b = startb + stepb \* 3;

for(int x = 0; x < 16; x++)

{

float a = stepa \* x;

va = vec3(sin(a) \* cos(b) \* OuterRadius, sin(b) \* OuterRadius, -cos(a) \* cos(b) \* OuterRadius);

vb = vec3(sin(a + stepa) \* cos(b) \* OuterRadius, sin(b) \* OuterRadius, -cos(a + stepa) \* cos(b) \* OuterRadius);

vc = vec3(0, OuterRadius, 0);

SkyDomeVertices[pos + 0] = va;

SkyDomeVertices[pos + 1] = vb;

SkyDomeVertices[pos + 2] = vc;

pos += 3;

}

glGenBuffers(1, &VBO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, 112 \* 3 \* 3 \* sizeof(float), SkyDomeVertices, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

delete [] SkyDomeVertices;

//glGenTextures(1, &ScreenTexture);

glGenTextures(1, &DepthTexture);

glGenTextures(4, SunTextures);

glGenFramebuffersEXT(1, &FBO);

glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, &vec4(vec3(0.0f), 1.0f));

glMaterialfv(GL\_FRONT\_AND\_BACK, GL\_AMBIENT, &vec4(1.0f));

glMaterialfv(GL\_FRONT\_AND\_BACK, GL\_DIFFUSE, &vec4(1.0f));

glEnable(GL\_LIGHT0);

glEnable(GL\_COLOR\_MATERIAL);

Camera.Look(vec3(10.0f, 1.75f, 0.0f), vec3(-512.0f, 0.0f, 0.0f), false);

return true;

}

void COpenGLRenderer::Render(float FrameTime)

{

// calculate sun direction, position and color

vec3 SunCDir = normalize(SunCPos);

vec3 SunWPos = Camera.Position + SunCPos;

float RefractionFactor = (1.0f - sqrt(max(0.0f, SunCDir.y)));

vec3 SunColor = 1.0f - vec3(0.0f, 0.5f, 1.0f) \* RefractionFactor;

// calculate ambient and diffuse light color

vec3 LightColor = 1.0f - vec3(0.0f, 0.25f, 0.5f) \* RefractionFactor;

float AmbientIntensity = 0.0625f + 0.1875f \* min(1.0f, max(0.0f, (0.375f + SunCDir.y) / 0.25f));

float DiffuseIntensity = 0.75f \* min(1.0f, max(0.0f, (0.03125f + SunCDir.y) / 0.0625f));

glViewport(0, 0, Width, Height);

//glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

//glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, ScreenTexture, 0);

//glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, DepthTexture, 0);

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

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&ProjectionMatrix);

// render sky

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glTranslatef(Camera.Position.x, Camera.Position.y - 10.0f /\*InnerRadius\*/, Camera.Position.z);

glEnableClientState(GL\_VERTEX\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glVertexPointer(3, GL\_FLOAT, 0, NULL);

glUseProgram(Sky);

glUniform3fv(glGetUniformLocation(Sky, "v3LightPos"), 1, &SunCDir);

glDrawArrays(GL\_TRIANGLES, 0, 112 \* 3);

glUseProgram(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glDisableClientState(GL\_VERTEX\_ARRAY);

// render scene

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glLightfv(GL\_LIGHT0, GL\_AMBIENT, &vec4(vec3(AmbientIntensity), 1.0f));

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, &vec4(LightColor \* DiffuseIntensity, 1.0f));

glLightfv(GL\_LIGHT0, GL\_POSITION, &vec4(SunWPos, 1.0f));

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

glEnable(GL\_LIGHTING);

glColor3f(1.0f, 1.0f, 1.0f);

glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, Texture);

glBegin(GL\_QUADS);

glNormal3f( 0.0f, 0.0f, 1.0f);

glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5f, -0.5f, 0.5f);

glTexCoord2f(1.0f, 0.0f); glVertex3f( 0.5f, -0.5f, 0.5f);

glTexCoord2f(1.0f, 1.0f); glVertex3f( 0.5f, 0.5f, 0.5f);

glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5f, 0.5f, 0.5f);

glNormal3f( 0.0f, 0.0f, -1.0f);

glTexCoord2f(0.0f, 0.0f); glVertex3f( 0.5f, -0.5f, -0.5f);

glTexCoord2f(1.0f, 0.0f); glVertex3f(-0.5f, -0.5f, -0.5f);

glTexCoord2f(1.0f, 1.0f); glVertex3f(-0.5f, 0.5f, -0.5f);

glTexCoord2f(0.0f, 1.0f); glVertex3f( 0.5f, 0.5f, -0.5f);

glNormal3f( 1.0f, 0.0f, 0.0f);

glTexCoord2f(0.0f, 0.0f); glVertex3f( 0.5f, -0.5f, 0.5f);

glTexCoord2f(1.0f, 0.0f); glVertex3f( 0.5f, -0.5f, -0.5f);

glTexCoord2f(1.0f, 1.0f); glVertex3f( 0.5f, 0.5f, -0.5f);

glTexCoord2f(0.0f, 1.0f); glVertex3f( 0.5f, 0.5f, 0.5f);

glNormal3f(-1.0f, 0.0f, 0.0f);

glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5f, -0.5f, -0.5f);

glTexCoord2f(1.0f, 0.0f); glVertex3f(-0.5f, -0.5f, 0.5f);

glTexCoord2f(1.0f, 1.0f); glVertex3f(-0.5f, 0.5f, 0.5f);

glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5f, 0.5f, -0.5f);

glNormal3f( 0.0f, 1.0f, 0.0f);

glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5f, 0.5f, 0.5f);

glTexCoord2f(1.0f, 0.0f); glVertex3f( 0.5f, 0.5f, 0.5f);

glTexCoord2f(1.0f, 1.0f); glVertex3f( 0.5f, 0.5f, -0.5f);

glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5f, 0.5f, -0.5f);

glNormal3f( 0.0f, -1.0f, 0.0f);

glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5f, -0.5f, -0.5f);

glTexCoord2f(1.0f, 0.0f); glVertex3f( 0.5f, -0.5f, -0.5f);

glTexCoord2f(1.0f, 1.0f); glVertex3f( 0.5f, -0.5f, 0.5f);

glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5f, -0.5f, 0.5f);

glEnd();

glDisable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBegin(GL\_QUADS);

glNormal3f( 0.0f, 1.0f, 0.0f);

glVertex3f(-4.0f, -0.5f, 4.0f);

glVertex3f( 4.0f, -0.5f, 4.0f);

glVertex3f( 4.0f, -0.5f, -4.0f);

glVertex3f(-4.0f, -0.5f, -4.0f);

glEnd();

GLUquadric \*obj = gluNewQuadric();

for(int z = -2; z <= 2; z += 1)

{

for(int x = -2; x <= 2; x += 1)

{

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glTranslatef((float)x, -0.25f, (float)z);

gluSphere(obj, 0.25f, 32, 32);

}

}

gluDeleteQuadric(obj);

glDisable(GL\_LIGHTING);

glDisable(GL\_CULL\_FACE);

glDisable(GL\_DEPTH\_TEST);

// copy depth buffer to texture

glBindTexture(GL\_TEXTURE\_2D, DepthTexture);

glCopyTexSubImage2D(GL\_TEXTURE\_2D, 0, 0, 0, 0, 0, Width, Height);

glBindTexture(GL\_TEXTURE\_2D, 0);

//glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

// test if sun is inside camera frustum

bool CalculateSunRaysLensFlareHalo = false;

int Test = 0, Tests = 16;

float Angle = 0.0f, AngleInc = 360.0f / Tests;

mat4x4 VPB = BiasMatrix \* ProjectionMatrix \* ViewMatrix;

while(Test < Tests && !CalculateSunRaysLensFlareHalo)

{

vec3 SunWEdge = SunWPos + rotate(Camera.X, Angle, Camera.Z) \* SunR;

if(SunWEdge.y - Camera.Position.y > 0.0f)

{

vec4 SunPosProj = VPB \* vec4(SunWEdge, 1.0f);

SunPosProj /= SunPosProj.w;

CalculateSunRaysLensFlareHalo |= (SunPosProj.x >= 0.0f && SunPosProj.x <= 1.0f && SunPosProj.y >= 0.0f && SunPosProj.y <= 1.0f && SunPosProj.z >= 0.0f && SunPosProj.z <= 1.0f);

}

Angle += AngleInc;

Test++;

}

// if it is then calculate lens flare

if(CalculateSunRaysLensFlareHalo)

{

vec4 SunPosProj = VPB \* vec4(SunWPos, 1.0f);

SunPosProj /= SunPosProj.w;

glViewport(0, 0, SunTextureWidth, SunTextureHeight);

// render sun sphere

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[1], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glClear(GL\_COLOR\_BUFFER\_BIT);

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

double pl[] = {0.0f, 1.0f, 0.0f, -Camera.Position.y};

glClipPlane(GL\_CLIP\_PLANE0, pl);

glEnable(GL\_CLIP\_PLANE0);

glTranslatef(SunWPos.x, SunWPos.y, SunWPos.z);

glColor3fv(&SunColor);

glEnable(GL\_CULL\_FACE);

GLUquadric \*obj = gluNewQuadric();

gluSphere(obj, SunR, 16, 16);

gluDeleteQuadric(obj);

glDisable(GL\_CULL\_FACE);

glDisable(GL\_CLIP\_PLANE0);

// test if sun sphere is behind scene geometry

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[0], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, SunTextures[1]);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, DepthTexture);

glUseProgram(SunDepthTest);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, 0);

// blur sun sphere horizontally (low)

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[3], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[0]);

glUseProgram(BlurH);

glUniform1i(glGetUniformLocation(BlurH, "Width"), 1);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

// blur sun sphere vertically (low)

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[1], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[3]);

glUseProgram(BlurV);

glUniform1i(glGetUniformLocation(BlurV, "Width"), 1);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

// blur sun sphere horizontally (high)

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[3], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[0]);

glUseProgram(BlurH);

glUniform1i(glGetUniformLocation(BlurH, "Width"), 10);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

// blur sun sphere vertically (high)

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[2], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[3]);

glUseProgram(BlurV);

glUniform1i(glGetUniformLocation(BlurV, "Width"), 10);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

// blur sun sphere radially and calculate lens flare and halo and apply dirt texture

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, SunTextures[3], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, SunTextures[1]);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, SunTextures[2]);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_2D, DirtTexture);

glUseProgram(SunRaysLensFlareHalo);

glUniform2fv(glGetUniformLocation(SunRaysLensFlareHalo, "SunPosProj"), 1, &SunPosProj);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

glViewport(0, 0, Width, Height);

}

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&ortho(0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f));

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

/\*glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, ScreenTexture);

glColor3f(1.0f, 1.0f, 1.0f);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f); glVertex2f(0.0f, 0.0f);

glTexCoord2f(1.0f, 0.0f); glVertex2f(1.0f, 0.0f);

glTexCoord2f(1.0f, 1.0f); glVertex2f(1.0f, 1.0f);

glTexCoord2f(0.0f, 1.0f); glVertex2f(0.0f, 1.0f);

glEnd();

glBindTexture(GL\_TEXTURE\_2D, 0);

glDisable(GL\_TEXTURE\_2D);\*/

// blend sun texture over the screen

if(CalculateSunRaysLensFlareHalo)

{

glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, SunTextures[3]);

glBlendFunc(GL\_ONE, GL\_ONE\_MINUS\_SRC\_COLOR);

glEnable(GL\_BLEND);

glColor3f(1.0f, 1.0f, 1.0f);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f); glVertex2f(0.0f, 0.0f);

glTexCoord2f(1.0f, 0.0f); glVertex2f(1.0f, 0.0f);

glTexCoord2f(1.0f, 1.0f); glVertex2f(1.0f, 1.0f);

glTexCoord2f(0.0f, 1.0f); glVertex2f(0.0f, 1.0f);

glEnd();

glDisable(GL\_BLEND);

glBindTexture(GL\_TEXTURE\_2D, 0);

glDisable(GL\_TEXTURE\_2D);

}

// move sun

if(!Pause)

{

SunCPos = rotate(SunCPos, 0.25f \* FrameTime, SunRotVec);

}

}

void COpenGLRenderer::Resize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

ProjectionMatrix = perspective(45.0f, (float)Width / (float)Height, 0.125f, 512.0f);

/\*glBindTexture(GL\_TEXTURE\_2D, ScreenTexture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);\*/

glBindTexture(GL\_TEXTURE\_2D, DepthTexture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT24, Width, Height, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);

SunTextureWidth = Width / 2;

SunTextureHeight = Height / 2;

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

{

glBindTexture(GL\_TEXTURE\_2D, SunTextures[i]);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, SunTextureWidth, SunTextureHeight, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);

}

glUseProgram(BlurH);

glUniform1f(glGetUniformLocation(BlurH, "odw"), 1.0f / (float)SunTextureWidth);

glUseProgram(BlurV);

glUniform1f(glGetUniformLocation(BlurV, "odh"), 1.0f / (float)SunTextureHeight);

glUseProgram(0);

}

void COpenGLRenderer::Destroy()

{

Texture.Destroy();

DirtTexture.Destroy();

//glDeleteTextures(1, &ScreenTexture);

glDeleteTextures(1, &DepthTexture);

glDeleteTextures(4, SunTextures);

Sky.Destroy();

SunDepthTest.Destroy();

BlurH.Destroy();

BlurV.Destroy();

SunRaysLensFlareHalo.Destroy();

glDeleteBuffers(1, &VBO);

if(GLEW\_EXT\_framebuffer\_object)

{

glDeleteFramebuffersEXT(1, &FBO);

}

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer OpenGLRenderer;

// ----------------------------------------------------------------------------------------------------------------------------

CString ModuleDirectory, ErrorLog;

// ----------------------------------------------------------------------------------------------------------------------------

void GetModuleDirectory()

{

char \*moduledirectory = new char[256];

GetModuleFileName(GetModuleHandle(NULL), moduledirectory, 256);

\*(strrchr(moduledirectory, '\\') + 1) = 0;

ModuleDirectory = moduledirectory;

delete [] moduledirectory;

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView::COpenGLView()

{

}

COpenGLView::~COpenGLView()

{

}

bool COpenGLView::Init(HINSTANCE hInstance, char \*Title, int Width, int Height, int Samples)

{

this->Title = Title;

this->Width = Width;

this->Height = Height;

WNDCLASSEX WndClassEx;

memset(&WndClassEx, 0, sizeof(WNDCLASSEX));

WndClassEx.cbSize = sizeof(WNDCLASSEX);

WndClassEx.style = CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

WndClassEx.lpfnWndProc = WndProc;

WndClassEx.hInstance = hInstance;

WndClassEx.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hCursor = LoadCursor(NULL, IDC\_ARROW);

WndClassEx.lpszClassName = "Win32OpenGLWindowClass";

if(RegisterClassEx(&WndClassEx) == 0)

{

ErrorLog.Set("RegisterClassEx failed!");

return false;

}

DWORD Style = WS\_OVERLAPPEDWINDOW | WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN;

hWnd = CreateWindowEx(WS\_EX\_APPWINDOW, WndClassEx.lpszClassName, Title, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL);

if(hWnd == NULL)

{

ErrorLog.Set("CreateWindowEx failed!");

return false;

}

HDC hDC = GetDC(hWnd);

if(hDC == NULL)

{

ErrorLog.Set("GetDC failed!");

return false;

}

PIXELFORMATDESCRIPTOR pfd;

memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));

pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);

pfd.nVersion = 1;

pfd.dwFlags = PFD\_DRAW\_TO\_WINDOW | PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER;

pfd.iPixelType = PFD\_TYPE\_RGBA;

pfd.cColorBits = 32;

pfd.cDepthBits = 24;

pfd.iLayerType = PFD\_MAIN\_PLANE;

int PixelFormat = ChoosePixelFormat(hDC, &pfd);

if(PixelFormat == 0)

{

ErrorLog.Set("ChoosePixelFormat failed!");

return false;

}

static int MSAAPixelFormat = 0;

if(SetPixelFormat(hDC, MSAAPixelFormat == 0 ? PixelFormat : MSAAPixelFormat, &pfd) == FALSE)

{

ErrorLog.Set("SetPixelFormat failed!");

return false;

}

hGLRC = wglCreateContext(hDC);

if(hGLRC == NULL)

{

ErrorLog.Set("wglCreateContext failed!");

return false;

}

if(wglMakeCurrent(hDC, hGLRC) == FALSE)

{

ErrorLog.Set("wglMakeCurrent failed!");

return false;

}

if(glewInit() != GLEW\_OK)

{

ErrorLog.Set("glewInit failed!");

return false;

}

if(!GLEW\_VERSION\_2\_1)

{

ErrorLog.Set("OpenGL 2.1 not supported!");

return false;

}

if(MSAAPixelFormat == 0 && Samples > 0)

{

if(GLEW\_ARB\_multisample && WGLEW\_ARB\_pixel\_format)

{

while(Samples > 0)

{

UINT NumFormats = 0;

int PFAttribs[] =

{

WGL\_DRAW\_TO\_WINDOW\_ARB, GL\_TRUE,

WGL\_SUPPORT\_OPENGL\_ARB, GL\_TRUE,

WGL\_DOUBLE\_BUFFER\_ARB, GL\_TRUE,

WGL\_PIXEL\_TYPE\_ARB, WGL\_TYPE\_RGBA\_ARB,

WGL\_COLOR\_BITS\_ARB, 32,

WGL\_DEPTH\_BITS\_ARB, 24,

WGL\_ACCELERATION\_ARB, WGL\_FULL\_ACCELERATION\_ARB,

WGL\_SAMPLE\_BUFFERS\_ARB, GL\_TRUE,

WGL\_SAMPLES\_ARB, Samples,

0

};

if(wglChoosePixelFormatARB(hDC, PFAttribs, NULL, 1, &MSAAPixelFormat, &NumFormats) == TRUE && NumFormats > 0) break;

Samples--;

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

UnregisterClass(WndClassEx.lpszClassName, hInstance);

return Init(hInstance, Title, Width, Height, Samples);

}

else

{

Samples = 0;

}

}

this->Samples = Samples;

GetModuleDirectory();

glGetIntegerv(GL\_MAX\_TEXTURE\_SIZE, &gl\_max\_texture\_size);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glGetIntegerv(GL\_MAX\_TEXTURE\_MAX\_ANISOTROPY\_EXT, &gl\_max\_texture\_max\_anisotropy\_ext);

}

if(WGLEW\_EXT\_swap\_control)

{

wglSwapIntervalEXT(0);

}

return OpenGLRenderer.Init();

}

void COpenGLView::Show(bool Maximized)

{

RECT dRect, wRect, cRect;

GetWindowRect(GetDesktopWindow(), &dRect);

GetWindowRect(hWnd, &wRect);

GetClientRect(hWnd, &cRect);

wRect.right += Width - cRect.right;

wRect.bottom += Height - cRect.bottom;

wRect.right -= wRect.left;

wRect.bottom -= wRect.top;

wRect.left = dRect.right / 2 - wRect.right / 2;

wRect.top = dRect.bottom / 2 - wRect.bottom / 2;

MoveWindow(hWnd, wRect.left, wRect.top, wRect.right, wRect.bottom, FALSE);

ShowWindow(hWnd, Maximized ? SW\_SHOWMAXIMIZED : SW\_SHOWNORMAL);

}

void COpenGLView::MessageLoop()

{

MSG Msg;

while(GetMessage(&Msg, NULL, 0, 0) > 0)

{

TranslateMessage(&Msg);

DispatchMessage(&Msg);

}

}

void COpenGLView::Destroy()

{

if(GLEW\_VERSION\_2\_1)

{

OpenGLRenderer.Destroy();

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

}

void COpenGLView::OnKeyDown(UINT Key)

{

switch(Key)

{

case VK\_ADD:

OpenGLRenderer.SunCPos = rotate(OpenGLRenderer.SunCPos, + 1.0f, OpenGLRenderer.SunRotVec);

break;

case VK\_SUBTRACT:

OpenGLRenderer.SunCPos = rotate(OpenGLRenderer.SunCPos, - 1.0f, OpenGLRenderer.SunRotVec);

break;

case VK\_SPACE:

OpenGLRenderer.Pause = !OpenGLRenderer.Pause;

break;

}

}

void COpenGLView::OnMouseMove(int X, int Y)

{

if(GetKeyState(VK\_RBUTTON) & 0x80)

{

Camera.OnMouseMove(LastX - X, LastY - Y);

LastX = X;

LastY = Y;

}

}

void COpenGLView::OnMouseWheel(short zDelta)

{

Camera.OnMouseWheel(zDelta);

}

void COpenGLView::OnPaint()

{

static DWORD LastFPSTime = GetTickCount(), LastFrameTime = LastFPSTime, FPS = 0;

PAINTSTRUCT ps;

HDC hDC = BeginPaint(hWnd, &ps);

DWORD Time = GetTickCount();

float FrameTime = (Time - LastFrameTime) \* 0.001f;

LastFrameTime = Time;

if(Time - LastFPSTime > 1000)

{

CString Text = Title;

if(OpenGLRenderer.Text[0] != 0)

{

Text.Append(" - " + OpenGLRenderer.Text);

}

Text.Append(" - %dx%d", Width, Height);

Text.Append(", ATF %dx", gl\_max\_texture\_max\_anisotropy\_ext);

Text.Append(", MSAA %dx", Samples);

Text.Append(", FPS: %d", FPS);

Text.Append(" - %s", glGetString(GL\_RENDERER));

SetWindowText(hWnd, Text);

LastFPSTime = Time;

FPS = 0;

}

else

{

FPS++;

}

BYTE Keys = 0x00;

if(GetKeyState('W') & 0x80) Keys |= 0x01;

if(GetKeyState('S') & 0x80) Keys |= 0x02;

if(GetKeyState('A') & 0x80) Keys |= 0x04;

if(GetKeyState('D') & 0x80) Keys |= 0x08;

if(GetKeyState('R') & 0x80) Keys |= 0x10;

if(GetKeyState('F') & 0x80) Keys |= 0x20;

if(GetKeyState(VK\_SHIFT) & 0x80) Keys |= 0x40;

if(GetKeyState(VK\_CONTROL) & 0x80) Keys |= 0x80;

if(Keys & 0x3F)

{

Camera.Move(Camera.OnKeys(Keys, FrameTime));

}

OpenGLRenderer.Render(FrameTime);

SwapBuffers(hDC);

EndPaint(hWnd, &ps);

InvalidateRect(hWnd, NULL, FALSE);

}

void COpenGLView::OnRButtonDown(int X, int Y)

{

LastX = X;

LastY = Y;

}

void COpenGLView::OnSize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

OpenGLRenderer.Resize(Width, Height);

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView OpenGLView;

// ----------------------------------------------------------------------------------------------------------------------------

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)

{

switch(uiMsg)

{

case WM\_CLOSE:

PostQuitMessage(0);

break;

case WM\_MOUSEMOVE:

OpenGLView.OnMouseMove(LOWORD(lParam), HIWORD(lParam));

break;

case 0x020A: // WM\_MOUSWHEEL

OpenGLView.OnMouseWheel(HIWORD(wParam));

break;

case WM\_KEYDOWN:

OpenGLView.OnKeyDown((UINT)wParam);

break;

case WM\_PAINT:

OpenGLView.OnPaint();

break;

case WM\_RBUTTONDOWN:

OpenGLView.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_SIZE:

OpenGLView.OnSize(LOWORD(lParam), HIWORD(lParam));

break;

default:

return DefWindowProc(hWnd, uiMsg, wParam, lParam);

}

return 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine, int iShow)

{

char \*AppName = "Atmospheric light scattering";

if(OpenGLView.Init(hInstance, AppName, 800, 600, 4))

{

OpenGLView.Show();

OpenGLView.MessageLoop();

}

else

{

MessageBox(NULL, ErrorLog, AppName, MB\_OK | MB\_ICONERROR);

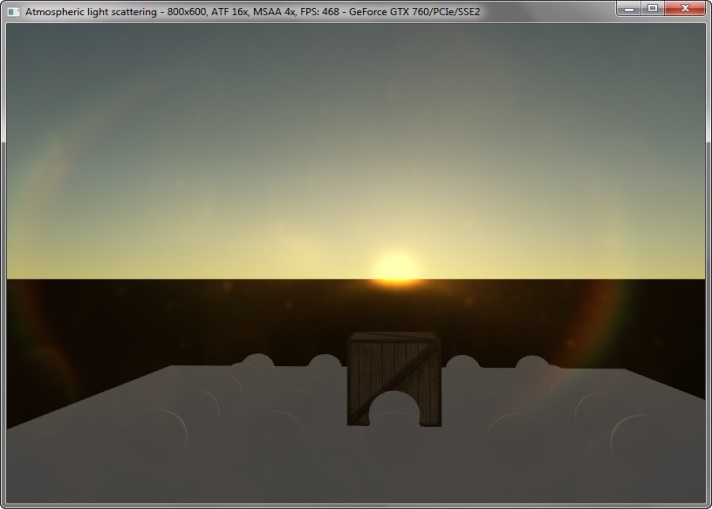
}

OpenGLView.Destroy();

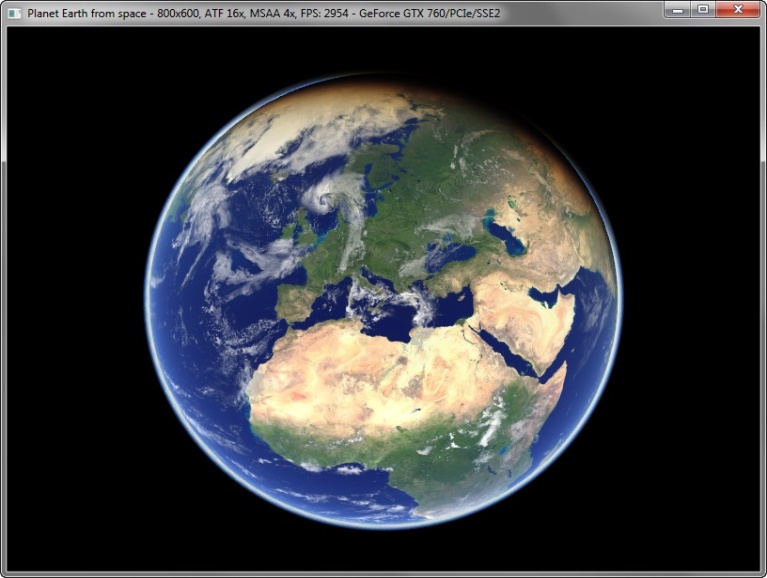
return 0;

}

////////////////////////////////插图///////////////////////////////////////



///////////////////////////////////插图////////////////////////////////////////////



1. groundfromatmosphere.vs

// Atmospheric scattering vertex shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform vec3 v3CameraPos; // The camera's current position

uniform vec3 v3LightPos; // The direction vector to the light source

uniform vec3 v3InvWavelength; // 1 / pow(wavelength, 4) for the red, green, and blue channels

uniform float fCameraHeight; // The camera's current height

uniform float fCameraHeight2; // fCameraHeight^2

uniform float fOuterRadius; // The outer (atmosphere) radius

uniform float fOuterRadius2; // fOuterRadius^2

uniform float fInnerRadius; // The inner (planetary) radius

uniform float fInnerRadius2; // fInnerRadius^2

uniform float fKrESun; // Kr \* ESun

uniform float fKmESun; // Km \* ESun

uniform float fKr4PI; // Kr \* 4 \* PI

uniform float fKm4PI; // Km \* 4 \* PI

uniform float fScale; // 1 / (fOuterRadius - fInnerRadius)

uniform float fScaleDepth; // The scale depth (i.e. the altitude at which the atmosphere's average density is found)

uniform float fScaleOverScaleDepth; // fScale / fScaleDepth

uniform int Samples;

float scale(float fCos)

{

float x = 1.0 - fCos;

return fScaleDepth \* exp(-0.00287 + x \* (0.459 + x \* (3.83 + x \* (-6.80 + x \* 5.25))));

}

void main(void)

{

// Get the ray from the camera to the vertex, and its length (which is the far point of the ray passing through the atmosphere)

vec3 v3Pos = (gl\_TextureMatrix[0] \* gl\_Vertex).xyz;

vec3 v3Ray = v3Pos - v3CameraPos;

float fFar = length(v3Ray);

v3Ray /= fFar;

// Calculate the ray's starting position, then calculate its scattering offset

vec3 v3Start = v3CameraPos;

float fDepth = exp((fInnerRadius - fCameraHeight) / fScaleDepth);

float fCameraAngle = dot(-v3Ray, v3Pos) / length(v3Pos);

float fLightAngle = dot(v3LightPos, v3Pos) / length(v3Pos);

float fCameraScale = scale(fCameraAngle);

float fLightScale = scale(fLightAngle);

float fCameraOffset = fDepth \* fCameraScale;

float fTemp = (fLightScale + fCameraScale);

// Initialize the scattering loop variables

float fSampleLength = fFar / Samples;

float fScaledLength = fSampleLength \* fScale;

vec3 v3SampleRay = v3Ray \* fSampleLength;

vec3 v3SamplePoint = v3Start + v3SampleRay \* 0.5;

// Now loop through the sample rays

vec3 v3FrontColor = vec3(0.0);

vec3 v3Attenuate;

for(int i = 0; i < Samples; i++)

{

float fHeight = length(v3SamplePoint);

float fDepth = exp(fScaleOverScaleDepth \* (fInnerRadius - fHeight));

float fScatter = fDepth\*fTemp - fCameraOffset;

v3Attenuate = exp(-fScatter \* (v3InvWavelength \* fKr4PI + fKm4PI));

v3FrontColor += v3Attenuate \* (fDepth \* fScaledLength);

v3SamplePoint += v3SampleRay;

}

gl\_FrontColor.rgb = v3FrontColor \* (v3InvWavelength \* fKrESun + fKmESun);

// Calculate the attenuation factor for the ground

gl\_FrontSecondaryColor.rgb = v3Attenuate;

gl\_Position = gl\_ModelViewProjectionMatrix \* gl\_Vertex;

gl\_TexCoord[0] = gl\_MultiTexCoord0;

}

1. groundfromatmosphere.fs

// Atmospheric scattering fragment shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform sampler2D s2Tex1;

uniform sampler2D s2Tex2;

//uniform sampler2D s2Tex3;

void main (void)

{

vec3 Clouds = texture2D(s2Tex2, gl\_TexCoord[0].st).rgb;

gl\_FragColor = vec4(gl\_Color.rgb + mix(texture2D(s2Tex1, gl\_TexCoord[0].st).rgb, Clouds, dot(vec3(0.33), Clouds)) \* gl\_SecondaryColor.rgb, 1.0);

}

1. groundfromspace.vs

// Atmospheric scattering vertex shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform vec3 v3CameraPos; // The camera's current position

uniform vec3 v3LightPos; // The direction vector to the light source

uniform vec3 v3InvWavelength; // 1 / pow(wavelength, 4) for the red, green, and blue channels

uniform float fCameraHeight; // The camera's current height

uniform float fCameraHeight2; // fCameraHeight^2

uniform float fOuterRadius; // The outer (atmosphere) radius

uniform float fOuterRadius2; // fOuterRadius^2

uniform float fInnerRadius; // The inner (planetary) radius

uniform float fInnerRadius2; // fInnerRadius^2

uniform float fKrESun; // Kr \* ESun

uniform float fKmESun; // Km \* ESun

uniform float fKr4PI; // Kr \* 4 \* PI

uniform float fKm4PI; // Km \* 4 \* PI

uniform float fScale; // 1 / (fOuterRadius - fInnerRadius)

uniform float fScaleDepth; // The scale depth (i.e. the altitude at which the atmosphere's average density is found)

uniform float fScaleOverScaleDepth; // fScale / fScaleDepth

uniform int Samples;

float scale(float fCos)

{

float x = 1.0 - fCos;

return fScaleDepth \* exp(-0.00287 + x \* (0.459 + x \* (3.83 + x \* (-6.80 + x \* 5.25))));

}

void main(void)

{

// Get the ray from the camera to the vertex and its length (which is the far point of the ray passing through the atmosphere)

vec3 v3Pos = (gl\_TextureMatrix[0] \* gl\_Vertex).xyz;

vec3 v3Ray = v3Pos - v3CameraPos;

float fFar = length(v3Ray);

v3Ray /= fFar;

// Calculate the closest intersection of the ray with the outer atmosphere (which is the near point of the ray passing through the atmosphere)

float B = 2.0 \* dot(v3CameraPos, v3Ray);

float C = fCameraHeight2 - fOuterRadius2;

float fDet = max(0.0, B \* B - 4.0 \* C);

float fNear = 0.5 \* (-B - sqrt(fDet));

// Calculate the ray's starting position, then calculate its scattering offset

vec3 v3Start = v3CameraPos + v3Ray \* fNear;

fFar -= fNear;

float fDepth = exp((fInnerRadius - fOuterRadius) / fScaleDepth);

float fCameraAngle = dot(-v3Ray, v3Pos) / length(v3Pos);

float fLightAngle = dot(v3LightPos, v3Pos) / length(v3Pos);

float fCameraScale = scale(fCameraAngle);

float fLightScale = scale(fLightAngle);

float fCameraOffset = fDepth \* fCameraScale;

float fTemp = (fLightScale + fCameraScale);

// Initialize the scattering loop variables

float fSampleLength = fFar / Samples;

float fScaledLength = fSampleLength \* fScale;

vec3 v3SampleRay = v3Ray \* fSampleLength;

vec3 v3SamplePoint = v3Start + v3SampleRay \* 0.5;

// Now loop through the sample rays

vec3 v3FrontColor = vec3(0.0);

vec3 v3Attenuate;

for(int i = 0; i < Samples; i++)

{

float fHeight = length(v3SamplePoint);

float fDepth = exp(fScaleOverScaleDepth \* (fInnerRadius - fHeight));

float fScatter = fDepth\*fTemp - fCameraOffset;

v3Attenuate = exp(-fScatter \* (v3InvWavelength \* fKr4PI + fKm4PI));

v3FrontColor += v3Attenuate \* (fDepth \* fScaledLength);

v3SamplePoint += v3SampleRay;

}

gl\_FrontColor.rgb = v3FrontColor \* (v3InvWavelength \* fKrESun + fKmESun);

// Calculate the attenuation factor for the ground

gl\_FrontSecondaryColor.rgb = v3Attenuate;

gl\_Position = gl\_ModelViewProjectionMatrix \* gl\_Vertex;

gl\_TexCoord[0] = gl\_MultiTexCoord0;

}

1. groundfromspace.fs

// Atmospheric scattering fragment shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform sampler2D s2Tex1;

uniform sampler2D s2Tex2;

//uniform sampler2D s2Tex3;

void main (void)

{

vec3 Clouds = texture2D(s2Tex2, gl\_TexCoord[0].st).rgb;

gl\_FragColor = vec4(gl\_Color.rgb + mix(texture2D(s2Tex1, gl\_TexCoord[0].st).rgb, Clouds, dot(vec3(0.33), Clouds)) \* gl\_SecondaryColor.rgb, 1.0);

}

1. skyfromatmosphere.vs

// Atmospheric scattering vertex shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform vec3 v3CameraPos; // The camera's current position

uniform vec3 v3LightPos; // The direction vector to the light source

uniform vec3 v3InvWavelength; // 1 / pow(wavelength, 4) for the red, green, and blue channels

uniform float fCameraHeight; // The camera's current height

uniform float fCameraHeight2; // fCameraHeight^2

uniform float fOuterRadius; // The outer (atmosphere) radius

uniform float fOuterRadius2; // fOuterRadius^2

uniform float fInnerRadius; // The inner (planetary) radius

uniform float fInnerRadius2; // fInnerRadius^2

uniform float fKrESun; // Kr \* ESun

uniform float fKmESun; // Km \* ESun

uniform float fKr4PI; // Kr \* 4 \* PI

uniform float fKm4PI; // Km \* 4 \* PI

uniform float fScale; // 1 / (fOuterRadius - fInnerRadius)

uniform float fScaleDepth; // The scale depth (i.e. the altitude at which the atmosphere's average density is found)

uniform float fScaleOverScaleDepth; // fScale / fScaleDepth

uniform int Samples;

varying vec3 v3Direction;

float scale(float fCos)

{

float x = 1.0 - fCos;

return fScaleDepth \* exp(-0.00287 + x \* (0.459 + x \* (3.83 + x \* (-6.80 + x \* 5.25))));

}

void main(void)

{

// Get the ray from the camera to the vertex, and its length (which is the far point of the ray passing through the atmosphere)

vec3 v3Pos = (gl\_TextureMatrix[0] \* gl\_Vertex).xyz;

vec3 v3Ray = v3Pos - v3CameraPos;

float fFar = length(v3Ray);

v3Ray /= fFar;

// Calculate the ray's starting position, then calculate its scattering offset

vec3 v3Start = v3CameraPos;

float fHeight = length(v3Start);

float fDepth = exp(fScaleOverScaleDepth \* (fInnerRadius - fCameraHeight));

float fStartAngle = dot(v3Ray, v3Start) / fHeight;

float fStartOffset = fDepth \* scale(fStartAngle);

// Initialize the scattering loop variables

float fSampleLength = fFar / Samples;

float fScaledLength = fSampleLength \* fScale;

vec3 v3SampleRay = v3Ray \* fSampleLength;

vec3 v3SamplePoint = v3Start + v3SampleRay \* 0.5;

// Now loop through the sample rays

vec3 v3FrontColor = vec3(0.0);

for(int i = 0; i < Samples; i++)

{

float fHeight = length(v3SamplePoint);

float fDepth = exp(fScaleOverScaleDepth \* (fInnerRadius - fHeight));

float fLightAngle = dot(v3LightPos, v3SamplePoint) / fHeight;

float fCameraAngle = dot(v3Ray, v3SamplePoint) / fHeight;

float fScatter = (fStartOffset + fDepth \* (scale(fLightAngle) - scale(fCameraAngle)));

vec3 v3Attenuate = exp(-fScatter \* (v3InvWavelength \* fKr4PI + fKm4PI));

v3FrontColor += v3Attenuate \* (fDepth \* fScaledLength);

v3SamplePoint += v3SampleRay;

}

// Finally, scale the Mie and Rayleigh colors and set up the varying variables for the pixel shader

gl\_FrontSecondaryColor.rgb = v3FrontColor \* fKmESun;

gl\_FrontColor.rgb = v3FrontColor \* (v3InvWavelength \* fKrESun);

gl\_Position = gl\_ModelViewProjectionMatrix \* gl\_Vertex;

v3Direction = v3CameraPos - v3Pos;

}

1. skyfromatmosphere.fs

// Atmospheric scattering fragment shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform vec3 v3LightPos;

uniform float g;

uniform float g2;

varying vec3 v3Direction;

void main (void)

{

float fCos = dot(v3LightPos, v3Direction) / length(v3Direction);

float fRayleighPhase = 1.0 + fCos \* fCos;

float fMiePhase = (1.0 - g2) / (2.0 + g2) \* (1.0 + fCos \* fCos) / pow(1.0 + g2 - 2.0 \* g \* fCos, 1.5);

gl\_FragColor = vec4(1.0 - exp(-1.5 \* (fRayleighPhase \* gl\_Color.rgb + fMiePhase \* gl\_SecondaryColor.rgb)), 1.0);

}

7>skyfromspace.vs

// Atmospheric scattering vertex shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform vec3 v3CameraPos; // The camera's current position

uniform vec3 v3LightPos; // The direction vector to the light source

uniform vec3 v3InvWavelength; // 1 / pow(wavelength, 4) for the red, green, and blue channels

uniform float fCameraHeight; // The camera's current height

uniform float fCameraHeight2; // fCameraHeight^2

uniform float fOuterRadius; // The outer (atmosphere) radius

uniform float fOuterRadius2; // fOuterRadius^2

uniform float fInnerRadius; // The inner (planetary) radius

uniform float fInnerRadius2; // fInnerRadius^2

uniform float fKrESun; // Kr \* ESun

uniform float fKmESun; // Km \* ESun

uniform float fKr4PI; // Kr \* 4 \* PI

uniform float fKm4PI; // Km \* 4 \* PI

uniform float fScale; // 1 / (fOuterRadius - fInnerRadius)

uniform float fScaleDepth; // The scale depth (i.e. the altitude at which the atmosphere's average density is found)

uniform float fScaleOverScaleDepth; // fScale / fScaleDepth

uniform int Samples;

varying vec3 v3Direction;

float scale(float fCos)

{

float x = 1.0 - fCos;

return fScaleDepth \* exp(-0.00287 + x \* (0.459 + x \* (3.83 + x \* (-6.80 + x \* 5.25))));

}

void main(void)

{

// Get the ray from the camera to the vertex and its length (which is the far point of the ray passing through the atmosphere)

vec3 v3Pos = (gl\_TextureMatrix[0] \* gl\_Vertex).xyz;

vec3 v3Ray = v3Pos - v3CameraPos;

float fFar = length(v3Ray);

v3Ray /= fFar;

// Calculate the closest intersection of the ray with the outer atmosphere (which is the near point of the ray passing through the atmosphere)

float B = 2.0 \* dot(v3CameraPos, v3Ray);

float C = fCameraHeight2 - fOuterRadius2;

float fDet = max(0.0, B \* B - 4.0 \* C);

float fNear = 0.5 \* (-B - sqrt(fDet));

// Calculate the ray's starting position, then calculate its scattering offset

vec3 v3Start = v3CameraPos + v3Ray \* fNear;

fFar -= fNear;

float fStartAngle = dot(v3Ray, v3Start) / fOuterRadius;

float fStartDepth = exp(-1.0 / fScaleDepth);

float fStartOffset = fStartDepth \* scale(fStartAngle);

// Initialize the scattering loop variables

float fSampleLength = fFar / Samples;

float fScaledLength = fSampleLength \* fScale;

vec3 v3SampleRay = v3Ray \* fSampleLength;

vec3 v3SamplePoint = v3Start + v3SampleRay \* 0.5;

// Now loop through the sample rays

vec3 v3FrontColor = vec3(0.0);

for(int i = 0; i < Samples; i++)

{

float fHeight = length(v3SamplePoint);

float fDepth = exp(fScaleOverScaleDepth \* (fInnerRadius - fHeight));

float fLightAngle = dot(v3LightPos, v3SamplePoint) / fHeight;

float fCameraAngle = dot(v3Ray, v3SamplePoint) / fHeight;

float fScatter = (fStartOffset + fDepth \* (scale(fLightAngle) - scale(fCameraAngle)));

vec3 v3Attenuate = exp(-fScatter \* (v3InvWavelength \* fKr4PI + fKm4PI));

v3FrontColor += v3Attenuate \* (fDepth \* fScaledLength);

v3SamplePoint += v3SampleRay;

}

// Finally, scale the Mie and Rayleigh colors and set up the varying variables for the pixel shader

gl\_FrontSecondaryColor.rgb = v3FrontColor \* fKmESun;

gl\_FrontColor.rgb = v3FrontColor \* (v3InvWavelength \* fKrESun);

gl\_Position = gl\_ModelViewProjectionMatrix \* gl\_Vertex;

v3Direction = v3CameraPos - v3Pos;

}

1. skyfromspace.fs

// Atmospheric scattering fragment shader

// Author: Sean O'Neil

// Copyright (c) 2004 Sean O'Neil

#version 120

uniform vec3 v3LightPos;

uniform float g;

uniform float g2;

varying vec3 v3Direction;

void main (void)

{

float fCos = dot(v3LightPos, v3Direction) / length(v3Direction);

float fRayleighPhase = 1.0 + fCos \* fCos;

float fMiePhase = (1.0 - g2) / (2.0 + g2) \* (1.0 + fCos \* fCos) / pow(1.0 + g2 - 2.0 \* g \* fCos, 1.5);

gl\_FragColor = vec4(1.0 - exp(-1.5 \* (fRayleighPhase \* gl\_Color.rgb + fMiePhase \* gl\_SecondaryColor.rgb)), 1.0);

}

/////////////////////////Source///////////////////////////

#include "opengl\_21\_tutorials\_win32\_framework.h"

// ----------------------------------------------------------------------------------------------------------------------------

CBuffer::CBuffer()

{

SetDefaults();

}

CBuffer::~CBuffer()

{

Empty();

}

void CBuffer::AddData(void \*Data, int DataSize)

{

int Remaining = BufferSize - Position;

if(DataSize > Remaining)

{

BYTE \*OldBuffer = Buffer;

int OldBufferSize = BufferSize;

int Needed = DataSize - Remaining;

BufferSize += Needed > BUFFER\_SIZE\_INCREMENT ? Needed : BUFFER\_SIZE\_INCREMENT;

Buffer = new BYTE[BufferSize];

memcpy(Buffer, OldBuffer, OldBufferSize);

delete [] OldBuffer;

}

memcpy(Buffer + Position, Data, DataSize);

Position += DataSize;

}

void CBuffer::Empty()

{

delete [] Buffer;

SetDefaults();

}

void \*CBuffer::GetData()

{

return Buffer;

}

int CBuffer::GetDataSize()

{

return Position;

}

void CBuffer::SetDefaults()

{

Buffer = NULL;

BufferSize = 0;

Position = 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int gl\_max\_texture\_size = 0, gl\_max\_texture\_max\_anisotropy\_ext = 0;

// ----------------------------------------------------------------------------------------------------------------------------

CTexture::CTexture()

{

Texture = 0;

}

CTexture::~CTexture()

{

}

CTexture::operator GLuint ()

{

return Texture;

}

bool CTexture::LoadTexture2D(char \*FileName)

{

CString DirectoryFileName = ModuleDirectory + FileName;

int Width, Height, BPP;

FIBITMAP \*dib = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(BPP == 8) Format = GL\_LUMINANCE;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileName);

FreeImage\_Unload(dib);

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_2D, Texture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_2D, GL\_GENERATE\_MIPMAP, GL\_TRUE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib));

glBindTexture(GL\_TEXTURE\_2D, 0);

FreeImage\_Unload(dib);

return true;

}

bool CTexture::LoadTextureCubeMap(char \*\*FileNames)

{

int Width, Height, BPP;

FIBITMAP \*dib[6];

bool Error = false;

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

{

CString DirectoryFileName = ModuleDirectory + FileNames[i];

dib[i] = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib[i] == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

Error = true;

}

}

if(Error)

{

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileNames[5]);

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, Texture);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_GENERATE\_MIPMAP, GL\_TRUE);

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

{

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib[i]));

}

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

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

{

FreeImage\_Unload(dib[i]);

}

return true;

}

void CTexture::Destroy()

{

glDeleteTextures(1, &Texture);

Texture = 0;

}

FIBITMAP \*CTexture::GetBitmap(char \*FileName, int &Width, int &Height, int &BPP)

{

FREE\_IMAGE\_FORMAT fif = FreeImage\_GetFileType(FileName);

if(fif == FIF\_UNKNOWN)

{

fif = FreeImage\_GetFIFFromFilename(FileName);

}

if(fif == FIF\_UNKNOWN)

{

return NULL;

}

FIBITMAP \*dib = NULL;

if(FreeImage\_FIFSupportsReading(fif))

{

dib = FreeImage\_Load(fif, FileName);

}

if(dib != NULL)

{

int OriginalWidth = FreeImage\_GetWidth(dib);

int OriginalHeight = FreeImage\_GetHeight(dib);

Width = OriginalWidth;

Height = OriginalHeight;

if(Width == 0 || Height == 0)

{

FreeImage\_Unload(dib);

return NULL;

}

BPP = FreeImage\_GetBPP(dib);

if(Width > gl\_max\_texture\_size) Width = gl\_max\_texture\_size;

if(Height > gl\_max\_texture\_size) Height = gl\_max\_texture\_size;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

Width = 1 << (int)floor((log((float)Width) / log(2.0f)) + 0.5f);

Height = 1 << (int)floor((log((float)Height) / log(2.0f)) + 0.5f);

}

if(Width != OriginalWidth || Height != OriginalHeight)

{

FIBITMAP \*rdib = FreeImage\_Rescale(dib, Width, Height, FILTER\_BICUBIC);

FreeImage\_Unload(dib);

dib = rdib;

}

}

return dib;

}

// ----------------------------------------------------------------------------------------------------------------------------

CShaderProgram::CShaderProgram()

{

SetDefaults();

}

CShaderProgram::~CShaderProgram()

{

}

CShaderProgram::operator GLuint ()

{

return Program;

}

bool CShaderProgram::Load(char \*VertexShaderFileName, char \*FragmentShaderFileName)

{

bool Error = false;

Destroy();

Error |= ((VertexShader = LoadShader(VertexShaderFileName, GL\_VERTEX\_SHADER)) == 0);

Error |= ((FragmentShader = LoadShader(FragmentShaderFileName, GL\_FRAGMENT\_SHADER)) == 0);

if(Error)

{

Destroy();

return false;

}

Program = glCreateProgram();

glAttachShader(Program, VertexShader);

glAttachShader(Program, FragmentShader);

glLinkProgram(Program);

int LinkStatus;

glGetProgramiv(Program, GL\_LINK\_STATUS, &LinkStatus);

if(LinkStatus == GL\_FALSE)

{

ErrorLog.Append("Error linking program (%s, %s)!\r\n", VertexShaderFileName, FragmentShaderFileName);

int InfoLogLength = 0;

glGetProgramiv(Program, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetProgramInfoLog(Program, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

Destroy();

return false;

}

return true;

}

void CShaderProgram::Destroy()

{

glDetachShader(Program, VertexShader);

glDetachShader(Program, FragmentShader);

glDeleteShader(VertexShader);

glDeleteShader(FragmentShader);

glDeleteProgram(Program);

delete [] UniformLocations;

delete [] AttribLocations;

SetDefaults();

}

GLuint CShaderProgram::LoadShader(char \*FileName, GLenum Type)

{

CString DirectoryFileName = ModuleDirectory + FileName;

FILE \*File;

if(fopen\_s(&File, DirectoryFileName, "rb") != 0)

{

ErrorLog.Append("Error loading file " + DirectoryFileName + "!\r\n");

return 0;

}

fseek(File, 0, SEEK\_END);

long Size = ftell(File);

fseek(File, 0, SEEK\_SET);

char \*Source = new char[Size + 1];

fread(Source, 1, Size, File);

fclose(File);

Source[Size] = 0;

GLuint Shader = glCreateShader(Type);

glShaderSource(Shader, 1, (const char\*\*)&Source, NULL);

delete [] Source;

glCompileShader(Shader);

int CompileStatus;

glGetShaderiv(Shader, GL\_COMPILE\_STATUS, &CompileStatus);

if(CompileStatus == GL\_FALSE)

{

ErrorLog.Append("Error compiling shader %s!\r\n", FileName);

int InfoLogLength = 0;

glGetShaderiv(Shader, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetShaderInfoLog(Shader, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

glDeleteShader(Shader);

return 0;

}

return Shader;

}

void CShaderProgram::SetDefaults()

{

VertexShader = 0;

FragmentShader = 0;

Program = 0;

UniformLocations = NULL;

AttribLocations = NULL;

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera::CCamera()

{

ViewMatrix = NULL;

ViewMatrixInverse = NULL;

X = vec3(1.0f, 0.0f, 0.0f);

Y = vec3(0.0f, 1.0f, 0.0f);

Z = vec3(0.0f, 0.0f, 1.0f);

Position = vec3(0.0f, 0.0f, 0.0f);

}

CCamera::~CCamera()

{

}

void CCamera::CalculateViewMatrix()

{

if(ViewMatrix != NULL)

{

\*ViewMatrix = mat4x4(X.x, Y.x, Z.x, 0.0f, X.y, Y.y, Z.y, 0.0f, X.z, Y.z, Z.z, 0.0f, -dot(X, Position), -dot(Y, Position), -dot(Z, Position), 1.0f);

if(ViewMatrixInverse != NULL)

{

\*ViewMatrixInverse = inverse(\*ViewMatrix);

}

}

}

void CCamera::Look(vec3 Position, vec3 Reference, float PlanetRadius)

{

this->Position = Position;

this->PlanetRadius = PlanetRadius;

Z = normalize(Position - Reference);

X = normalize(cross(vec3(0.0f, 1.0f, 0.0f), Z));

Y = cross(Z, X);

CalculateViewMatrix();

}

void CCamera::OnKeys(SHORT Keys, float FrameTime)

{

float Speed = max(1.0f, min(PlanetRadius / 2.0f, length(Position) - PlanetRadius));

if(Keys & 0x100) Speed \*= 2.0f;

if(Keys & 0x200) Speed \*= 0.5f;

float Distance = Speed \* FrameTime;

vec3 Up = Y \* Distance;

vec3 Right = X \* Distance;

vec3 Forward = -Z \* Distance;

vec3 Movement;

if(Keys & 0x001) Movement += Forward;

if(Keys & 0x002) Movement -= Forward;

if(Keys & 0x004) Movement -= Right;

if(Keys & 0x008) Movement += Right;

if(Keys & 0x010) Movement += Up;

if(Keys & 0x020) Movement -= Up;

if(Keys & 0x040)

{

X = rotate(X, 45.0f \* FrameTime, Z);

Y = rotate(Y, 45.0f \* FrameTime, Z);

}

if(Keys & 0x080)

{

X = rotate(X, - 45.0f \* FrameTime, Z);

Y = rotate(Y, - 45.0f \* FrameTime, Z);

}

Position += Movement;

if(length(Position) < PlanetRadius + 0.125f)

{

Position = normalize(Position) \* (PlanetRadius + 0.125f);

}

CalculateViewMatrix();

}

void CCamera::OnMouseMove(int dx, int dy)

{

float Sensitivity = 0.25f;

float DeltaX = (float)dx \* Sensitivity;

X = rotate(X, DeltaX, Y);

Z = rotate(Z, DeltaX, Y);

float DeltaY = (float)dy \* Sensitivity;

Y = rotate(Y, DeltaY, X);

Z = rotate(Z, DeltaY, X);

if((GetKeyState(VK\_LBUTTON) & 0x80))

{

Position = rotate(Position, DeltaX, Y);

Position = rotate(Position, DeltaY, X);

}

CalculateViewMatrix();

}

void CCamera::SetViewMatrixPointer(float \*ViewMatrix, float \*ViewMatrixInverse)

{

this->ViewMatrix = (mat4x4\*)ViewMatrix;

this->ViewMatrixInverse = (mat4x4\*)ViewMatrixInverse;

CalculateViewMatrix();

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera Camera;

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer::COpenGLRenderer()

{

Camera.SetViewMatrixPointer(&ViewMatrix);

}

COpenGLRenderer::~COpenGLRenderer()

{

}

bool COpenGLRenderer::Init()

{

bool Error = false;

Error |= !EarthMap.LoadTexture2D("earthmap.jpg");

Error |= !CloudsMap.LoadTexture2D("cloudsmap.jpg");

//Error |= !LightsMap.LoadTexture2D("lightsmap.jpg");

Error |= !gfs.Load("groundfromspace.vs", "groundfromspace.fs");

Error |= !gfa.Load("groundfromatmosphere.vs", "groundfromatmosphere.fs");

Error |= !sfs.Load("skyfromspace.vs", "skyfromspace.fs");

Error |= !sfa.Load("skyfromatmosphere.vs", "skyfromatmosphere.fs");

if(Error)

{

return false;

}

float Kr = 0.0030f;

float Km = 0.0015f;

float ESun = 16.0f;

float g = -0.75f;

InnerRadius = 10.0f \* 25.0f;

OuterRadius = 10.25f \* 25.0f;

float Scale = 1.0f / (OuterRadius - InnerRadius);

float ScaleDepth = 0.25f;

float ScaleOverScaleDepth = Scale / ScaleDepth;

GLuint programs[] = {gfs, gfa, sfs, sfa};

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

{

glUseProgram(programs[i]);

glUniform3f(glGetUniformLocation(programs[i], "v3LightPos"), 0.0f, 0.0f, 1.0f);

glUniform3f(glGetUniformLocation(programs[i], "v3InvWavelength"), 1.0f / powf(0.650f, 4.0f), 1.0f / powf(0.570f, 4.0f), 1.0f / powf(0.475f, 4.0f));

glUniform1f(glGetUniformLocation(programs[i], "fInnerRadius"), InnerRadius);

glUniform1f(glGetUniformLocation(programs[i], "fInnerRadius2"), InnerRadius \* InnerRadius);

glUniform1f(glGetUniformLocation(programs[i], "fOuterRadius"), OuterRadius);

glUniform1f(glGetUniformLocation(programs[i], "fOuterRadius2"), OuterRadius \* OuterRadius);

glUniform1f(glGetUniformLocation(programs[i], "fKrESun"), Kr \* ESun);

glUniform1f(glGetUniformLocation(programs[i], "fKmESun"), Km \* ESun);

glUniform1f(glGetUniformLocation(programs[i], "fKr4PI"), Kr \* 4.0f \* (float)M\_PI);

glUniform1f(glGetUniformLocation(programs[i], "fKm4PI"), Km \* 4.0f \* (float)M\_PI);

glUniform1f(glGetUniformLocation(programs[i], "fScale"), Scale);

glUniform1f(glGetUniformLocation(programs[i], "fScaleDepth"), ScaleDepth);

glUniform1f(glGetUniformLocation(programs[i], "fScaleOverScaleDepth"), ScaleOverScaleDepth);

glUniform1f(glGetUniformLocation(programs[i], "g"), g);

glUniform1f(glGetUniformLocation(programs[i], "g2"), g \* g);

glUniform1i(glGetUniformLocation(programs[i], "Samples"), 4);

glUniform1i(glGetUniformLocation(programs[i], "s2Tex1"), 0);

glUniform1i(glGetUniformLocation(programs[i], "s2Tex2"), 1);

//glUniform1i(glGetUniformLocation(programs[i], "s2Tex3"), 2);

glUseProgram(0);

}

int X = 128, Y = X / 2, vpos = 0, tpos = 0;

float a, stepa = (float)M\_PI \* 2.0f / (float)X, stepb = (float)M\_PI / (float)Y, b = -(float)M\_PI\_2 + stepb;

vec3 \*vertices = new vec3[X \* (Y - 1)];

for(int y = 0; y < (Y - 1); y++)

{

a = -(float)M\_PI;

for(int x = 0; x < X; x++)

{

vertices[y \* X + x] = normalize(vec3(sin(a) \* cos(b), sin(b), cos(a) \* cos(b)));

a += stepa;

}

b += stepb;

}

VerticesCount = (X \* (Y - 2) \* 2 + X \* 2) \* 3;

vec3 \*Vertices = new vec3[VerticesCount];

vec2 \*TexCoords = new vec2[VerticesCount];

for(int x = 0; x < X; x++)

{

Vertices[vpos++] = vec3(0.0f, -1.0f, 0.0f);

Vertices[vpos++] = vertices[(0 + 0) \* X + ((x + 1) % X)];

Vertices[vpos++] = vertices[(0 + 0) \* X + ((x + 0) % X)];

TexCoords[tpos++] = vec2((float)(x + 0.5f) / (float)X, 0.0f);

TexCoords[tpos++] = vec2((float)(x + 1) / (float)X, (float)(0 + 1) / (float)Y);

TexCoords[tpos++] = vec2((float)(x + 0) / (float)X, (float)(0 + 1) / (float)Y);

}

for(int y = 0; y < Y - 2; y++)

{

for(int x = 0; x < X; x++)

{

Vertices[vpos++] = vertices[(y + 0) \* X + ((x + 0) % X)];

Vertices[vpos++] = vertices[(y + 0) \* X + ((x + 1) % X)];

Vertices[vpos++] = vertices[(y + 1) \* X + ((x + 1) % X)];

TexCoords[tpos++] = vec2((float)(x + 0) / (float)X, (float)(1 + y + 0) / (float)Y);

TexCoords[tpos++] = vec2((float)(x + 1) / (float)X, (float)(1 + y + 0) / (float)Y);

TexCoords[tpos++] = vec2((float)(x + 1) / (float)X, (float)(1 + y + 1) / (float)Y);

Vertices[vpos++] = vertices[(y + 1) \* X + ((x + 1) % X)];

Vertices[vpos++] = vertices[(y + 1) \* X + ((x + 0) % X)];

Vertices[vpos++] = vertices[(y + 0) \* X + ((x + 0) % X)];

TexCoords[tpos++] = vec2((float)(x + 1) / (float)X, float(1 + y + 1) / (float)Y);

TexCoords[tpos++] = vec2((float)(x + 0) / (float)X, float(1 + y + 1) / (float)Y);

TexCoords[tpos++] = vec2((float)(x + 0) / (float)X, float(1 + y + 0) / (float)Y);

}

}

for(int x = 0; x < X; x++)

{

Vertices[vpos++] = vertices[(Y - 2) \* X + ((x + 0) % X)];

Vertices[vpos++] = vertices[(Y - 2) \* X + ((x + 1) % X)];

Vertices[vpos++] = vec3(0.0f, 1.0f, 0.0f);

TexCoords[tpos++] = vec2((float)(x + 0) / (float)X, (float)(Y - 1) / (float)Y);

TexCoords[tpos++] = vec2((float)(x + 1) / (float)X, (float)(Y - 1) / (float)Y);

TexCoords[tpos++] = vec2((float)(x + 0.5f) / (float)X, 1.0f);

}

glGenBuffers(1, &VerticesVBO);

glBindBuffer(GL\_ARRAY\_BUFFER, VerticesVBO);

glBufferData(GL\_ARRAY\_BUFFER, VerticesCount \* 3 \* 4, Vertices, GL\_STATIC\_DRAW);

glGenBuffers(1, &TexCoordsVBO);

glBindBuffer(GL\_ARRAY\_BUFFER, TexCoordsVBO);

glBufferData(GL\_ARRAY\_BUFFER, VerticesCount \* 2 \* 4, TexCoords, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

delete [] vertices;

delete [] Vertices;

delete [] TexCoords;

Camera.Look(vec3(0.0f, 0.0f, 768.0f), vec3(0.0f, 0.0f, 0.0f), InnerRadius);

return true;

}

void COpenGLRenderer::Render(float FrameTime)

{

static float a = 0.0f;

float ainc = 0.25f \* FrameTime;

a += ainc;

if(length(Camera.Position) < OuterRadius \* 1.1f)

{

vec3 Y = vec3(0.0f, 1.0f, 0.0f);

Camera.Position = rotate(Camera.Position, ainc, Y);

Camera.X = rotate(Camera.X, ainc, Y);

Camera.Y = rotate(Camera.Y, ainc, Y);

Camera.Z = rotate(Camera.Z, ainc, Y);

Camera.CalculateViewMatrix();

}

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

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

GLuint program;

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glScalef(InnerRadius, InnerRadius, InnerRadius);

glRotatef(a, 0.0f, 1.0f, 0.0f);

glMatrixMode(GL\_TEXTURE);

glLoadIdentity();

glScalef(InnerRadius, InnerRadius, InnerRadius);

glRotatef(a, 0.0f, 1.0f, 0.0f);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, EarthMap);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, CloudsMap);

//glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_2D, LightsMap);

if(length(Camera.Position) > OuterRadius) program = gfs;

else program = gfa;

glUseProgram(program);

glUniform3fv(glGetUniformLocation(program, "v3CameraPos"), 1, &Camera.Position);

glUniform1f(glGetUniformLocation(program, "fCameraHeight"), length(Camera.Position));

glUniform1f(glGetUniformLocation(program, "fCameraHeight2"), length(Camera.Position) \* length(Camera.Position));

glEnableClientState(GL\_VERTEX\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, VerticesVBO);

glVertexPointer(3, GL\_FLOAT, 0, NULL);

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, TexCoordsVBO);

glTexCoordPointer(2, GL\_FLOAT, 0, NULL);

glDrawArrays(GL\_TRIANGLES, 0, VerticesCount);

glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glDisableClientState(GL\_VERTEX\_ARRAY);

glUseProgram(0);

//glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, 0);

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

glScalef(OuterRadius, OuterRadius, OuterRadius);

glMatrixMode(GL\_TEXTURE);

glLoadIdentity();

glScalef(OuterRadius, OuterRadius, OuterRadius);

if(length(Camera.Position) > OuterRadius) program = sfs;

else program = sfa;

//glBlendFunc(GL\_ONE, GL\_ONE\_MINUS\_SRC\_ALPHA);

//glEnable(GL\_BLEND);

glCullFace(GL\_FRONT);

glUseProgram(program);

glUniform3fv(glGetUniformLocation(program, "v3CameraPos"), 1, &Camera.Position);

glUniform1f(glGetUniformLocation(program, "fCameraHeight"), length(Camera.Position));

glUniform1f(glGetUniformLocation(program, "fCameraHeight2"), length(Camera.Position) \* length(Camera.Position));

glEnableClientState(GL\_VERTEX\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, VerticesVBO);

glVertexPointer(3, GL\_FLOAT, 0, NULL);

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, TexCoordsVBO);

glTexCoordPointer(2, GL\_FLOAT, 0, NULL);

glDrawArrays(GL\_TRIANGLES, 0, VerticesCount);

glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glDisableClientState(GL\_VERTEX\_ARRAY);

glUseProgram(0);

glCullFace(GL\_BACK);

glDisable(GL\_CULL\_FACE);

glDisable(GL\_DEPTH\_TEST);

}

void COpenGLRenderer::Resize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

glViewport(0, 0, Width, Height);

ProjectionMatrix = perspective(45.0f, (float)Width / (float)Height, 0.125f, 4096.0f);

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&ProjectionMatrix);

}

void COpenGLRenderer::Destroy()

{

EarthMap.Destroy();

CloudsMap.Destroy();

//LightsMap.Destroy();

glDeleteBuffers(1, &VerticesVBO);

glDeleteBuffers(1, &TexCoordsVBO);

gfs.Destroy();

gfa.Destroy();

sfs.Destroy();

sfa.Destroy();

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer OpenGLRenderer;

// ----------------------------------------------------------------------------------------------------------------------------

CString ModuleDirectory, ErrorLog;

// ----------------------------------------------------------------------------------------------------------------------------

void GetModuleDirectory()

{

char \*moduledirectory = new char[256];

GetModuleFileName(GetModuleHandle(NULL), moduledirectory, 256);

\*(strrchr(moduledirectory, '\\') + 1) = 0;

ModuleDirectory = moduledirectory;

delete [] moduledirectory;

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView::COpenGLView()

{

}

COpenGLView::~COpenGLView()

{

}

bool COpenGLView::Init(HINSTANCE hInstance, char \*Title, int Width, int Height, int Samples)

{

this->Title = Title;

this->Width = Width;

this->Height = Height;

WNDCLASSEX WndClassEx;

memset(&WndClassEx, 0, sizeof(WNDCLASSEX));

WndClassEx.cbSize = sizeof(WNDCLASSEX);

WndClassEx.style = CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

WndClassEx.lpfnWndProc = WndProc;

WndClassEx.hInstance = hInstance;

WndClassEx.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hCursor = LoadCursor(NULL, IDC\_ARROW);

WndClassEx.lpszClassName = "Win32OpenGLWindowClass";

if(RegisterClassEx(&WndClassEx) == 0)

{

ErrorLog.Set("RegisterClassEx failed!");

return false;

}

DWORD Style = WS\_OVERLAPPEDWINDOW | WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN;

hWnd = CreateWindowEx(WS\_EX\_APPWINDOW, WndClassEx.lpszClassName, Title, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL);

if(hWnd == NULL)

{

ErrorLog.Set("CreateWindowEx failed!");

return false;

}

HDC hDC = GetDC(hWnd);

if(hDC == NULL)

{

ErrorLog.Set("GetDC failed!");

return false;

}

PIXELFORMATDESCRIPTOR pfd;

memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));

pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);

pfd.nVersion = 1;

pfd.dwFlags = PFD\_DRAW\_TO\_WINDOW | PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER;

pfd.iPixelType = PFD\_TYPE\_RGBA;

pfd.cColorBits = 32;

pfd.cDepthBits = 24;

pfd.iLayerType = PFD\_MAIN\_PLANE;

int PixelFormat = ChoosePixelFormat(hDC, &pfd);

if(PixelFormat == 0)

{

ErrorLog.Set("ChoosePixelFormat failed!");

return false;

}

static int MSAAPixelFormat = 0;

if(SetPixelFormat(hDC, MSAAPixelFormat == 0 ? PixelFormat : MSAAPixelFormat, &pfd) == FALSE)

{

ErrorLog.Set("SetPixelFormat failed!");

return false;

}

hGLRC = wglCreateContext(hDC);

if(hGLRC == NULL)

{

ErrorLog.Set("wglCreateContext failed!");

return false;

}

if(wglMakeCurrent(hDC, hGLRC) == FALSE)

{

ErrorLog.Set("wglMakeCurrent failed!");

return false;

}

if(glewInit() != GLEW\_OK)

{

ErrorLog.Set("glewInit failed!");

return false;

}

if(!GLEW\_VERSION\_2\_1)

{

ErrorLog.Set("OpenGL 2.1 not supported!");

return false;

}

if(MSAAPixelFormat == 0 && Samples > 0)

{

if(GLEW\_ARB\_multisample && WGLEW\_ARB\_pixel\_format)

{

while(Samples > 0)

{

UINT NumFormats = 0;

int PFAttribs[] =

{

WGL\_DRAW\_TO\_WINDOW\_ARB, GL\_TRUE,

WGL\_SUPPORT\_OPENGL\_ARB, GL\_TRUE,

WGL\_DOUBLE\_BUFFER\_ARB, GL\_TRUE,

WGL\_PIXEL\_TYPE\_ARB, WGL\_TYPE\_RGBA\_ARB,

WGL\_COLOR\_BITS\_ARB, 32,

WGL\_DEPTH\_BITS\_ARB, 24,

WGL\_ACCELERATION\_ARB, WGL\_FULL\_ACCELERATION\_ARB,

WGL\_SAMPLE\_BUFFERS\_ARB, GL\_TRUE,

WGL\_SAMPLES\_ARB, Samples,

0

};

if(wglChoosePixelFormatARB(hDC, PFAttribs, NULL, 1, &MSAAPixelFormat, &NumFormats) == TRUE && NumFormats > 0) break;

Samples--;

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

UnregisterClass(WndClassEx.lpszClassName, hInstance);

return Init(hInstance, Title, Width, Height, Samples);

}

else

{

Samples = 0;

}

}

this->Samples = Samples;

GetModuleDirectory();

glGetIntegerv(GL\_MAX\_TEXTURE\_SIZE, &gl\_max\_texture\_size);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glGetIntegerv(GL\_MAX\_TEXTURE\_MAX\_ANISOTROPY\_EXT, &gl\_max\_texture\_max\_anisotropy\_ext);

}

if(WGLEW\_EXT\_swap\_control)

{

wglSwapIntervalEXT(0);

}

return OpenGLRenderer.Init();

}

void COpenGLView::Show(bool Maximized)

{

RECT dRect, wRect, cRect;

GetWindowRect(GetDesktopWindow(), &dRect);

GetWindowRect(hWnd, &wRect);

GetClientRect(hWnd, &cRect);

wRect.right += Width - cRect.right;

wRect.bottom += Height - cRect.bottom;

wRect.right -= wRect.left;

wRect.bottom -= wRect.top;

wRect.left = dRect.right / 2 - wRect.right / 2;

wRect.top = dRect.bottom / 2 - wRect.bottom / 2;

MoveWindow(hWnd, wRect.left, wRect.top, wRect.right, wRect.bottom, FALSE);

ShowWindow(hWnd, Maximized ? SW\_SHOWMAXIMIZED : SW\_SHOWNORMAL);

}

void COpenGLView::MessageLoop()

{

MSG Msg;

while(GetMessage(&Msg, NULL, 0, 0) > 0)

{

TranslateMessage(&Msg);

DispatchMessage(&Msg);

}

}

void COpenGLView::Destroy()

{

if(GLEW\_VERSION\_2\_1)

{

OpenGLRenderer.Destroy();

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

}

void COpenGLView::OnKeyDown(UINT Key)

{

/\*switch(Key)

{

case VK\_F1:

break;

case VK\_SPACE:

break;

}\*/

}

void COpenGLView::OnLButtonDown(int X, int Y)

{

LastX = X;

LastY = Y;

}

void COpenGLView::OnMouseMove(int X, int Y)

{

if((GetKeyState(VK\_LBUTTON) & 0x80) || GetKeyState(VK\_RBUTTON) & 0x80)

{

Camera.OnMouseMove(LastX - X, LastY - Y);

LastX = X;

LastY = Y;

}

}

void COpenGLView::OnMouseWheel(short zDelta)

{

}

void COpenGLView::OnPaint()

{

static DWORD LastFPSTime = GetTickCount(), LastFrameTime = LastFPSTime, FPS = 0;

PAINTSTRUCT ps;

HDC hDC = BeginPaint(hWnd, &ps);

DWORD Time = GetTickCount();

float FrameTime = (Time - LastFrameTime) \* 0.001f;

LastFrameTime = Time;

if(Time - LastFPSTime > 1000)

{

CString Text = Title;

if(OpenGLRenderer.Text[0] != 0)

{

Text.Append(" - " + OpenGLRenderer.Text);

}

Text.Append(" - %dx%d", Width, Height);

Text.Append(", ATF %dx", gl\_max\_texture\_max\_anisotropy\_ext);

Text.Append(", MSAA %dx", Samples);

Text.Append(", FPS: %d", FPS);

Text.Append(" - %s", glGetString(GL\_RENDERER));

SetWindowText(hWnd, Text);

LastFPSTime = Time;

FPS = 0;

}

else

{

FPS++;

}

SHORT Keys = 0x000;

if(GetKeyState('W') & 0x80) Keys |= 0x001;

if(GetKeyState('S') & 0x80) Keys |= 0x002;

if(GetKeyState('A') & 0x80) Keys |= 0x004;

if(GetKeyState('D') & 0x80) Keys |= 0x008;

if(GetKeyState('R') & 0x80) Keys |= 0x010;

if(GetKeyState('F') & 0x80) Keys |= 0x020;

if(GetKeyState('Q') & 0x80) Keys |= 0x040;

if(GetKeyState('E') & 0x80) Keys |= 0x080;

if(GetKeyState(VK\_SHIFT) & 0x80) Keys |= 0x100;

if(GetKeyState(VK\_CONTROL) & 0x80) Keys |= 0x200;

if(Keys & 0xFF)

{

Camera.OnKeys(Keys, FrameTime);

}

OpenGLRenderer.Render(FrameTime);

SwapBuffers(hDC);

EndPaint(hWnd, &ps);

InvalidateRect(hWnd, NULL, FALSE);

}

void COpenGLView::OnRButtonDown(int X, int Y)

{

LastX = X;

LastY = Y;

}

void COpenGLView::OnSize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

OpenGLRenderer.Resize(Width, Height);

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView OpenGLView;

// ----------------------------------------------------------------------------------------------------------------------------

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)

{

switch(uiMsg)

{

case WM\_CLOSE:

PostQuitMessage(0);

break;

case WM\_KEYDOWN:

OpenGLView.OnKeyDown((UINT)wParam);

break;

case WM\_LBUTTONDOWN:

OpenGLView.OnLButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_MOUSEMOVE:

OpenGLView.OnMouseMove(LOWORD(lParam), HIWORD(lParam));

break;

case 0x020A: // WM\_MOUSWHEEL

OpenGLView.OnMouseWheel(HIWORD(wParam));

break;

case WM\_PAINT:

OpenGLView.OnPaint();

break;

case WM\_RBUTTONDOWN:

OpenGLView.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_SIZE:

OpenGLView.OnSize(LOWORD(lParam), HIWORD(lParam));

break;

default:

return DefWindowProc(hWnd, uiMsg, wParam, lParam);

}

return 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine, int iShow)

{

char \*AppName = "Planet Earth from space";

if(OpenGLView.Init(hInstance, AppName, 800, 600, 4))

{

OpenGLView.Show();

OpenGLView.MessageLoop();

}

else

{

MessageBox(NULL, ErrorLog, AppName, MB\_OK | MB\_ICONERROR);

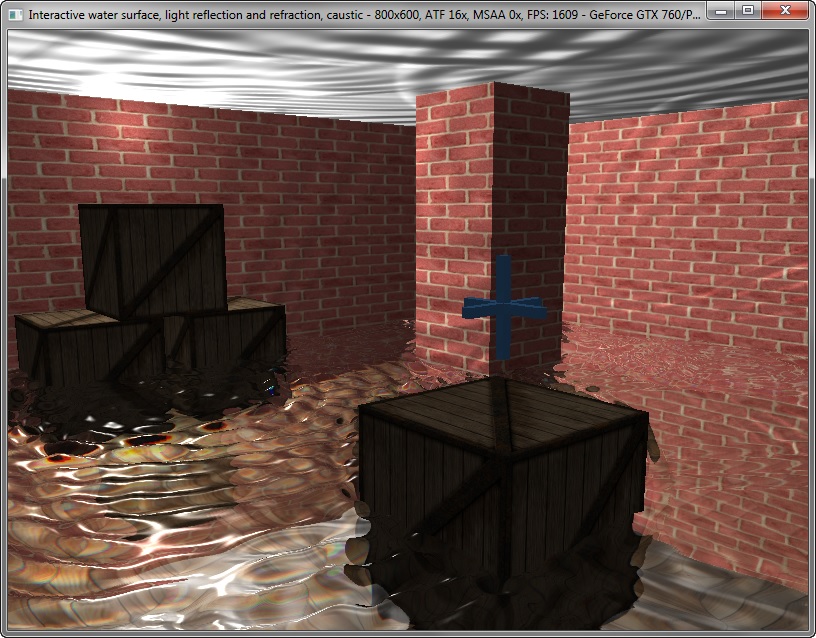
}

OpenGLView.Destroy();

return 0;

}

///////////////////////////////////////////////////////////////Water/////////////////////////////////////////////



VS:1>scene.vs

#version 120

uniform mat3x3 NormalMatrix;

uniform mat4x4 ModelMatrix;

varying vec3 Position, Normal;

void main()

{

Position = (ModelMatrix \* gl\_Vertex).xyz;

Normal = NormalMatrix \* gl\_Normal;

gl\_FrontColor = gl\_Color;

gl\_TexCoord[0] = gl\_MultiTexCoord0;

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

}

2>scene.fs

#version 120

uniform int ClipType, Texturing;

uniform float WaterLevel;

uniform sampler2D Texture;

varying vec3 Position, Normal;

void main()

{

if(ClipType == 1) if(Position.y < WaterLevel) discard;

if(ClipType == 2) if(Position.y > WaterLevel) discard;

vec3 LightDirection = gl\_LightSource[0].position.xyz - Position;

float LightDistance2 = dot(LightDirection, LightDirection);

float LightDistance = sqrt(LightDistance2);

LightDirection /= LightDistance;

float NdotLD = max(0.0, dot(Normal, LightDirection));

float att = gl\_LightSource[0].constantAttenuation;

att += gl\_LightSource[0].linearAttenuation \* LightDistance;

att += gl\_LightSource[0].quadraticAttenuation \* LightDistance2;

vec3 Light = (gl\_LightSource[0].ambient.rgb + gl\_LightSource[0].diffuse.rgb \* NdotLD) / att;

gl\_FragColor = gl\_Color;

if(Texturing == 1) gl\_FragColor.rgb \*= texture2D(Texture, gl\_TexCoord[0].st).rgb;

gl\_FragColor.rgb \*= Light;

}

3>water.vs

#version 120

#define MAX\_WAVES 16

struct CWave

{

float StartTime, Speed, MaxY, FrequencyMPIM2;

vec2 Position;

};

uniform float Time, WaterLevel, WMSDWMR, WMSDWMRM2;

uniform mat3x3 NormalMatrix;

uniform CWave Waves[MAX\_WAVES];

varying vec3 Position, Normal;

void main()

{

vec3 Vertices[5];

Vertices[0] = vec3(gl\_Vertex.x + WMSDWMR, WaterLevel, gl\_Vertex.y);

Vertices[1] = vec3(gl\_Vertex.x, WaterLevel, gl\_Vertex.y - WMSDWMR);

Vertices[2] = vec3(gl\_Vertex.x - WMSDWMR, WaterLevel, gl\_Vertex.y);

Vertices[3] = vec3(gl\_Vertex.x, WaterLevel, gl\_Vertex.y + WMSDWMR);

Vertices[4] = vec3(gl\_Vertex.x, WaterLevel, gl\_Vertex.y);

for(int wi = 0; wi < MAX\_WAVES; wi++)

{

for(int vi = 0; vi < 5; vi++)

{

float d = distance(Waves[wi].Position, Vertices[vi].xz);

float t = Time - Waves[wi].StartTime - d / Waves[wi].Speed;

if(t > 0.0)

{

float maxy = Waves[wi].MaxY - Waves[wi].MaxY \* t;

if(maxy > 0.0)

{

Vertices[vi].y -= sin(t \* Waves[wi].FrequencyMPIM2) \* maxy / (1.0 + d);

}

}

}

}

Position = Vertices[4];

Normal = NormalMatrix \* vec3(Vertices[2].y - Vertices[0].y, WMSDWMRM2, Vertices[1].y - Vertices[3].y);

gl\_TexCoord[0] = gl\_ModelViewProjectionMatrix \* vec4(Position, 1.0);

gl\_Position = gl\_TexCoord[0];

}

4>water.fs

#version 120

uniform sampler2D ReflectionTexture, RefractionTexture;

uniform float WaterLevel;

uniform vec3 CameraPosition;

varying vec3 Position, Normal;

void main()

{

vec2 TexCoord = gl\_TexCoord[0].st / gl\_TexCoord[0].w \* 0.5 + 0.5;

vec3 NormalizedNormal = normalize(Normal);

vec2 Offset = NormalizedNormal.xz \* vec2(0.05, -0.05);

if(CameraPosition.y > WaterLevel)

{

vec3 Reflection = texture2D(ReflectionTexture, TexCoord + Offset).rgb;

vec3 Refraction;

Refraction.r = texture2D(RefractionTexture, TexCoord - Offset \* 0.8).r;

Refraction.g = texture2D(RefractionTexture, TexCoord - Offset \* 0.9).g;

Refraction.b = texture2D(RefractionTexture, TexCoord - Offset \* 1.0).b;

vec3 LightDirection = normalize(Position - gl\_LightSource[0].position.xyz);

vec3 LightDirectionReflected = reflect(LightDirection, NormalizedNormal);

vec3 CameraDirection = normalize(CameraPosition - Position);

float CDdotLDR = max(dot(CameraDirection, LightDirectionReflected), 0.0);

vec3 Specular = gl\_LightSource[0].specular.rgb \* pow(CDdotLDR, 128.0);

float NdotCD = max(dot(NormalizedNormal, CameraDirection), 0.0);

gl\_FragColor.rgb = mix(Reflection, Refraction, NdotCD) + Specular;

}

else

{

gl\_FragColor.r = texture2D(RefractionTexture, TexCoord - Offset \* 0.8).r;

gl\_FragColor.g = texture2D(RefractionTexture, TexCoord - Offset \* 0.9).g;

gl\_FragColor.b = texture2D(RefractionTexture, TexCoord - Offset \* 1.0).b;

}

}

////////////////////////////Source/////////////////////////////

/ ----------------------------------------------------------------------------------------------------------------------------

#define MAX\_WAVES 16

class CWave

{

public:

float StartTime, Speed, MaxY, FrequencyMPIM2;

vec2 Position;

public:

CWave();

~CWave();

};

// ----------------------------------------------------------------------------------------------------------------------------

#define WMS 10.0f // water mesh size

#define WMR 256 // water mesh resolution

#define MAXY 0.0625f // waves amplitude

class COpenGLRenderer

{

protected:

int Width, Height;

mat3x3 NormalMatrix;

mat4x4 ModelMatrix, ViewMatrix, ViewMatrixInverse, ProjectionMatrix, ProjectionBiasMatrixInverse;

protected:

CTexture Texture[3];

CShaderProgram Scene, Water;

GLuint TexCoordsVBO, NormalsVBO, VerticesVBO, WaterVerticesVBO;

GLuint ReflectionTexture, RefractionTexture, DepthTexture, FBO;

float WaterLevel;

CWave Waves[MAX\_WAVES];

int Wave, WaterVerticesCount;

public:

bool Pause, WireFrame;

vec3 LightColor, LightPosition;

public:

CString Text;

public:

COpenGLRenderer();

~COpenGLRenderer();

bool Init();

void Render(float FrameTime);

void Resize(int Width, int Height);

void Destroy();

void RenderScene(int ClipType);

void AddWaveByMouseClick(int x, int y);

void SetWaveInShaders(int Wave);

void IncreaseWaterLevel(float Increment);

void InitWaterVertexBuffer();

void InitSceneVertexBuffers();

};

////////////////CPP/////////////////////

#include "opengl\_21\_tutorials\_win32\_framework.h"

// ----------------------------------------------------------------------------------------------------------------------------

CBuffer::CBuffer()

{

SetDefaults();

}

CBuffer::~CBuffer()

{

Empty();

}

void CBuffer::AddData(void \*Data, int DataSize)

{

int Remaining = BufferSize - Position;

if(DataSize > Remaining)

{

BYTE \*OldBuffer = Buffer;

int OldBufferSize = BufferSize;

int Needed = DataSize - Remaining;

BufferSize += Needed > BUFFER\_SIZE\_INCREMENT ? Needed : BUFFER\_SIZE\_INCREMENT;

Buffer = new BYTE[BufferSize];

memcpy(Buffer, OldBuffer, OldBufferSize);

delete [] OldBuffer;

}

memcpy(Buffer + Position, Data, DataSize);

Position += DataSize;

}

void CBuffer::Empty()

{

delete [] Buffer;

SetDefaults();

}

void \*CBuffer::GetData()

{

return Buffer;

}

int CBuffer::GetDataSize()

{

return Position;

}

void CBuffer::SetDefaults()

{

Buffer = NULL;

BufferSize = 0;

Position = 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int gl\_max\_texture\_size = 0, gl\_max\_texture\_max\_anisotropy\_ext = 0;

// ----------------------------------------------------------------------------------------------------------------------------

CTexture::CTexture()

{

Texture = 0;

}

CTexture::~CTexture()

{

}

CTexture::operator GLuint ()

{

return Texture;

}

bool CTexture::LoadTexture2D(char \*FileName)

{

CString DirectoryFileName = ModuleDirectory + FileName;

int Width, Height, BPP;

FIBITMAP \*dib = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileName);

FreeImage\_Unload(dib);

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_2D, Texture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_2D, GL\_GENERATE\_MIPMAP, GL\_TRUE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib));

glBindTexture(GL\_TEXTURE\_2D, 0);

FreeImage\_Unload(dib);

return true;

}

bool CTexture::LoadTextureCubeMap(char \*\*FileNames)

{

int Width, Height, BPP;

FIBITMAP \*dib[6];

bool Error = false;

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

{

CString DirectoryFileName = ModuleDirectory + FileNames[i];

dib[i] = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib[i] == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

Error = true;

}

}

if(Error)

{

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileNames[5]);

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, Texture);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_GENERATE\_MIPMAP, GL\_TRUE);

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

{

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib[i]));

}

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

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

{

FreeImage\_Unload(dib[i]);

}

return true;

}

void CTexture::Destroy()

{

glDeleteTextures(1, &Texture);

Texture = 0;

}

FIBITMAP \*CTexture::GetBitmap(char \*FileName, int &Width, int &Height, int &BPP)

{

FREE\_IMAGE\_FORMAT fif = FreeImage\_GetFileType(FileName);

if(fif == FIF\_UNKNOWN)

{

fif = FreeImage\_GetFIFFromFilename(FileName);

}

if(fif == FIF\_UNKNOWN)

{

return NULL;

}

FIBITMAP \*dib = NULL;

if(FreeImage\_FIFSupportsReading(fif))

{

dib = FreeImage\_Load(fif, FileName);

}

if(dib != NULL)

{

int OriginalWidth = FreeImage\_GetWidth(dib);

int OriginalHeight = FreeImage\_GetHeight(dib);

Width = OriginalWidth;

Height = OriginalHeight;

if(Width == 0 || Height == 0)

{

FreeImage\_Unload(dib);

return NULL;

}

BPP = FreeImage\_GetBPP(dib);

if(Width > gl\_max\_texture\_size) Width = gl\_max\_texture\_size;

if(Height > gl\_max\_texture\_size) Height = gl\_max\_texture\_size;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

Width = 1 << (int)floor((log((float)Width) / log(2.0f)) + 0.5f);

Height = 1 << (int)floor((log((float)Height) / log(2.0f)) + 0.5f);

}

if(Width != OriginalWidth || Height != OriginalHeight)

{

FIBITMAP \*rdib = FreeImage\_Rescale(dib, Width, Height, FILTER\_BICUBIC);

FreeImage\_Unload(dib);

dib = rdib;

}

}

return dib;

}

// ----------------------------------------------------------------------------------------------------------------------------

CShaderProgram::CShaderProgram()

{

SetDefaults();

}

CShaderProgram::~CShaderProgram()

{

}

CShaderProgram::operator GLuint ()

{

return Program;

}

bool CShaderProgram::Load(char \*VertexShaderFileName, char \*FragmentShaderFileName)

{

bool Error = false;

Destroy();

Error |= ((VertexShader = LoadShader(VertexShaderFileName, GL\_VERTEX\_SHADER)) == 0);

Error |= ((FragmentShader = LoadShader(FragmentShaderFileName, GL\_FRAGMENT\_SHADER)) == 0);

if(Error)

{

Destroy();

return false;

}

Program = glCreateProgram();

glAttachShader(Program, VertexShader);

glAttachShader(Program, FragmentShader);

glLinkProgram(Program);

int LinkStatus;

glGetProgramiv(Program, GL\_LINK\_STATUS, &LinkStatus);

if(LinkStatus == GL\_FALSE)

{

ErrorLog.Append("Error linking program (%s, %s)!\r\n", VertexShaderFileName, FragmentShaderFileName);

int InfoLogLength = 0;

glGetProgramiv(Program, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetProgramInfoLog(Program, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

Destroy();

return false;

}

return true;

}

void CShaderProgram::Destroy()

{

glDetachShader(Program, VertexShader);

glDetachShader(Program, FragmentShader);

glDeleteShader(VertexShader);

glDeleteShader(FragmentShader);

glDeleteProgram(Program);

delete [] UniformLocations;

delete [] AttribLocations;

SetDefaults();

}

GLuint CShaderProgram::LoadShader(char \*FileName, GLenum Type)

{

CString DirectoryFileName = ModuleDirectory + FileName;

FILE \*File;

if(fopen\_s(&File, DirectoryFileName, "rb") != 0)

{

ErrorLog.Append("Error loading file " + DirectoryFileName + "!\r\n");

return 0;

}

fseek(File, 0, SEEK\_END);

long Size = ftell(File);

fseek(File, 0, SEEK\_SET);

char \*Source = new char[Size + 1];

fread(Source, 1, Size, File);

fclose(File);

Source[Size] = 0;

GLuint Shader = glCreateShader(Type);

glShaderSource(Shader, 1, (const char\*\*)&Source, NULL);

delete [] Source;

glCompileShader(Shader);

int CompileStatus;

glGetShaderiv(Shader, GL\_COMPILE\_STATUS, &CompileStatus);

if(CompileStatus == GL\_FALSE)

{

ErrorLog.Append("Error compiling shader %s!\r\n", FileName);

int InfoLogLength = 0;

glGetShaderiv(Shader, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetShaderInfoLog(Shader, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

glDeleteShader(Shader);

return 0;

}

return Shader;

}

void CShaderProgram::SetDefaults()

{

VertexShader = 0;

FragmentShader = 0;

Program = 0;

UniformLocations = NULL;

AttribLocations = NULL;

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera::CCamera()

{

ViewMatrix = NULL;

ViewMatrixInverse = NULL;

X = vec3(1.0f, 0.0f, 0.0f);

Y = vec3(0.0f, 1.0f, 0.0f);

Z = vec3(0.0f, 0.0f, 1.0f);

Position = vec3(0.0f, 0.0f, 5.0f);

Reference = vec3(0.0f, 0.0f, 0.0f);

}

CCamera::~CCamera()

{

}

void CCamera::Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference)

{

this->Position = Position;

this->Reference = Reference;

Z = normalize(Position - Reference);

X = normalize(cross(vec3(0.0f, 1.0f, 0.0f), Z));

Y = cross(Z, X);

if(!RotateAroundReference)

{

this->Reference = this->Position;

this->Position += Z \* 0.05f;

}

CalculateViewMatrix();

}

void CCamera::Move(const vec3 &Movement)

{

Position += Movement;

Reference += Movement;

CalculateViewMatrix();

}

vec3 CCamera::OnKeys(BYTE Keys, float FrameTime)

{

float Speed = 5.0f;

if(Keys & 0x40) Speed \*= 2.0f;

if(Keys & 0x80) Speed \*= 0.5f;

float Distance = Speed \* FrameTime;

vec3 Up(0.0f, 1.0f, 0.0f);

vec3 Right = X;

vec3 Forward = cross(Up, Right);

Up \*= Distance;

Right \*= Distance;

Forward \*= Distance;

vec3 Movement;

if(Keys & 0x01) Movement += Forward;

if(Keys & 0x02) Movement -= Forward;

if(Keys & 0x04) Movement -= Right;

if(Keys & 0x08) Movement += Right;

if(Keys & 0x10) Movement += Up;

if(Keys & 0x20) Movement -= Up;

return Movement;

}

void CCamera::OnMouseMove(int dx, int dy)

{

float Sensitivity = 0.25f;

Position -= Reference;

if(dx != 0)

{

float DeltaX = (float)dx \* Sensitivity;

X = rotate(X, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Y = rotate(Y, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Z = rotate(Z, DeltaX, vec3(0.0f, 1.0f, 0.0f));

}

if(dy != 0)

{

float DeltaY = (float)dy \* Sensitivity;

Y = rotate(Y, DeltaY, X);

Z = rotate(Z, DeltaY, X);

if(Y.y < 0.0f)

{

Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);

Y = cross(Z, X);

}

}

Position = Reference + Z \* length(Position);

CalculateViewMatrix();

}

void CCamera::OnMouseWheel(float zDelta)

{

Position -= Reference;

if(zDelta < 0 && length(Position) < 500.0f)

{

Position += Position \* 0.1f;

}

if(zDelta > 0 && length(Position) > 0.05f)

{

Position -= Position \* 0.1f;

}

Position += Reference;

CalculateViewMatrix();

}

void CCamera::SetViewMatrixPointer(float \*ViewMatrix, float \*ViewMatrixInverse)

{

this->ViewMatrix = (mat4x4\*)ViewMatrix;

this->ViewMatrixInverse = (mat4x4\*)ViewMatrixInverse;

CalculateViewMatrix();

}

void CCamera::CalculateViewMatrix()

{

if(ViewMatrix != NULL)

{

\*ViewMatrix = mat4x4(X.x, Y.x, Z.x, 0.0f, X.y, Y.y, Z.y, 0.0f, X.z, Y.z, Z.z, 0.0f, -dot(X, Position), -dot(Y, Position), -dot(Z, Position), 1.0f);

if(ViewMatrixInverse != NULL)

{

\*ViewMatrixInverse = inverse(\*ViewMatrix);

}

}

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera Camera;

// ----------------------------------------------------------------------------------------------------------------------------

CWave::CWave()

{

Speed = 1.0f;

FrequencyMPIM2 = 4.0f \* (float)M\_PI \* 2.0f;

}

CWave::~CWave()

{

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer::COpenGLRenderer()

{

Pause = false;

WireFrame = false;

WaterLevel = 0.5f;

Wave = 0;

Camera.SetViewMatrixPointer(&ViewMatrix, &ViewMatrixInverse);

}

COpenGLRenderer::~COpenGLRenderer()

{

}

bool COpenGLRenderer::Init()

{

bool Error = false;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

ErrorLog.Append("GL\_ARB\_texture\_non\_power\_of\_two not supported!\r\n");

Error = true;

}

if(!GLEW\_EXT\_framebuffer\_object)

{

ErrorLog.Append("GL\_EXT\_framebuffer\_object not supported!\r\n");

Error = true;

}

char \*TextureFileName[] = {"kocka.jpg", "podlaha.jpg", "stena.jpg"};

for(int i = 0; i < 3; i++)

{

Error |= !Texture[i].LoadTexture2D(TextureFileName[i]);

}

Error |= !Scene.Load("scene.vs", "scene.fs");

Error |= !Water.Load("water.vs", "water.fs");

if(Error)

{

return false;

}

glUseProgram(Water);

glUniform1i(glGetUniformLocation(Water, "ReflectionTexture"), 0);

glUniform1i(glGetUniformLocation(Water, "RefractionTexture"), 1);

glUniform1f(glGetUniformLocation(Water, "WMSDWMR"), WMS / (float)WMR);

glUniform1f(glGetUniformLocation(Water, "WMSDWMRM2"), WMS / (float)WMR \* 2.0f);

glUseProgram(0);

Scene.UniformLocations = new GLuint[5];

Scene.UniformLocations[0] = glGetUniformLocation(Scene, "ClipType");

Scene.UniformLocations[1] = glGetUniformLocation(Scene, "WaterLevel");

Scene.UniformLocations[2] = glGetUniformLocation(Scene, "Texturing");

Scene.UniformLocations[3] = glGetUniformLocation(Scene, "NormalMatrix");

Scene.UniformLocations[4] = glGetUniformLocation(Scene, "ModelMatrix");

Water.UniformLocations = new GLuint[4 + MAX\_WAVES \* 5];

Water.UniformLocations[0] = glGetUniformLocation(Water, "Time");

Water.UniformLocations[1] = glGetUniformLocation(Water, "WaterLevel");

Water.UniformLocations[2] = glGetUniformLocation(Water, "CameraPosition");

Water.UniformLocations[3] = glGetUniformLocation(Water, "NormalMatrix");

CString Variable;

for(int Wave = 0; Wave < MAX\_WAVES; Wave++)

{

Variable.Set("Waves[%d].Position", Wave);

Water.UniformLocations[4 + Wave \* 5 + 0] = glGetUniformLocation(Water, Variable);

Variable.Set("Waves[%d].StartTime", Wave);

Water.UniformLocations[4 + Wave \* 5 + 1] = glGetUniformLocation(Water, Variable);

Variable.Set("Waves[%d].Speed", Wave);

Water.UniformLocations[4 + Wave \* 5 + 2] = glGetUniformLocation(Water, Variable);

Variable.Set("Waves[%d].MaxY", Wave);

Water.UniformLocations[4 + Wave \* 5 + 3] = glGetUniformLocation(Water, Variable);

Variable.Set("Waves[%d].FrequencyMPIM2", Wave);

Water.UniformLocations[4 + Wave \* 5 + 4] = glGetUniformLocation(Water, Variable);

}

glGenBuffers(1, &TexCoordsVBO);

glGenBuffers(1, &NormalsVBO);

glGenBuffers(1, &VerticesVBO);

glGenBuffers(1, &WaterVerticesVBO);

InitSceneVertexBuffers();

InitWaterVertexBuffer();

glGenTextures(1, &ReflectionTexture);

glGenTextures(1, &RefractionTexture);

glGenTextures(1, &DepthTexture);

glGenFramebuffersEXT(1, &FBO);

LightColor = vec3(1.0f, 1.0f, 1.0f);

LightPosition = vec3(0.0f, 2.75f, -4.75f);

glLightfv(GL\_LIGHT0, GL\_AMBIENT, &vec4(LightColor \* 0.25f, 1.0f));

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, &vec4(LightColor \* 0.75f, 1.0f));

glLightfv(GL\_LIGHT0, GL\_SPECULAR, &vec4(LightColor, 1.0f));

glLightf(GL\_LIGHT0, GL\_LINEAR\_ATTENUATION, 1.0f / 128.0f);

glLightf(GL\_LIGHT0, GL\_QUADRATIC\_ATTENUATION, 1.0f / 256.0f);

Camera.Look(vec3(0.0f, 1.75f, 5.0f), vec3(0.0f, 1.5f, 0.0f), true);

// Camera.Look(vec3(0.0f, 1.75f, 4.0f), vec3(0.0f, 0.5f, 0.0f));

srand(GetTickCount());

return true;

}

void COpenGLRenderer::Render(float FrameTime)

{

// add wave ---------------------------------------------------------------------------------------------------------------

static DWORD LastTime = GetTickCount();

if(!Pause)

{

DWORD Time = GetTickCount();

if(Time - LastTime > 250)

{

Waves[Wave].Position.x = -WMS / 2.0f + WMS \* (float)rand() / (float)RAND\_MAX;

Waves[Wave].Position.y = -WMS / 2.0f + WMS \* (float)rand() / (float)RAND\_MAX;

Waves[Wave].StartTime = (float)Time \* 0.001f;

Waves[Wave].MaxY = MAXY \* (float)rand() / (float)RAND\_MAX;

LastTime = Time;

SetWaveInShaders(Wave++);

Wave %= MAX\_WAVES;

}

}

// reset modelview matrix and set light position --------------------------------------------------------------------------

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glLightfv(GL\_LIGHT0, GL\_POSITION, &vec4(LightPosition, 1.0f));

// render reflection texture ----------------------------------------------------------------------------------------------

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, ReflectionTexture, 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, DepthTexture, 0);

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

glLoadMatrixf(&ViewMatrix);

RenderScene(1);

glTranslatef(0.0f, WaterLevel \* 2.0f, 0.0f);

glScalef(1.0f, -1.0f, 1.0f);

glCullFace(GL\_FRONT);

RenderScene(1);

glCullFace(GL\_BACK);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

// render refraction texture ----------------------------------------------------------------------------------------------

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

glLoadMatrixf(&ViewMatrix);

RenderScene(0);

glBindTexture(GL\_TEXTURE\_2D, RefractionTexture);

glCopyTexSubImage2D(GL\_TEXTURE\_2D, 0, 0, 0, 0, 0, Width, Height);

glBindTexture(GL\_TEXTURE\_2D, 0);

// render water mesh ------------------------------------------------------------------------------------------------------

glEnable(GL\_DEPTH\_TEST);

if(WireFrame)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

}

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, ReflectionTexture);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, RefractionTexture);

glEnableClientState(GL\_VERTEX\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, WaterVerticesVBO);

glVertexPointer(2, GL\_FLOAT, 8, (void\*)0);

glUseProgram(Water);

glUniform1f(Water.UniformLocations[0], (float)GetTickCount() \* 0.001f);

glUniform1f(Water.UniformLocations[1], WaterLevel);

glUniform3fv(Water.UniformLocations[2], 1, &Camera.Position);

vec3 X = Camera.X, Y = vec3(0.0f, 1.0f, 0.0f), Z = cross(X, Y);

NormalMatrix = mat3x3(X.x, Y.x, Z.x, X.y, Y.y, Z.y, X.z, Y.z, Z.z);

glUniformMatrix3fv(Water.UniformLocations[3], 1, GL\_FALSE, &NormalMatrix);

glDrawArrays(GL\_QUADS, 0, WaterVerticesCount);

glUseProgram(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glDisableClientState(GL\_VERTEX\_ARRAY);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, 0);

if(WireFrame) glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);

glDisable(GL\_DEPTH\_TEST);

// rotate object ----------------------------------------------------------------------------------------------------------

if(!Pause)

{

static float a = 0.0f;

ModelMatrix = translate(0.0f, 1.5f, 0.0f) \* rotate(a, vec3(0.0f, 1.0f, 0.0f)) \* rotate(a, vec3(1.0f, 0.0f, 0.0f));

a += 22.5f \* FrameTime;

}

}

void COpenGLRenderer::Resize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

glViewport(0, 0, Width, Height);

ProjectionMatrix = perspective(45.0f, (float)Width / (float)Height, 0.125f, 512.0f);

ProjectionBiasMatrixInverse = inverse(ProjectionMatrix) \* BiasMatrixInverse;

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&ProjectionMatrix);

glBindTexture(GL\_TEXTURE\_2D, ReflectionTexture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindTexture(GL\_TEXTURE\_2D, RefractionTexture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindTexture(GL\_TEXTURE\_2D, DepthTexture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT24, Width, Height, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL);

glBindTexture(GL\_TEXTURE\_2D, 0);

}

void COpenGLRenderer::Destroy()

{

for(int i = 0; i < 3; i++)

{

Texture[i].Destroy();

}

Scene.Destroy();

Water.Destroy();

glDeleteBuffers(1, &TexCoordsVBO);

glDeleteBuffers(1, &NormalsVBO);

glDeleteBuffers(1, &VerticesVBO);

glDeleteBuffers(1, &WaterVerticesVBO);

glDeleteTextures(1, &ReflectionTexture);

glDeleteTextures(1, &RefractionTexture);

glDeleteTextures(1, &DepthTexture);

if(GLEW\_EXT\_framebuffer\_object)

{

glDeleteFramebuffersEXT(1, &FBO);

}

}

void COpenGLRenderer::RenderScene(int ClipType)

{

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

glBindBuffer(GL\_ARRAY\_BUFFER, TexCoordsVBO);

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glTexCoordPointer(2, GL\_FLOAT, 8, (void\*)0);

glBindBuffer(GL\_ARRAY\_BUFFER, NormalsVBO);

glEnableClientState(GL\_NORMAL\_ARRAY);

glNormalPointer(GL\_FLOAT, 12, (void\*)0);

glBindBuffer(GL\_ARRAY\_BUFFER, VerticesVBO);

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(3, GL\_FLOAT, 12, (void\*)0);

glUseProgram(Scene);

glUniform1i(Scene.UniformLocations[0], ClipType);

glUniform1f(Scene.UniformLocations[1], WaterLevel);

glUniform1i(Scene.UniformLocations[2], 1);

glUniformMatrix3fv(Scene.UniformLocations[3], 1, GL\_FALSE, &mat3x3());

glUniformMatrix4fv(Scene.UniformLocations[4], 1, GL\_FALSE, &mat4x4());

glColor3f(1.0f, 1.0f, 1.0f);

glBindTexture(GL\_TEXTURE\_2D, Texture[0]);

glDrawArrays(GL\_QUADS, 0, 96);

glBindTexture(GL\_TEXTURE\_2D, Texture[1]);

glDrawArrays(GL\_QUADS, 96, 4);

glBindTexture(GL\_TEXTURE\_2D, Texture[2]);

glDrawArrays(GL\_QUADS, 100, 80);

glBindTexture(GL\_TEXTURE\_2D, 0);

glUniform1i(Scene.UniformLocations[2], 0);

glDrawArrays(GL\_QUADS, 180, 4);

glUniformMatrix3fv(Scene.UniformLocations[3], 1, GL\_FALSE, &mat3x3(ModelMatrix));

glUniformMatrix4fv(Scene.UniformLocations[4], 1, GL\_FALSE, &ModelMatrix);

glColor3f(0.33f, 0.66f, 1.0f);

glDrawArrays(GL\_QUADS, 184, 72);

glUseProgram(0);

glDisableClientState(GL\_VERTEX\_ARRAY);

glDisableClientState(GL\_NORMAL\_ARRAY);

glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glDisable(GL\_CULL\_FACE);

glDisable(GL\_DEPTH\_TEST);

}

void COpenGLRenderer::AddWaveByMouseClick(int x, int y)

{

float s = (float)x / (float)(Width - 1);

float t = 1.0f - (float)y / (float)(Height - 1);

vec4 Position = ViewMatrixInverse \* (ProjectionBiasMatrixInverse \* vec4(s, t, 0.5f, 1.0f));

Position /= Position.w;

vec3 Ray = normalize(\*(vec3\*)&Position - Camera.Position);

vec3 Normal = vec3(0.0f, 1.0f, 0.0f);

float D = -dot(Normal, vec3(0.0f, WaterLevel, 0.0f));

float NdotR = -dot(Normal, Ray);

if(NdotR != 0.0f)

{

float Distance = (dot(Normal, Camera.Position) + D) / NdotR;

if(Distance > 0.0)

{

vec3 Position = Camera.Position + Ray \* Distance;

float WMSD2 = WMS / 2.0f, MWMSD2 = -WMSD2;

if(Position.x >= MWMSD2 && Position.x <= WMSD2 && Position.z >= MWMSD2 && Position.z <= WMSD2)

{

Waves[Wave].Position.x = Position.x;

Waves[Wave].Position.y = Position.z;

Waves[Wave].StartTime = (float)GetTickCount() \* 0.001f;

Waves[Wave].MaxY = MAXY;

SetWaveInShaders(Wave++);

Wave %= MAX\_WAVES;

}

}

}

}

void COpenGLRenderer::SetWaveInShaders(int Wave)

{

glUseProgram(Water);

glUniform2fv(Water.UniformLocations[4 + Wave \* 5 + 0], 1, &Waves[Wave].Position);

glUniform1f(Water.UniformLocations[4 + Wave \* 5 + 1], Waves[Wave].StartTime);

glUniform1f(Water.UniformLocations[4 + Wave \* 5 + 2], Waves[Wave].Speed);

glUniform1f(Water.UniformLocations[4 + Wave \* 5 + 3], Waves[Wave].MaxY);

glUniform1f(Water.UniformLocations[4 + Wave \* 5 + 4], Waves[Wave].FrequencyMPIM2);

glUseProgram(0);

}

void COpenGLRenderer::IncreaseWaterLevel(float Increment)

{

WaterLevel += Increment;

if(WaterLevel < 0.25f) WaterLevel = 0.25f;

if(WaterLevel > 2.75f) WaterLevel = 2.75f;

}

void COpenGLRenderer::InitWaterVertexBuffer()

{

WaterVerticesCount = WMR \* WMR \* 4;

vec2 \*WaterVertices = new vec2[WaterVerticesCount];

float WMSD2 = WMS / 2.0f, MWMSD2 = -WMSD2, WMSDWMR = WMS / (float)WMR;

int i = 0;

for(int y = 0; y < WMR; y++)

{

for(int x = 0; x < WMR; x++)

{

WaterVertices[i].x = MWMSD2 + x \* WMSDWMR;

WaterVertices[i++].y = WMSD2 - y \* WMSDWMR;

WaterVertices[i].x = MWMSD2 + (x + 1) \* WMSDWMR;

WaterVertices[i++].y = WMSD2 - y \* WMSDWMR;

WaterVertices[i].x = MWMSD2 + (x + 1) \* WMSDWMR;

WaterVertices[i++].y = WMSD2 - (y + 1) \* WMSDWMR;

WaterVertices[i].x = MWMSD2 + x \* WMSDWMR;

WaterVertices[i++].y = WMSD2 - (y + 1) \* WMSDWMR;

}

}

glBindBuffer(GL\_ARRAY\_BUFFER, WaterVerticesVBO);

glBufferData(GL\_ARRAY\_BUFFER, WaterVerticesCount \* 2 \* 4, WaterVertices, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

delete [] WaterVertices;

}

void COpenGLRenderer::InitSceneVertexBuffers()

{

vec2 \*TexCoords = new vec2[256];

vec3 \*Normals = new vec3[256];

vec3 \*Vertices = new vec3[256];

int pos = 0;

vec3 m;

// cubes

m = vec3( 0.0f, 0.5f, 0.0f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

m = vec3( 0.0f, 1.5f,-4.5f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

m = vec3(-0.625f, 0.5f,-4.5f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

m = vec3( 0.625f, 0.5f,-4.5f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x,-0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.5f + m.y, 0.5f + m.z);

// floor

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-5.0f, 0.0f, 5.0f);

TexCoords[pos] = vec2(10.0f, 0.0f); Normals[pos] = vec3(0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 5.0f, 0.0f, 5.0f);

TexCoords[pos] = vec2(10.0f, 10.0f); Normals[pos] = vec3(0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 5.0f, 0.0f, -5.0f);

TexCoords[pos] = vec2( 0.0f, 10.0f); Normals[pos] = vec3(0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-5.0f, 0.0f, -5.0f);

// walls

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 5.0f, 0.0f,-5.0f);

TexCoords[pos] = vec2(10.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 5.0f, 0.0f, 5.0f);

TexCoords[pos] = vec2(10.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 5.0f, 3.0f, 5.0f);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 5.0f, 3.0f,-5.0f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-5.0f, 0.0f, 5.0f);

TexCoords[pos] = vec2(10.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-5.0f, 0.0f,-5.0f);

TexCoords[pos] = vec2(10.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-5.0f, 3.0f,-5.0f);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-5.0f, 3.0f, 5.0f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, -1.0f); Vertices[pos++] = vec3( 5.0f, 0.0f, 5.0f);

TexCoords[pos] = vec2(10.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, -1.0f); Vertices[pos++] = vec3(-5.0f, 0.0f, 5.0f);

TexCoords[pos] = vec2(10.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, -1.0f); Vertices[pos++] = vec3(-5.0f, 3.0f, 5.0f);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, -1.0f); Vertices[pos++] = vec3( 5.0f, 3.0f, 5.0f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-5.0f, 0.0f,-5.0f);

TexCoords[pos] = vec2(10.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 5.0f, 0.0f,-5.0f);

TexCoords[pos] = vec2(10.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 5.0f, 3.0f,-5.0f);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-5.0f, 3.0f,-5.0f);

// pillars

m = vec3(-2.5f, 0.0f,-2.5f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f, 0.5f + m.z);

m = vec3( 2.5f, 0.0f,-2.5f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f, 0.5f + m.z);

m = vec3( 2.5f, 0.0f, 2.5f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f, 0.5f + m.z);

m = vec3(-2.5f, 0.0f, 2.5f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f,-0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 0.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 1.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.5f + m.x, 3.0f, 0.5f + m.z);

TexCoords[pos] = vec2( 0.0f, 3.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.5f + m.x, 3.0f, 0.5f + m.z);

// ceiling

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(0.0f, -1.0f, 0.0f); Vertices[pos++] = vec3(-5.0f, 3.0f,-5.0f);

TexCoords[pos] = vec2(10.0f, 0.0f); Normals[pos] = vec3(0.0f, -1.0f, 0.0f); Vertices[pos++] = vec3( 5.0f, 3.0f,-5.0f);

TexCoords[pos] = vec2(10.0f, 10.0f); Normals[pos] = vec3(0.0f, -1.0f, 0.0f); Vertices[pos++] = vec3( 5.0f, 3.0f, 5.0f);

TexCoords[pos] = vec2( 0.0f, 10.0f); Normals[pos] = vec3(0.0f, -1.0f, 0.0f); Vertices[pos++] = vec3(-5.0f, 3.0f, 5.0f);

// rotating object

m = vec3(0.0f);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.25f + m.x,-0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.25f + m.x,-0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.25f + m.x, 0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.25f + m.x, 0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.25f + m.x,-0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.25f + m.x,-0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.25f + m.x, 0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.25f + m.x, 0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.25f + m.x,-0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.25f + m.x,-0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.25f + m.x,-0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.25f + m.x,-0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.25f + m.x, 0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.25f + m.x, 0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.25f + m.x, 0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.25f + m.x, 0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.25f + m.x,-0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.25f + m.x,-0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.25f + m.x, 0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.25f + m.x, 0.025f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.25f + m.x,-0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.25f + m.x,-0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.25f + m.x, 0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.25f + m.x, 0.025f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.25f + m.y,-0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.25f + m.y, 0.025f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3(-1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 1.0f, 0.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f,-1.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 1.0f, 0.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f,-1.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.025f + m.y,-0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.025f + m.x,-0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 0.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.025f + m.x,-0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 1.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3( 0.025f + m.x, 0.025f + m.y, 0.25f + m.z);

TexCoords[pos] = vec2( 0.0f, 1.0f); Normals[pos] = vec3( 0.0f, 0.0f, 1.0f); Vertices[pos++] = vec3(-0.025f + m.x, 0.025f + m.y, 0.25f + m.z);

glBindBuffer(GL\_ARRAY\_BUFFER, TexCoordsVBO);

glBufferData(GL\_ARRAY\_BUFFER, 256 \* 2 \* 4, TexCoords, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, NormalsVBO);

glBufferData(GL\_ARRAY\_BUFFER, 256 \* 3 \* 4, Normals, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, VerticesVBO);

glBufferData(GL\_ARRAY\_BUFFER, 256 \* 3 \* 4, Vertices, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

delete [] TexCoords;

delete [] Normals;

delete [] Vertices;

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer OpenGLRenderer;

// ----------------------------------------------------------------------------------------------------------------------------

CString ModuleDirectory, ErrorLog;

// ----------------------------------------------------------------------------------------------------------------------------

void GetModuleDirectory()

{

char \*moduledirectory = new char[256];

GetModuleFileName(GetModuleHandle(NULL), moduledirectory, 256);

\*(strrchr(moduledirectory, '\\') + 1) = 0;

ModuleDirectory = moduledirectory;

delete [] moduledirectory;

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView::COpenGLView()

{

}

COpenGLView::~COpenGLView()

{

}

bool COpenGLView::Init(HINSTANCE hInstance, char \*Title, int Width, int Height, int Samples)

{

this->Title = Title;

this->Width = Width;

this->Height = Height;

WNDCLASSEX WndClassEx;

memset(&WndClassEx, 0, sizeof(WNDCLASSEX));

WndClassEx.cbSize = sizeof(WNDCLASSEX);

WndClassEx.style = CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

WndClassEx.lpfnWndProc = WndProc;

WndClassEx.hInstance = hInstance;

WndClassEx.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hCursor = LoadCursor(NULL, IDC\_ARROW);

WndClassEx.lpszClassName = "Win32OpenGLWindowClass";

if(RegisterClassEx(&WndClassEx) == 0)

{

ErrorLog.Set("RegisterClassEx failed!");

return false;

}

DWORD Style = WS\_OVERLAPPEDWINDOW | WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN;

hWnd = CreateWindowEx(WS\_EX\_APPWINDOW, WndClassEx.lpszClassName, Title, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL);

if(hWnd == NULL)

{

ErrorLog.Set("CreateWindowEx failed!");

return false;

}

HDC hDC = GetDC(hWnd);

if(hDC == NULL)

{

ErrorLog.Set("GetDC failed!");

return false;

}

PIXELFORMATDESCRIPTOR pfd;

memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));

pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);

pfd.nVersion = 1;

pfd.dwFlags = PFD\_DRAW\_TO\_WINDOW | PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER;

pfd.iPixelType = PFD\_TYPE\_RGBA;

pfd.cColorBits = 32;

pfd.cDepthBits = 24;

pfd.iLayerType = PFD\_MAIN\_PLANE;

int PixelFormat = ChoosePixelFormat(hDC, &pfd);

if(PixelFormat == 0)

{

ErrorLog.Set("ChoosePixelFormat failed!");

return false;

}

static int MSAAPixelFormat = 0;

if(SetPixelFormat(hDC, MSAAPixelFormat == 0 ? PixelFormat : MSAAPixelFormat, &pfd) == FALSE)

{

ErrorLog.Set("SetPixelFormat failed!");

return false;

}

hGLRC = wglCreateContext(hDC);

if(hGLRC == NULL)

{

ErrorLog.Set("wglCreateContext failed!");

return false;

}

if(wglMakeCurrent(hDC, hGLRC) == FALSE)

{

ErrorLog.Set("wglMakeCurrent failed!");

return false;

}

if(glewInit() != GLEW\_OK)

{

ErrorLog.Set("glewInit failed!");

return false;

}

if(!GLEW\_VERSION\_2\_1)

{

ErrorLog.Set("OpenGL 2.1 not supported!");

return false;

}

if(MSAAPixelFormat == 0 && Samples > 0)

{

if(GLEW\_ARB\_multisample && WGLEW\_ARB\_pixel\_format)

{

while(Samples > 0)

{

UINT NumFormats = 0;

int PFAttribs[] =

{

WGL\_DRAW\_TO\_WINDOW\_ARB, GL\_TRUE,

WGL\_SUPPORT\_OPENGL\_ARB, GL\_TRUE,

WGL\_DOUBLE\_BUFFER\_ARB, GL\_TRUE,

WGL\_PIXEL\_TYPE\_ARB, WGL\_TYPE\_RGBA\_ARB,

WGL\_COLOR\_BITS\_ARB, 32,

WGL\_DEPTH\_BITS\_ARB, 24,

WGL\_ACCELERATION\_ARB, WGL\_FULL\_ACCELERATION\_ARB,

WGL\_SAMPLE\_BUFFERS\_ARB, GL\_TRUE,

WGL\_SAMPLES\_ARB, Samples,

0

};

if(wglChoosePixelFormatARB(hDC, PFAttribs, NULL, 1, &MSAAPixelFormat, &NumFormats) == TRUE && NumFormats > 0) break;

Samples--;

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

UnregisterClass(WndClassEx.lpszClassName, hInstance);

return Init(hInstance, Title, Width, Height, Samples);

}

else

{

Samples = 0;

}

}

this->Samples = Samples;

GetModuleDirectory();

glGetIntegerv(GL\_MAX\_TEXTURE\_SIZE, &gl\_max\_texture\_size);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glGetIntegerv(GL\_MAX\_TEXTURE\_MAX\_ANISOTROPY\_EXT, &gl\_max\_texture\_max\_anisotropy\_ext);

}

if(WGLEW\_EXT\_swap\_control)

{

wglSwapIntervalEXT(0);

}

return OpenGLRenderer.Init();

}

void COpenGLView::Show(bool Maximized)

{

RECT dRect, wRect, cRect;

GetWindowRect(GetDesktopWindow(), &dRect);

GetWindowRect(hWnd, &wRect);

GetClientRect(hWnd, &cRect);

wRect.right += Width - cRect.right;

wRect.bottom += Height - cRect.bottom;

wRect.right -= wRect.left;

wRect.bottom -= wRect.top;

wRect.left = dRect.right / 2 - wRect.right / 2;

wRect.top = dRect.bottom / 2 - wRect.bottom / 2;

MoveWindow(hWnd, wRect.left, wRect.top, wRect.right, wRect.bottom, FALSE);

ShowWindow(hWnd, Maximized ? SW\_SHOWMAXIMIZED : SW\_SHOWNORMAL);

}

void COpenGLView::MessageLoop()

{

MSG Msg;

while(GetMessage(&Msg, NULL, 0, 0) > 0)

{

TranslateMessage(&Msg);

DispatchMessage(&Msg);

}

}

void COpenGLView::Destroy()

{

if(GLEW\_VERSION\_2\_1)

{

OpenGLRenderer.Destroy();

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

}

void COpenGLView::OnKeyDown(UINT Key)

{

switch(Key)

{

case VK\_F1:

OpenGLRenderer.WireFrame = !OpenGLRenderer.WireFrame;

break;

case VK\_SPACE:

OpenGLRenderer.Pause = !OpenGLRenderer.Pause;

break;

case VK\_ADD:

OpenGLRenderer.IncreaseWaterLevel(0.05f);

break;

case VK\_SUBTRACT:

OpenGLRenderer.IncreaseWaterLevel(-0.05f);

break;

}

}

void COpenGLView::OnLButtonDown(int X, int Y)

{

OpenGLRenderer.AddWaveByMouseClick(X, Y);

}

void COpenGLView::OnMouseMove(int X, int Y)

{

if(GetKeyState(VK\_RBUTTON) & 0x80)

{

Camera.OnMouseMove(LastX - X, LastY - Y);

LastX = X;

LastY = Y;

}

}

void COpenGLView::OnMouseWheel(short zDelta)

{

Camera.OnMouseWheel(zDelta);

}

void COpenGLView::OnPaint()

{

static DWORD LastFPSTime = GetTickCount(), LastFrameTime = LastFPSTime, FPS = 0;

PAINTSTRUCT ps;

HDC hDC = BeginPaint(hWnd, &ps);

DWORD Time = GetTickCount();

float FrameTime = (Time - LastFrameTime) \* 0.001f;

LastFrameTime = Time;

if(Time - LastFPSTime > 1000)

{

CString Text = Title;

if(OpenGLRenderer.Text[0] != 0)

{

Text.Append(" - " + OpenGLRenderer.Text);

}

Text.Append(" - %dx%d", Width, Height);

Text.Append(", ATF %dx", gl\_max\_texture\_max\_anisotropy\_ext);

Text.Append(", MSAA %dx", Samples);

Text.Append(", FPS: %d", FPS);

Text.Append(" - %s", glGetString(GL\_RENDERER));

SetWindowText(hWnd, Text);

LastFPSTime = Time;

FPS = 0;

}

else

{

FPS++;

}

BYTE Keys = 0x00;

if(GetKeyState('W') & 0x80) Keys |= 0x01;

if(GetKeyState('S') & 0x80) Keys |= 0x02;

if(GetKeyState('A') & 0x80) Keys |= 0x04;

if(GetKeyState('D') & 0x80) Keys |= 0x08;

if(GetKeyState('R') & 0x80) Keys |= 0x10;

if(GetKeyState('F') & 0x80) Keys |= 0x20;

if(GetKeyState(VK\_SHIFT) & 0x80) Keys |= 0x40;

if(GetKeyState(VK\_CONTROL) & 0x80) Keys |= 0x80;

if(Keys & 0x3F)

{

Camera.Move(Camera.OnKeys(Keys, FrameTime));

}

OpenGLRenderer.Render(FrameTime);

SwapBuffers(hDC);

EndPaint(hWnd, &ps);

InvalidateRect(hWnd, NULL, FALSE);

}

void COpenGLView::OnRButtonDown(int X, int Y)

{

LastX = X;

LastY = Y;

}

void COpenGLView::OnSize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

OpenGLRenderer.Resize(Width, Height);

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView OpenGLView;

// ----------------------------------------------------------------------------------------------------------------------------

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)

{

switch(uiMsg)

{

case WM\_CLOSE:

PostQuitMessage(0);

break;

case WM\_LBUTTONDOWN:

OpenGLView.OnLButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_MOUSEMOVE:

OpenGLView.OnMouseMove(LOWORD(lParam), HIWORD(lParam));

break;

case 0x020A: // WM\_MOUSWHEEL

OpenGLView.OnMouseWheel(HIWORD(wParam));

break;

case WM\_KEYDOWN:

OpenGLView.OnKeyDown((UINT)wParam);

break;

case WM\_PAINT:

OpenGLView.OnPaint();

break;

case WM\_RBUTTONDOWN:

OpenGLView.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_SIZE:

OpenGLView.OnSize(LOWORD(lParam), HIWORD(lParam));

break;

default:

return DefWindowProc(hWnd, uiMsg, wParam, lParam);

}

return 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine, int iShow)

{

char \*AppName = "Interactive water surface, reflection and refraction";

if(OpenGLView.Init(hInstance, AppName, 800, 600, 0))

{

OpenGLView.Show();

OpenGLView.MessageLoop();

}

else

{

MessageBox(NULL, ErrorLog, AppName, MB\_OK | MB\_ICONERROR);

}

OpenGLView.Destroy();

return 0;

}

//////////////////////////////////////////Comments////////////////////////////////////////////////////////////////////////

**Comments (14)**

|  |
| --- |
| **Admin**, February 13, 2014, 06:00 PM |
| Calculating y offsets of the water mesh vertices in water.zip/water.vs:  ...  void main() {  ...   vec3 Vertices[5];   ...   Vertices[4] = vec3(gl\_Vertex.x, WaterLevel, gl\_Vertex.y);   for(int wi = 0; wi < MAX\_WAVES; wi++)  {  for(int vi = 0; vi < 5; vi++)  {  float d = distance(Waves[wi].Position, Vertices[vi].xz);  float t = Time - Waves[wi].StartTime - d / Waves[wi].Speed;   if(t > 0.0)  {  float maxy = Waves[wi].MaxY - Waves[wi].MaxY \* t;   if(maxy > 0.0)  {  Vertices[vi].y -= sin(t \* Waves[wi].FrequencyMPIM2) \* maxy / (1.0 + d);  }  }  }  }   Position = Vertices[4];   ... }  A wave is spreading "Time - Waves[wi].StartTime" seconds.  "d / Waves[wi].Speed" is time that takes a wave to reach a vertex.  If t < 0, then a wave hasn't reached a vertex yet.  Every vertex moves up and down for only 1 second and this waving is attenuated by time and distance. Thus "- Waves[wi].MaxY \* t" and "/ (1.0 + d)". |

|  |
| --- |
| **Admin**, February 13, 2014, 06:01 PM |
| Calculating normals of the water mesh vertices in water.zip/water.vs:  ...  void main() {  ...   Vertices[0] = vec3(gl\_Vertex.x + WMSDWMR, WaterLevel, gl\_Vertex.y);  Vertices[1] = vec3(gl\_Vertex.x, WaterLevel, gl\_Vertex.y - WMSDWMR);  Vertices[2] = vec3(gl\_Vertex.x - WMSDWMR, WaterLevel, gl\_Vertex.y);  Vertices[3] = vec3(gl\_Vertex.x, WaterLevel, gl\_Vertex.y + WMSDWMR);   ...   Normal = NormalMatrix \* vec3(Vertices[2].y - Vertices[0].y, WMSDWMRM2, Vertices[1].y - Vertices[3].y);   ... }  Normal is calculated as a sum of 4 cross products of 4 3D vectors to 4 closest neighboring vertices.  vec3(Vertices[2].y - Vertices[0].y, WMSDWMRM2, Vertices[1].y - Vertices[3].y)  is a result of a mathematical simplification:  vec3 Normal = vec3(0.0);  vec3 a = Vertices[0] - Vertices[4] = vec3(WMSDWMR, y0, 0.0); vec3 b = Vertices[1] - Vertices[4] = vec3(0.0, y1, -WMSDWMR); vec3 c = Vertices[2] - Vertices[4] = vec3(-WMSDWMR, y2, 0.0); vec3 d = Vertices[3] - Vertices[4] = vec3(0.0, y3, WMSDWMR);  Normal += cross(a, b); Normal += cross(b, c); Normal += cross(c, d); Normal += cross(d, a);  Normal = normalize(Normal);  vectors to neighboring vertices:  vec3 a = vec3( WMSDWMR, y0, 0.0); vec3 b = vec3( 0.0, y1, -WMSDWMR); vec3 c = vec3(-WMSDWMR, y2, 0.0); vec3 d = vec3( 0.0, y3, WMSDWMR);  cheat table to calculate cross products:  y0, 0.0, WMSDWMR, y0 y1, -WMSDWMR, 0.0, y1 y2, 0.0, -WMSDWMR, y2 y3, WMSDWMR, 0.0, y3 y0, 0.0, WMSDWMR, y0  cross products:  vec3 ab = cross(a, b) = vec3(-WMSDWMR \* y0, WMSDWMR \* WMSDWMR, WMSDWMR \* y1); vec3 bc = cross(b, c) = vec3( WMSDWMR \* y2, WMSDWMR \* WMSDWMR, WMSDWMR \* y1); vec3 cd = cross(c, d) = vec3( WMSDWMR \* y2, WMSDWMR \* WMSDWMR, -WMSDWMR \* y3); vec3 da = cross(d, a) = vec3(-WMSDWMR \* y0, WMSDWMR \* WMSDWMR, -WMSDWMR \* y3);  sum of all cross products:  vec3 abcd = ab + bc + cd + da;  vec3 abcd = vec3(2.0 \* WMSDWMR \* y2 - 2.0 \* WMSDWMR \* y0, 4.0 \* WMSDWMR \* WMSDWMR, 2.0 \* WMSDWMR \* y1 - 2.0 \* WMSDWMR \* y3);  divided by 2.0 \* WMSDWMR:  abcd /= 2.0 \* WMSDWMR;  abcd = vec3(y2 - y0, 2.0 \* WMSDWMR, y1 - y3)  normalize to get normal  Notice that the y component of the not normalized normal is double the distance to neighboring vertices.  Normal is rotated because the texture coordinates to fetch the reflection and refraction textures depend on the normal and the angle between the world and camera space +X axes. |

|  |
| --- |
| **Admin**, February 13, 2014, 06:02 PM |
| The code in the water.zip archive is now improved, but it still isn't an efficient way to compute water effect.   It takes too much time to calculate y offsets of the water mesh vertices in water.zip/water.vs because y offsets of the same vertices are calculated multiple times.   I released this code just to show you the first version of the Interactive water surface tutorial. |

|  |
| --- |
| **Admin**, February 13, 2014, 07:03 PM |
| In the Interactive water surface tutorial y offsets are precalculated, stored in the alpha channel of the 512x512 normal / bump map texture and subsequently used to calculate normals, which are stored in the rgb channels.  Texture coordinates to fetch the normal / bump map texture are mapped to the water mesh vertices in this way:  "s = 0" is mapped to "x = - WMS / 2" "t = 0" is mapped to "z = + WMS / 2" "s = 1" is mapped to "x = + WMS / 2" "t = 1" is mapped to "z = - WMS / 2" |

|  |
| --- |
| **Admin**, February 16, 2014, 03:06 PM |
| Calculating caustic in interactive\_water\_surface.zip/scene.fs:  ...  void main() {  ...   if(Caustic == 1 && NdotLD > 0.0 && gl\_LightSource[0].position.y != WaterLevel)  {  vec2 TexCoord = vec2(Position.x \* ODWMS + 0.5, 0.5 - Position.z \* ODWMS);  float waterlevel = WaterLevel + texture2D(WaterNormalBumpMapTexture, TexCoord).a;   if((gl\_LightSource[0].position.y > WaterLevel && Position.y > waterlevel) || (gl\_LightSource[0].position.y < WaterLevel && Position.y < waterlevel))  {  LightDirection.xz = gl\_LightSource[0].position.xz - Position.xz;  LightDirection.y = WaterLevel \* 2.0 - gl\_LightSource[0].position.y - Position.y;   LightDistance2 = dot(LightDirection, LightDirection);  LightDistance = sqrt(LightDistance2);   LightDirection /= LightDistance;   NdotLD = max(0.0, dot(Normal, LightDirection));   Attenuation = gl\_LightSource[0].constantAttenuation;  Attenuation += gl\_LightSource[0].linearAttenuation \* LightDistance;  Attenuation += gl\_LightSource[0].quadraticAttenuation \* LightDistance2;  }   float Distance = (WaterLevel - Position.y) / LightDirection.y;   TexCoord = LightDirection.xz \* Distance + Position.xz;   TexCoord.s = TexCoord.s \* ODWMS + 0.5;  TexCoord.t = 0.5 - TexCoord.t \* ODWMS;   vec3 WaterNormal = texture2D(WaterNormalBumpMapTexture, TexCoord).rgb;   float WNdotLD = abs(dot(WaterNormal, LightDirection));  float Caustic = NdotLD \* pow(WNdotLD, 4.0) / Attenuation;  Light += gl\_LightSource[0].specular.rgb \* Caustic;  }   ... }  Position is the world space position of a pixel that belongs to an object.  We need to find texture coordinates to fetch a normal from the normal / bump map texture.  If "(gl\_LightSource[0].position.y > WaterLevel && Position.y > waterlevel) || (gl\_LightSource[0].position.y < WaterLevel && Position.y < waterlevel)", then the light is reflected, otherwise it's refracted.  "WaterLevel \* 2.0 - gl\_LightSource[0].position.y" is the light position as seen under the water surface, because it's reflected on the water surface.  If the WaterLevel is for example "0.5" and the light position y coordinate is for example "3.0", then the new light position y coordinate will be "0.5 \* 2.0 - 3.0 = -2.0".  To find the texture coordinates we need to do a ray plane intersection.  The ray origin is Position, the ray direction is LightDirection and the plane is the flat water surface with the equation "0.0 \* x + 1.0 \* y + 0.0 \* z - WaterLevel = 0.0".  "0.0 \* x + 1.0 \* y + 0.0 \* z" because the normal of this plane is "vec3(0.0, 1.0, 0.0)", because the flat water surface is parallel to the XZ plane.  And "- WaterLevel" because a point that lies on this plane is for example "vec3(0.0, WaterLevel, 0.0)".  And because "Normal.x = Normal.z = 0.0", the ray plane intersection can be simplified.  Negative cosinus of the angle between the ray direction and the normal is "-dot(Normal, RayDirection) = -LightDirection.y" and because it's used as a denominator, it can't be equal to "0.0", but this won't happen.  The distance of the ray origin from the flat water surface is "(dot(Normal, Position) - WaterLevel) / -LightDirection.y = (Position.y - WaterLevel) / -LightDirection.y = (WaterLevel - Position.y) / LightDirection.y".  The ray plane intersection point is "LightDirection \* Distance + Position" and because we're intrested in it's xz coordinates, it's simplified to "LightDirection.xz \* Distance + Position.xz". |

|  |
| --- |
| **Alan**, February 20, 2014, 02:38 PM |
| Thanks for the detailed explanation. |

|  |
| --- |
| **zxx43**, January 5, 2015, 07:13 PM |
| When render reflection texture, why do you call RenderScene(1) twice ?  I know the meaning of the second RenderScene(1), but whats the meaning of the first ?  In water.fs why the texcoord of reflection texture is (TexCoord + Offset) instead of TexCoord ?   Why the texcoord of refraction (TexCoord - Offset \* 0.8), (TexCoord - Offset \* 0.9), (TexCoord - Offset \* 1.0) ?  In water.fs Normal.xz \* vec2(0.05, -0.05) whats the meaning of vec2(0.05, -0.05) ?  Why Normal = NormalMatrix \* normalize(texture2D(WaterNormalBumpMapTexture, gl\_TexCoord[0].st).rgb) ?   How can you make LightDirectionReflected = reflect(LightDirection, Normal) ? I think LightDirection is in world space but Normal is not. |

|  |
| --- |
| **Admin**, January 8, 2015, 03:19 PM |
| Don't call the RenderScene(1) function for the first time and display the reflection texture on the screen, you'll see.  Try not to add the Offset vector to the TexCoord vector, you'll understand.  A trick to achieve chromatic dispersion.  Normal vector needs to be scaled a lot in order to be used as a texture coordinate offset vector.  To transform Normal to view space.  LightDirection is in view space because the MODELVIEW matrix is not identity matrix when the glLightfv function is called. |

|  |
| --- |
| **zxx43**, January 9, 2015, 08:29 AM |
| How did you calculate the value of vec2(0.05, -0.05) and how did you know they are 0.05 not 0.005 or 0.5? |

|  |
| --- |
| **Admin**, January 9, 2015, 08:47 AM |
| To clarify, the NormalMatrix is actually not a normal matrix, but just an Y-axis rotation matrix and the LightDirection really is in world space, so this is a bug - specular lighting is not calculated correctly because the original not rotated normal must be used in order to calculate the LightDirectionReflected vector. |

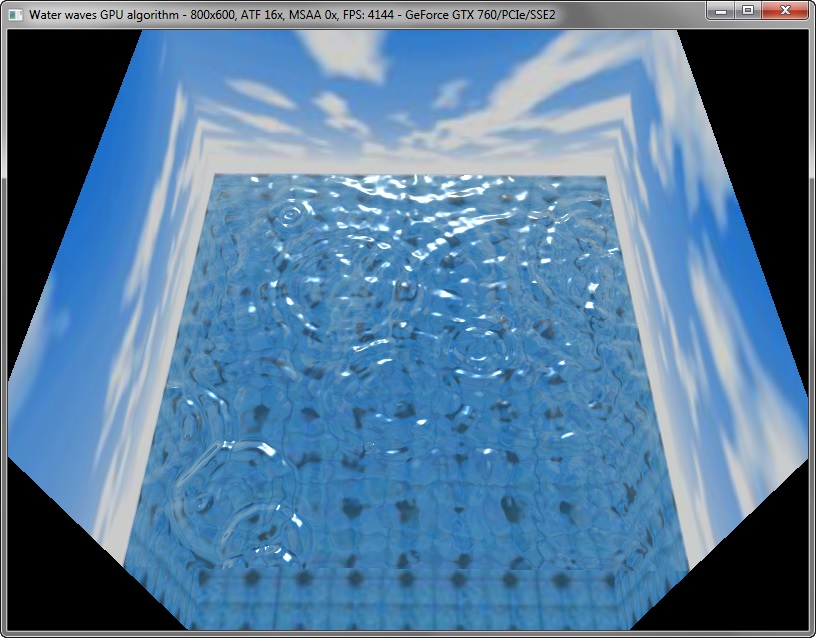
|  |
| --- |
| **Admin**, January 9, 2015, 08:49 AM |
| We did not calculate the vec2(0.05, -0.05) scale vector, we just used this specific values. |

|  |
| --- |
| **zxx43**, January 10, 2015, 07:04 AM |
| So the scale 0.05 is an empirical value. |

|  |
| --- |
| **zxx43**, January 14, 2015, 04:28 AM |
| There are some issues when render the refraction texture. You draw both things under water and above water, when there are some waves texture coordinates offset works, so we will see something above water in refraction texture. |

|  |
| --- |
| **Admin**, January 14, 2015, 04:51 PM |
| That is not an issue, that is a fact. This is not a realistic water effect. In modern games it's done in the same way.  **Interactive water surface, light reflection and refraction, caustic**  First, the water surface 512x512 normal-bump-map (NBM) RGBA32F texture is created using a wave function in the fragment shader. The surface normal is calculated as an optimized cross product of 4 neighboring vertices and is stored in the rgb channels. The surface height (y value) is stored in the alpha channel.  The light reflection and refraction effects are a bit tricky. There is one reflection, one refraction and one depth texture. All of them are the size of the screen. The reflection texture is rendered via FBO with a clip plane (flat water surface) applied. Everything above the water is rendered twice. First time the normal way, second time upside down (the y-coordinates of the objects are scaled by -1 and then translated by the water level multiplied by 2) thus rendered to the region under the water. The refraction and depth textures are just copies of the color and depth buffers after rendering the whole (unclipped) scene for the third time.  It takes only one line (fragment and light positions) plane (flat water surface) optimized intersection test to calculate the coordinates to fetch the NBM texture and one dot product of the NBM normal and light direction to calculate caustic at the time when lighting is performed.  Finally, the water surface 128x128 polygonal model is rendered with the water shader applied. In the vertex shader the vertex is vertically displaced according to the height read from the alpha channel of the NBM texture. In the fragment shader the reflection and refraction textures are fetched with a little offset depending on the NBM normal's xz coordinates.  You can change the water level with the +/- keyboard keys and add waves by clicking on the water surface with the left mouse button. |

///////////////////////////////////////////Water Waves////////////////////////////////////////////////////////////////////



1. poolsky.vs

#version 120

void main()

{

gl\_TexCoord[0].stp = vec3(gl\_Vertex.x, -gl\_Vertex.yz);

gl\_Position = gl\_ModelViewProjectionMatrix \* gl\_Vertex;

}

2>poolsky.fs

#version 120

uniform samplerCube PoolSkyCubeMap;

void main()

{

gl\_FragColor = textureCube(PoolSkyCubeMap, gl\_TexCoord[0].stp);

}

3>water.vs  
#version 120

uniform sampler2D WaterHeightMap;

varying vec3 Position;

void main()

{

gl\_TexCoord[0].st = vec2(gl\_Vertex.x \* 0.5 + 0.5, 0.5 - gl\_Vertex.z \* 0.5);

Position = gl\_Vertex.xyz;

Position.y += texture2D(WaterHeightMap, gl\_TexCoord[0].st).g;

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

}

4>water.fs

#version 120

uniform sampler2D WaterNormalMap;

uniform samplerCube PoolSkyCubeMap;

uniform vec3 LightPosition, CubeMapNormals[6], CameraPosition;

varying vec3 Position;

vec3 IntersectCubeMap(vec3 Position, vec3 Direction)

{

vec3 Point;

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

{

float NdotR = -dot(CubeMapNormals[i], Direction);

if(NdotR > 0.0)

{

float Distance = (dot(CubeMapNormals[i], Position) + 1.0) / NdotR;

if(Distance > -0.03)

{

Point = Direction \* Distance + Position;

if(Point.x > -1.001 && Point.x < 1.001 && Point.y > -1.001 && Point.y < 1.001 && Point.z > -1.001 && Point.z < 1.001)

{

break;

}

}

}

}

return vec3(Point.x, -Point.yz);

}

void main()

{

vec3 Normal = normalize(texture2D(WaterNormalMap, gl\_TexCoord[0].st).rgb);

vec3 Direction = normalize(Position - CameraPosition);

if(CameraPosition.y > 0)

{

vec3 ReflectedColor = textureCube(PoolSkyCubeMap, IntersectCubeMap(Position, reflect(Direction, Normal))).rgb;

vec3 RefractedColor = textureCube(PoolSkyCubeMap, IntersectCubeMap(Position, refract(Direction, Normal, 0.750395))).rgb;

vec3 LightDirectionReflected = reflect(normalize(Position - LightPosition), Normal);

float Specular = pow(max(-dot(Direction, LightDirectionReflected), 0.0), 128);

gl\_FragColor.rgb = mix(ReflectedColor, RefractedColor, -dot(Normal, Direction)) + Specular;

}

else

{

Normal = -Normal;

vec3 ReflectedColor = textureCube(PoolSkyCubeMap, IntersectCubeMap(Position, reflect(Direction, Normal))).rgb;

vec3 DirectionRefracted = refract(Direction, Normal, 1.332631);

if(DirectionRefracted.x == 0.0 && DirectionRefracted.y == 0.0 && DirectionRefracted.z == 0.0)

{

gl\_FragColor.rgb = ReflectedColor;

}

else

{

vec3 RefractedColor = textureCube(PoolSkyCubeMap, IntersectCubeMap(Position, DirectionRefracted)).rgb;

gl\_FragColor.rgb = mix(ReflectedColor, RefractedColor, -dot(Normal, Direction));

}

}

}

5>wateradddrop.vs

#version 120

void main()

{

gl\_TexCoord[0] = gl\_Vertex;

gl\_Position = gl\_Vertex \* 2.0 - 1.0;

}

6>wateradddrop.fs

#version 120

uniform sampler2D WaterHeightMap;

uniform float DropRadius;

uniform vec2 Position;

void main()

{

vec2 vh = texture2D(WaterHeightMap, gl\_TexCoord[0].st).rg;

float d = distance(gl\_TexCoord[0].st, Position);

gl\_FragColor = vec4(vh.r, vh.g - 4.0f \* max(DropRadius - d, 0.0), 0.0, 0.0);

}

7>waterheightmap.vs

#version 120

void main()

{

gl\_TexCoord[0] = gl\_Vertex;

gl\_Position = gl\_Vertex \* 2.0 - 1.0;

}

1. waterheightmap.fs

#version 120

uniform sampler2D WaterHeightMap;

uniform float ODWHMR;

void main()

{

vec2 vh = texture2D(WaterHeightMap, gl\_TexCoord[0].st).rg;

float force = 0.0;

force += 0.707107 \* (texture2D(WaterHeightMap, gl\_TexCoord[0].st - vec2(ODWHMR, ODWHMR)).g - vh.g);

force += texture2D(WaterHeightMap, gl\_TexCoord[0].st - vec2(0.0, ODWHMR)).g - vh.g;

force += 0.707107 \* (texture2D(WaterHeightMap, gl\_TexCoord[0].st + vec2(ODWHMR, -ODWHMR)).g - vh.g);

force += texture2D(WaterHeightMap, gl\_TexCoord[0].st - vec2(ODWHMR, 0.0)).g - vh.g;

force += texture2D(WaterHeightMap, gl\_TexCoord[0].st + vec2(ODWHMR, 0.0)).g - vh.g;

force += 0.707107 \* (texture2D(WaterHeightMap, gl\_TexCoord[0].st + vec2(-ODWHMR, ODWHMR)).g - vh.g);

force += texture2D(WaterHeightMap, gl\_TexCoord[0].st + vec2(0.0, ODWHMR)).g - vh.g;

force += 0.707107 \* (texture2D(WaterHeightMap, gl\_TexCoord[0].st + vec2(ODWHMR, ODWHMR)).g - vh.g);

force \*= 0.125;

vh.r += force;

vh.g += vh.r;

vh.g \*= 0.99;

gl\_FragColor = vec4(vh, 0.0, 0.0);

}

1. waternormalmap.vs

#version 120

void main()

{

gl\_TexCoord[0] = gl\_Vertex;

gl\_Position = gl\_Vertex \* 2.0 - 1.0;

}

1. waternormalmap.fs

#version 120

uniform sampler2D WaterHeightMap;

uniform float ODWNMR, WMSDWNMRM2;

void main()

{

float y[4];

y[0] = texture2D(WaterHeightMap, gl\_TexCoord[0].st + vec2(ODWNMR, 0.0)).g;

y[1] = texture2D(WaterHeightMap, gl\_TexCoord[0].st + vec2(0.0, ODWNMR)).g;

y[2] = texture2D(WaterHeightMap, gl\_TexCoord[0].st - vec2(ODWNMR, 0.0)).g;

y[3] = texture2D(WaterHeightMap, gl\_TexCoord[0].st - vec2(0.0, ODWNMR)).g;

vec3 Normal = normalize(vec3(y[2] - y[0], WMSDWNMRM2, y[1] - y[3]));

gl\_FragColor = vec4(Normal, 1.0);

}

////////////////////////////////////////Source/////////////////////////////

#define WMR 128 // water mesh resolution

#define WHMR 128 // water height map resolution

#define WNMR 256 // water normal map resolution

class COpenGLRenderer

{

protected:

int Width, Height;

mat3x3 NormalMatrix;

mat4x4 ModelMatrix, ViewMatrix, ViewMatrixInverse, ProjectionMatrix, ProjectionBiasMatrixInverse;

protected:

CTexture PoolSkyCubeMap;

GLuint WaterHeightMaps[2], WHMID, WaterNormalMap, PoolSkyVBO, WaterVBO, FBO;

CShaderProgram WaterAddDropProgram, WaterHeightMapProgram, WaterNormalMapProgram, PoolSkyProgram, WaterProgram;

int QuadsVerticesCount;

public:

bool WireFrame, Pause;

float DropRadius;

public:

CString Text;

public:

COpenGLRenderer();

~COpenGLRenderer();

bool Init();

void Render(float FrameTime);

void Resize(int Width, int Height);

void Destroy();

void AddDrop(float x, float y, float DropRadius);

void AddDropByMouseClick(int x, int y);

};

#include "opengl\_21\_tutorials\_win32\_framework.h"

// ----------------------------------------------------------------------------------------------------------------------------

CBuffer::CBuffer()

{

SetDefaults();

}

CBuffer::~CBuffer()

{

Empty();

}

void CBuffer::AddData(void \*Data, int DataSize)

{

int Remaining = BufferSize - Position;

if(DataSize > Remaining)

{

BYTE \*OldBuffer = Buffer;

int OldBufferSize = BufferSize;

int Needed = DataSize - Remaining;

BufferSize += Needed > BUFFER\_SIZE\_INCREMENT ? Needed : BUFFER\_SIZE\_INCREMENT;

Buffer = new BYTE[BufferSize];

memcpy(Buffer, OldBuffer, OldBufferSize);

delete [] OldBuffer;

}

memcpy(Buffer + Position, Data, DataSize);

Position += DataSize;

}

void CBuffer::Empty()

{

delete [] Buffer;

SetDefaults();

}

void \*CBuffer::GetData()

{

return Buffer;

}

int CBuffer::GetDataSize()

{

return Position;

}

void CBuffer::SetDefaults()

{

Buffer = NULL;

BufferSize = 0;

Position = 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int gl\_max\_texture\_size = 0, gl\_max\_texture\_max\_anisotropy\_ext = 0;

// ----------------------------------------------------------------------------------------------------------------------------

CTexture::CTexture()

{

Texture = 0;

}

CTexture::~CTexture()

{

}

CTexture::operator GLuint ()

{

return Texture;

}

bool CTexture::LoadTexture2D(char \*FileName)

{

CString DirectoryFileName = ModuleDirectory + FileName;

int Width, Height, BPP;

FIBITMAP \*dib = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileName);

FreeImage\_Unload(dib);

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_2D, Texture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_2D, GL\_GENERATE\_MIPMAP, GL\_TRUE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib));

glBindTexture(GL\_TEXTURE\_2D, 0);

FreeImage\_Unload(dib);

return true;

}

bool CTexture::LoadTextureCubeMap(char \*\*FileNames)

{

int Width, Height, BPP;

FIBITMAP \*dib[6];

bool Error = false;

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

{

CString DirectoryFileName = ModuleDirectory + FileNames[i];

dib[i] = GetBitmap(DirectoryFileName, Width, Height, BPP);

if(dib[i] == NULL)

{

ErrorLog.Append("Error loading texture " + DirectoryFileName + "!\r\n");

Error = true;

}

}

if(Error)

{

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

GLenum Format = 0;

if(BPP == 32) Format = GL\_BGRA;

if(BPP == 24) Format = GL\_BGR;

if(Format == 0)

{

ErrorLog.Append("Unsupported texture format (%s)!\r\n", FileNames[5]);

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

{

FreeImage\_Unload(dib[i]);

}

return false;

}

Destroy();

glGenTextures(1, &Texture);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, Texture);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_GENERATE\_MIPMAP, GL\_TRUE);

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

{

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGBA8, Width, Height, 0, Format, GL\_UNSIGNED\_BYTE, FreeImage\_GetBits(dib[i]));

}

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

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

{

FreeImage\_Unload(dib[i]);

}

return true;

}

void CTexture::Destroy()

{

glDeleteTextures(1, &Texture);

Texture = 0;

}

FIBITMAP \*CTexture::GetBitmap(char \*FileName, int &Width, int &Height, int &BPP)

{

FREE\_IMAGE\_FORMAT fif = FreeImage\_GetFileType(FileName);

if(fif == FIF\_UNKNOWN)

{

fif = FreeImage\_GetFIFFromFilename(FileName);

}

if(fif == FIF\_UNKNOWN)

{

return NULL;

}

FIBITMAP \*dib = NULL;

if(FreeImage\_FIFSupportsReading(fif))

{

dib = FreeImage\_Load(fif, FileName);

}

if(dib != NULL)

{

int OriginalWidth = FreeImage\_GetWidth(dib);

int OriginalHeight = FreeImage\_GetHeight(dib);

Width = OriginalWidth;

Height = OriginalHeight;

if(Width == 0 || Height == 0)

{

FreeImage\_Unload(dib);

return NULL;

}

BPP = FreeImage\_GetBPP(dib);

if(Width > gl\_max\_texture\_size) Width = gl\_max\_texture\_size;

if(Height > gl\_max\_texture\_size) Height = gl\_max\_texture\_size;

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

Width = 1 << (int)floor((log((float)Width) / log(2.0f)) + 0.5f);

Height = 1 << (int)floor((log((float)Height) / log(2.0f)) + 0.5f);

}

if(Width != OriginalWidth || Height != OriginalHeight)

{

FIBITMAP \*rdib = FreeImage\_Rescale(dib, Width, Height, FILTER\_BICUBIC);

FreeImage\_Unload(dib);

dib = rdib;

}

}

return dib;

}

// ----------------------------------------------------------------------------------------------------------------------------

CShaderProgram::CShaderProgram()

{

SetDefaults();

}

CShaderProgram::~CShaderProgram()

{

}

CShaderProgram::operator GLuint ()

{

return Program;

}

bool CShaderProgram::Load(char \*VertexShaderFileName, char \*FragmentShaderFileName)

{

bool Error = false;

Destroy();

Error |= ((VertexShader = LoadShader(VertexShaderFileName, GL\_VERTEX\_SHADER)) == 0);

Error |= ((FragmentShader = LoadShader(FragmentShaderFileName, GL\_FRAGMENT\_SHADER)) == 0);

if(Error)

{

Destroy();

return false;

}

Program = glCreateProgram();

glAttachShader(Program, VertexShader);

glAttachShader(Program, FragmentShader);

glLinkProgram(Program);

int LinkStatus;

glGetProgramiv(Program, GL\_LINK\_STATUS, &LinkStatus);

if(LinkStatus == GL\_FALSE)

{

ErrorLog.Append("Error linking program (%s, %s)!\r\n", VertexShaderFileName, FragmentShaderFileName);

int InfoLogLength = 0;

glGetProgramiv(Program, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetProgramInfoLog(Program, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

Destroy();

return false;

}

return true;

}

void CShaderProgram::Destroy()

{

glDetachShader(Program, VertexShader);

glDetachShader(Program, FragmentShader);

glDeleteShader(VertexShader);

glDeleteShader(FragmentShader);

glDeleteProgram(Program);

delete [] UniformLocations;

delete [] AttribLocations;

SetDefaults();

}

GLuint CShaderProgram::LoadShader(char \*FileName, GLenum Type)

{

CString DirectoryFileName = ModuleDirectory + FileName;

FILE \*File;

if(fopen\_s(&File, DirectoryFileName, "rb") != 0)

{

ErrorLog.Append("Error loading file " + DirectoryFileName + "!\r\n");

return 0;

}

fseek(File, 0, SEEK\_END);

long Size = ftell(File);

fseek(File, 0, SEEK\_SET);

char \*Source = new char[Size + 1];

fread(Source, 1, Size, File);

fclose(File);

Source[Size] = 0;

GLuint Shader = glCreateShader(Type);

glShaderSource(Shader, 1, (const char\*\*)&Source, NULL);

delete [] Source;

glCompileShader(Shader);

int CompileStatus;

glGetShaderiv(Shader, GL\_COMPILE\_STATUS, &CompileStatus);

if(CompileStatus == GL\_FALSE)

{

ErrorLog.Append("Error compiling shader %s!\r\n", FileName);

int InfoLogLength = 0;

glGetShaderiv(Shader, GL\_INFO\_LOG\_LENGTH, &InfoLogLength);

if(InfoLogLength > 0)

{

char \*InfoLog = new char[InfoLogLength];

int CharsWritten = 0;

glGetShaderInfoLog(Shader, InfoLogLength, &CharsWritten, InfoLog);

ErrorLog.Append(InfoLog);

delete [] InfoLog;

}

glDeleteShader(Shader);

return 0;

}

return Shader;

}

void CShaderProgram::SetDefaults()

{

VertexShader = 0;

FragmentShader = 0;

Program = 0;

UniformLocations = NULL;

AttribLocations = NULL;

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera::CCamera()

{

ViewMatrix = NULL;

ViewMatrixInverse = NULL;

X = vec3(1.0f, 0.0f, 0.0f);

Y = vec3(0.0f, 1.0f, 0.0f);

Z = vec3(0.0f, 0.0f, 1.0f);

Position = vec3(0.0f, 0.0f, 5.0f);

Reference = vec3(0.0f, 0.0f, 0.0f);

}

CCamera::~CCamera()

{

}

void CCamera::Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference)

{

this->Position = Position;

this->Reference = Reference;

Z = normalize(Position - Reference);

X = normalize(cross(vec3(0.0f, 1.0f, 0.0f), Z));

Y = cross(Z, X);

if(!RotateAroundReference)

{

this->Reference = this->Position;

this->Position += Z \* 0.05f;

}

CalculateViewMatrix();

}

void CCamera::Move(const vec3 &Movement)

{

Position += Movement;

Reference += Movement;

CalculateViewMatrix();

}

vec3 CCamera::OnKeys(BYTE Keys, float FrameTime)

{

float Speed = 5.0f;

if(Keys & 0x40) Speed \*= 2.0f;

if(Keys & 0x80) Speed \*= 0.5f;

float Distance = Speed \* FrameTime;

vec3 Up(0.0f, 1.0f, 0.0f);

vec3 Right = X;

vec3 Forward = cross(Up, Right);

Up \*= Distance;

Right \*= Distance;

Forward \*= Distance;

vec3 Movement;

if(Keys & 0x01) Movement += Forward;

if(Keys & 0x02) Movement -= Forward;

if(Keys & 0x04) Movement -= Right;

if(Keys & 0x08) Movement += Right;

if(Keys & 0x10) Movement += Up;

if(Keys & 0x20) Movement -= Up;

return Movement;

}

void CCamera::OnMouseMove(int dx, int dy)

{

float Sensitivity = 0.25f;

Position -= Reference;

if(dx != 0)

{

float DeltaX = (float)dx \* Sensitivity;

X = rotate(X, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Y = rotate(Y, DeltaX, vec3(0.0f, 1.0f, 0.0f));

Z = rotate(Z, DeltaX, vec3(0.0f, 1.0f, 0.0f));

}

if(dy != 0)

{

float DeltaY = (float)dy \* Sensitivity;

Y = rotate(Y, DeltaY, X);

Z = rotate(Z, DeltaY, X);

if(Y.y < 0.0f)

{

Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);

Y = cross(Z, X);

}

}

Position = Reference + Z \* length(Position);

CalculateViewMatrix();

}

void CCamera::OnMouseWheel(float zDelta)

{

Position -= Reference;

if(zDelta < 0 && length(Position) < 500.0f)

{

Position += Position \* 0.1f;

}

if(zDelta > 0 && length(Position) > 0.05f)

{

Position -= Position \* 0.1f;

}

Position += Reference;

CalculateViewMatrix();

}

void CCamera::SetViewMatrixPointer(float \*ViewMatrix, float \*ViewMatrixInverse)

{

this->ViewMatrix = (mat4x4\*)ViewMatrix;

this->ViewMatrixInverse = (mat4x4\*)ViewMatrixInverse;

CalculateViewMatrix();

}

void CCamera::CalculateViewMatrix()

{

if(ViewMatrix != NULL)

{

\*ViewMatrix = mat4x4(X.x, Y.x, Z.x, 0.0f, X.y, Y.y, Z.y, 0.0f, X.z, Y.z, Z.z, 0.0f, -dot(X, Position), -dot(Y, Position), -dot(Z, Position), 1.0f);

if(ViewMatrixInverse != NULL)

{

\*ViewMatrixInverse = inverse(\*ViewMatrix);

}

}

}

// ----------------------------------------------------------------------------------------------------------------------------

CCamera Camera;

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer::COpenGLRenderer()

{

WHMID = 0;

WireFrame = false;

Pause = false;

DropRadius = 4.0f / 128.0f;

Camera.SetViewMatrixPointer(&ViewMatrix, &ViewMatrixInverse);

}

COpenGLRenderer::~COpenGLRenderer()

{

}

bool COpenGLRenderer::Init()

{

// ------------------------------------------------------------------------------------------------------------------------

bool Error = false;

// ------------------------------------------------------------------------------------------------------------------------

if(!GLEW\_ARB\_texture\_non\_power\_of\_two)

{

ErrorLog.Append("GL\_ARB\_texture\_non\_power\_of\_two not supported!\r\n");

Error = true;

}

if(!GLEW\_ARB\_texture\_float)

{

ErrorLog.Append("GL\_ARB\_texture\_float not supported!\r\n");

Error = true;

}

if(!GLEW\_EXT\_framebuffer\_object)

{

ErrorLog.Append("GL\_EXT\_framebuffer\_object not supported!\r\n");

Error = true;

}

// ------------------------------------------------------------------------------------------------------------------------

char \*PoolSkyCubeMapFileNames[] = {"pool\\right.jpg", "pool\\left.jpg", "pool\\bottom.jpg", "pool\\top.jpg", "pool\\front.jpg", "pool\\back.jpg"};

Error |= !PoolSkyCubeMap.LoadTextureCubeMap(PoolSkyCubeMapFileNames);

// ------------------------------------------------------------------------------------------------------------------------

Error |= !WaterAddDropProgram.Load("wateradddrop.vs", "wateradddrop.fs");

Error |= !WaterHeightMapProgram.Load("waterheightmap.vs", "waterheightmap.fs");

Error |= !WaterNormalMapProgram.Load("waternormalmap.vs", "waternormalmap.fs");

Error |= !PoolSkyProgram.Load("poolsky.vs", "poolsky.fs");

Error |= !WaterProgram.Load("water.vs", "water.fs");

// ------------------------------------------------------------------------------------------------------------------------

if(Error)

{

return false;

}

// ------------------------------------------------------------------------------------------------------------------------

vec3 LightPosition = vec3(0.0f, 5.5f, -9.5f);

vec3 CubeMapNormals[6] = {

vec3(-1.0f, 0.0f, 0.0f),

vec3(1.0f, 0.0f, 0.0f),

vec3(0.0f, -1.0f, 0.0f),

vec3(0.0f, 1.0f, 0.0f),

vec3(0.0f, 0.0f, -1.0f),

vec3(0.0f, 0.0f, 1.0f),

};

// ------------------------------------------------------------------------------------------------------------------------

glUseProgram(WaterHeightMapProgram);

glUniform1f(glGetUniformLocation(WaterHeightMapProgram, "ODWHMR"), 1.0f / (float)WHMR);

glUseProgram(0);

glUseProgram(WaterNormalMapProgram);

glUniform1f(glGetUniformLocation(WaterNormalMapProgram, "ODWNMR"), 1.0f / (float)WNMR);

glUniform1f(glGetUniformLocation(WaterNormalMapProgram, "WMSDWNMRM2"), 2.0f / (float)WNMR \* 2.0f);

glUseProgram(0);

glUseProgram(WaterProgram);

glUniform1i(glGetUniformLocation(WaterProgram, "WaterHeightMap"), 0);

glUniform1i(glGetUniformLocation(WaterProgram, "WaterNormalMap"), 1);

glUniform1i(glGetUniformLocation(WaterProgram, "PoolSkyCubeMap"), 2);

glUniform1f(glGetUniformLocation(WaterProgram, "ODWMS"), 1.0f / 2.0f);

glUniform3fv(glGetUniformLocation(WaterProgram, "LightPosition"), 1, &LightPosition);

glUniform3fv(glGetUniformLocation(WaterProgram, "CubeMapNormals"), 6, (float\*)CubeMapNormals);

glUseProgram(0);

// ------------------------------------------------------------------------------------------------------------------------

WaterAddDropProgram.UniformLocations = new GLuint[2];

WaterAddDropProgram.UniformLocations[0] = glGetUniformLocation(WaterAddDropProgram, "DropRadius");

WaterAddDropProgram.UniformLocations[1] = glGetUniformLocation(WaterAddDropProgram, "Position");

WaterProgram.UniformLocations = new GLuint[1];

WaterProgram.UniformLocations[0] = glGetUniformLocation(WaterProgram, "CameraPosition");

// ------------------------------------------------------------------------------------------------------------------------

glGenTextures(2, WaterHeightMaps);

vec4 \*Heights = new vec4[WHMR \* WHMR];

for(int i = 0; i < WHMR \* WHMR; i++)

{

Heights[i] = vec4(0.0f, 0.0f, 0.0f, 0.0f);

}

for(int i = 0; i < 2; i++)

{

glBindTexture(GL\_TEXTURE\_2D, WaterHeightMaps[i]);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA16F, WHMR, WHMR, 0, GL\_RGBA, GL\_FLOAT, Heights);

glGenerateMipmapEXT(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, 0);

}

delete [] Heights;

// ------------------------------------------------------------------------------------------------------------------------

glGenTextures(1, &WaterNormalMap);

vec4 \*Normals = new vec4[WNMR \* WNMR];

for(int i = 0; i < WNMR \* WNMR; i++)

{

Normals[i] = vec4(0.0f, 1.0f, 0.0f, 1.0f);

}

glBindTexture(GL\_TEXTURE\_2D, WaterNormalMap);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT, gl\_max\_texture\_max\_anisotropy\_ext);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA16F, WNMR, WNMR, 0, GL\_RGBA, GL\_FLOAT, Normals);

glGenerateMipmapEXT(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, 0);

delete [] Normals;

// ------------------------------------------------------------------------------------------------------------------------

glGenBuffers(1, &PoolSkyVBO);

float PoolSkyVertices[] =

{ // x, y, z, x, y, z, x, y, z, x, y, z

1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, -1.0f, // +X

-1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, // -X

-1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, // +Y

-1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, // -Y

1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, // +Z

-1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f // -Z

};

glBindBuffer(GL\_ARRAY\_BUFFER, PoolSkyVBO);

glBufferData(GL\_ARRAY\_BUFFER, 288, PoolSkyVertices, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

// ------------------------------------------------------------------------------------------------------------------------

glGenBuffers(1, &WaterVBO);

int WMRP1 = WMR + 1;

vec3 \*Vertices = new vec3[WMRP1 \* WMRP1];

float WMSDWMR = 2.0f / (float)WMR;

for(int y = 0; y <= WMR; y++)

{

for(int x = 0; x <= WMR; x++)

{

Vertices[WMRP1 \* y + x].x = x \* WMSDWMR - 1.0f;

Vertices[WMRP1 \* y + x].y = 0.0f;

Vertices[WMRP1 \* y + x].z = 1.0f - y \* WMSDWMR;

}

}

CBuffer Quads;

for(int y = 0; y < WMR; y++)

{

int yp1 = y + 1;

for(int x = 0; x < WMR; x++)

{

int xp1 = x + 1;

int a = WMRP1 \* y + x;

int b = WMRP1 \* y + xp1;

int c = WMRP1 \* yp1 + xp1;

int d = WMRP1 \* yp1 + x;

Quads.AddData(&Vertices[a], 12);

Quads.AddData(&Vertices[b], 12);

Quads.AddData(&Vertices[c], 12);

Quads.AddData(&Vertices[d], 12);

}

}

glBindBuffer(GL\_ARRAY\_BUFFER, WaterVBO);

glBufferData(GL\_ARRAY\_BUFFER, Quads.GetDataSize(), Quads.GetData(), GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

QuadsVerticesCount = Quads.GetDataSize() / 12;

Quads.Empty();

delete [] Vertices;

// ------------------------------------------------------------------------------------------------------------------------

glGenFramebuffersEXT(1, &FBO);

// ------------------------------------------------------------------------------------------------------------------------

Camera.Look(vec3(0.0f, 1.0f, 2.5f), vec3(0.0f, -0.5f, 0.0f), true);

// ------------------------------------------------------------------------------------------------------------------------

srand(GetTickCount());

// ------------------------------------------------------------------------------------------------------------------------

return true;

}

void COpenGLRenderer::Render(float FrameTime)

{

// add drops --------------------------------------------------------------------------------------------------------------

if(!Pause)

{

static DWORD LastTime = GetTickCount();

DWORD Time = GetTickCount();

if(Time - LastTime > 100)

{

LastTime = Time;

AddDrop(2.0f \* (float)rand() / (float)RAND\_MAX - 1.0f, 1.0f - 2.0f \* (float)rand() / (float)RAND\_MAX, 4.0f / 128.0f \* (float)rand() / (float)RAND\_MAX);

}

}

// update water surface ---------------------------------------------------------------------------------------------------

static DWORD LastTime = GetTickCount();

DWORD Time = GetTickCount();

if(Time - LastTime >= 16)

{

LastTime = Time;

// update water height map --------------------------------------------------------------------------------------------

glViewport(0, 0, WHMR, WHMR);

GLuint whmid = (WHMID + 1) % 2;

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, WaterHeightMaps[whmid], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, WaterHeightMaps[WHMID]);

glUseProgram(WaterHeightMapProgram);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

glBindTexture(GL\_TEXTURE\_2D, WaterHeightMaps[whmid]);

glGenerateMipmapEXT(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, 0);

++WHMID %= 2;

// update water normal map --------------------------------------------------------------------------------------------

glViewport(0, 0, WNMR, WNMR);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, WaterNormalMap, 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, WaterHeightMaps[WHMID]);

glUseProgram(WaterNormalMapProgram);

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

glBindTexture(GL\_TEXTURE\_2D, WaterNormalMap);

glGenerateMipmapEXT(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, 0);

}

// render pool sky mesh ---------------------------------------------------------------------------------------------------

glViewport(0, 0, Width, Height);

glMatrixMode(GL\_PROJECTION);

glLoadMatrixf(&ProjectionMatrix);

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

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

glMatrixMode(GL\_MODELVIEW);

glLoadMatrixf(&ViewMatrix);

if(WireFrame)

{

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

}

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, PoolSkyCubeMap);

glUseProgram(PoolSkyProgram);

glBindBuffer(GL\_ARRAY\_BUFFER, PoolSkyVBO);

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(3, GL\_FLOAT, 12, (void\*)0);

glDrawArrays(GL\_QUADS, 0, 24);

glDisableClientState(GL\_VERTEX\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

glDisable(GL\_CULL\_FACE);

// render water surface ---------------------------------------------------------------------------------------------------

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, WaterHeightMaps[WHMID]);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, WaterNormalMap);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_CUBE\_MAP, PoolSkyCubeMap);

glUseProgram(WaterProgram);

glUniform3fv(WaterProgram.UniformLocations[0], 1, &Camera.Position);

glBindBuffer(GL\_ARRAY\_BUFFER, WaterVBO);

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(3, GL\_FLOAT, 12, (void\*)0);

glDrawArrays(GL\_QUADS, 0, QuadsVerticesCount);

glDisableClientState(GL\_VERTEX\_ARRAY);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glUseProgram(0);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, 0);

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, 0);

if(WireFrame)

{

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);

}

glDisable(GL\_DEPTH\_TEST);

}

void COpenGLRenderer::Resize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

ProjectionMatrix = perspective(45.0f, (float)Width / (float)Height, 0.125f, 512.0f);

ProjectionBiasMatrixInverse = inverse(ProjectionMatrix) \* BiasMatrixInverse;

}

void COpenGLRenderer::Destroy()

{

PoolSkyCubeMap.Destroy();

WaterAddDropProgram.Destroy();

WaterHeightMapProgram.Destroy();

WaterNormalMapProgram.Destroy();

PoolSkyProgram.Destroy();

WaterProgram.Destroy();

glDeleteTextures(2, WaterHeightMaps);

glDeleteTextures(1, &WaterNormalMap);

glDeleteBuffers(1, &PoolSkyVBO);

glDeleteBuffers(1, &WaterVBO);

if(GLEW\_EXT\_framebuffer\_object)

{

glDeleteFramebuffersEXT(1, &FBO);

}

}

void COpenGLRenderer::AddDrop(float x, float y, float DropRadius)

{

if(x >= -1.0f && x <= 1.0f && y >= -1.0f && y <= 1.0f)

{

glViewport(0, 0, WMR, WMR);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, FBO);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_COLOR\_ATTACHMENT0\_EXT, GL\_TEXTURE\_2D, WaterHeightMaps[(WHMID + 1) % 2], 0);

glFramebufferTexture2DEXT(GL\_FRAMEBUFFER\_EXT, GL\_DEPTH\_ATTACHMENT\_EXT, GL\_TEXTURE\_2D, 0, 0);

glBindTexture(GL\_TEXTURE\_2D, WaterHeightMaps[WHMID]);

glUseProgram(WaterAddDropProgram);

glUniform1f(WaterAddDropProgram.UniformLocations[0], DropRadius);

glUniform2fv(WaterAddDropProgram.UniformLocations[1], 1, &vec2(x \* 0.5f + 0.5f, 0.5f - y \* 0.5f));

glBegin(GL\_QUADS);

glVertex2f(0.0f, 0.0f);

glVertex2f(1.0f, 0.0f);

glVertex2f(1.0f, 1.0f);

glVertex2f(0.0f, 1.0f);

glEnd();

glUseProgram(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

glBindFramebufferEXT(GL\_FRAMEBUFFER\_EXT, 0);

++WHMID %= 2;

}

}

void COpenGLRenderer::AddDropByMouseClick(int x, int y)

{

float s = (float)x / (float)(Width - 1);

float t = 1.0f - (float)y / (float)(Height - 1);

vec4 Position = ViewMatrixInverse \* (ProjectionBiasMatrixInverse \* vec4(s, t, 0.5f, 1.0f));

Position /= Position.w;

vec3 Ray = normalize(\*(vec3\*)&Position - Camera.Position);

vec3 Normal = vec3(0.0f, 1.0f, 0.0f);

float D = -dot(Normal, vec3(0.0f, 0.0f, 0.0f));

float NdotR = -dot(Normal, Ray);

if(NdotR != 0.0f)

{

float Distance = (dot(Normal, Camera.Position) + D) / NdotR;

if(Distance > 0.0f)

{

vec3 Position = Ray \* Distance + Camera.Position;

AddDrop(Position.x, Position.z, DropRadius);

}

}

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLRenderer OpenGLRenderer;

// ----------------------------------------------------------------------------------------------------------------------------

CString ModuleDirectory, ErrorLog;

// ----------------------------------------------------------------------------------------------------------------------------

void GetModuleDirectory()

{

char \*moduledirectory = new char[256];

GetModuleFileName(GetModuleHandle(NULL), moduledirectory, 256);

\*(strrchr(moduledirectory, '\\') + 1) = 0;

ModuleDirectory = moduledirectory;

delete [] moduledirectory;

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView::COpenGLView()

{

}

COpenGLView::~COpenGLView()

{

}

bool COpenGLView::Init(HINSTANCE hInstance, char \*Title, int Width, int Height, int Samples)

{

this->Title = Title;

this->Width = Width;

this->Height = Height;

WNDCLASSEX WndClassEx;

memset(&WndClassEx, 0, sizeof(WNDCLASSEX));

WndClassEx.cbSize = sizeof(WNDCLASSEX);

WndClassEx.style = CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

WndClassEx.lpfnWndProc = WndProc;

WndClassEx.hInstance = hInstance;

WndClassEx.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

WndClassEx.hCursor = LoadCursor(NULL, IDC\_ARROW);

WndClassEx.lpszClassName = "Win32OpenGLWindowClass";

if(RegisterClassEx(&WndClassEx) == 0)

{

ErrorLog.Set("RegisterClassEx failed!");

return false;

}

DWORD Style = WS\_OVERLAPPEDWINDOW | WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN;

hWnd = CreateWindowEx(WS\_EX\_APPWINDOW, WndClassEx.lpszClassName, Title, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL);

if(hWnd == NULL)

{

ErrorLog.Set("CreateWindowEx failed!");

return false;

}

HDC hDC = GetDC(hWnd);

if(hDC == NULL)

{

ErrorLog.Set("GetDC failed!");

return false;

}

PIXELFORMATDESCRIPTOR pfd;

memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));

pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);

pfd.nVersion = 1;

pfd.dwFlags = PFD\_DRAW\_TO\_WINDOW | PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER;

pfd.iPixelType = PFD\_TYPE\_RGBA;

pfd.cColorBits = 32;

pfd.cDepthBits = 24;

pfd.iLayerType = PFD\_MAIN\_PLANE;

int PixelFormat = ChoosePixelFormat(hDC, &pfd);

if(PixelFormat == 0)

{

ErrorLog.Set("ChoosePixelFormat failed!");

return false;

}

static int MSAAPixelFormat = 0;

if(SetPixelFormat(hDC, MSAAPixelFormat == 0 ? PixelFormat : MSAAPixelFormat, &pfd) == FALSE)

{

ErrorLog.Set("SetPixelFormat failed!");

return false;

}

hGLRC = wglCreateContext(hDC);

if(hGLRC == NULL)

{

ErrorLog.Set("wglCreateContext failed!");

return false;

}

if(wglMakeCurrent(hDC, hGLRC) == FALSE)

{

ErrorLog.Set("wglMakeCurrent failed!");

return false;

}

if(glewInit() != GLEW\_OK)

{

ErrorLog.Set("glewInit failed!");

return false;

}

if(!GLEW\_VERSION\_2\_1)

{

ErrorLog.Set("OpenGL 2.1 not supported!");

return false;

}

if(MSAAPixelFormat == 0 && Samples > 0)

{

if(GLEW\_ARB\_multisample && WGLEW\_ARB\_pixel\_format)

{

while(Samples > 0)

{

UINT NumFormats = 0;

int PFAttribs[] =

{

WGL\_DRAW\_TO\_WINDOW\_ARB, GL\_TRUE,

WGL\_SUPPORT\_OPENGL\_ARB, GL\_TRUE,

WGL\_DOUBLE\_BUFFER\_ARB, GL\_TRUE,

WGL\_PIXEL\_TYPE\_ARB, WGL\_TYPE\_RGBA\_ARB,

WGL\_COLOR\_BITS\_ARB, 32,

WGL\_DEPTH\_BITS\_ARB, 24,

WGL\_ACCELERATION\_ARB, WGL\_FULL\_ACCELERATION\_ARB,

WGL\_SAMPLE\_BUFFERS\_ARB, GL\_TRUE,

WGL\_SAMPLES\_ARB, Samples,

0

};

if(wglChoosePixelFormatARB(hDC, PFAttribs, NULL, 1, &MSAAPixelFormat, &NumFormats) == TRUE && NumFormats > 0) break;

Samples--;

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

UnregisterClass(WndClassEx.lpszClassName, hInstance);

return Init(hInstance, Title, Width, Height, Samples);

}

else

{

Samples = 0;

}

}

this->Samples = Samples;

GetModuleDirectory();

glGetIntegerv(GL\_MAX\_TEXTURE\_SIZE, &gl\_max\_texture\_size);

if(GLEW\_EXT\_texture\_filter\_anisotropic)

{

glGetIntegerv(GL\_MAX\_TEXTURE\_MAX\_ANISOTROPY\_EXT, &gl\_max\_texture\_max\_anisotropy\_ext);

}

if(WGLEW\_EXT\_swap\_control)

{

wglSwapIntervalEXT(0);

}

return OpenGLRenderer.Init();

}

void COpenGLView::Show(bool Maximized)

{

RECT dRect, wRect, cRect;

GetWindowRect(GetDesktopWindow(), &dRect);

GetWindowRect(hWnd, &wRect);

GetClientRect(hWnd, &cRect);

wRect.right += Width - cRect.right;

wRect.bottom += Height - cRect.bottom;

wRect.right -= wRect.left;

wRect.bottom -= wRect.top;

wRect.left = dRect.right / 2 - wRect.right / 2;

wRect.top = dRect.bottom / 2 - wRect.bottom / 2;

MoveWindow(hWnd, wRect.left, wRect.top, wRect.right, wRect.bottom, FALSE);

ShowWindow(hWnd, Maximized ? SW\_SHOWMAXIMIZED : SW\_SHOWNORMAL);

}

void COpenGLView::MessageLoop()

{

MSG Msg;

while(GetMessage(&Msg, NULL, 0, 0) > 0)

{

TranslateMessage(&Msg);

DispatchMessage(&Msg);

}

}

void COpenGLView::Destroy()

{

if(GLEW\_VERSION\_2\_1)

{

OpenGLRenderer.Destroy();

}

wglDeleteContext(hGLRC);

DestroyWindow(hWnd);

}

void COpenGLView::OnKeyDown(UINT Key)

{

switch(Key)

{

case VK\_F1:

OpenGLRenderer.WireFrame = !OpenGLRenderer.WireFrame;

break;

case '1':

OpenGLRenderer.DropRadius = 4.0f / 256.0f;

break;

case '2':

OpenGLRenderer.DropRadius = 4.0f / 128.0f;

break;

case '3':

OpenGLRenderer.DropRadius = 4.0f / 64.0f;

break;

case '4':

OpenGLRenderer.DropRadius = 4.0f / 32.0f;

break;

case '5':

OpenGLRenderer.DropRadius = 4.0f / 16.0f;

break;

case VK\_SPACE:

OpenGLRenderer.Pause = !OpenGLRenderer.Pause;

break;

}

}

void COpenGLView::OnLButtonDown(int X, int Y)

{

OpenGLRenderer.AddDropByMouseClick(X, Y);

}

void COpenGLView::OnMouseMove(int X, int Y)

{

if(GetKeyState(VK\_RBUTTON) & 0x80)

{

Camera.OnMouseMove(LastX - X, LastY - Y);

LastX = X;

LastY = Y;

}

}

void COpenGLView::OnMouseWheel(short zDelta)

{

Camera.OnMouseWheel(zDelta);

}

void COpenGLView::OnPaint()

{

static DWORD LastFPSTime = GetTickCount(), LastFrameTime = LastFPSTime, FPS = 0;

PAINTSTRUCT ps;

HDC hDC = BeginPaint(hWnd, &ps);

DWORD Time = GetTickCount();

float FrameTime = (Time - LastFrameTime) \* 0.001f;

LastFrameTime = Time;

if(Time - LastFPSTime > 1000)

{

CString Text = Title;

if(OpenGLRenderer.Text[0] != 0)

{

Text.Append(" - " + OpenGLRenderer.Text);

}

Text.Append(" - %dx%d", Width, Height);

Text.Append(", ATF %dx", gl\_max\_texture\_max\_anisotropy\_ext);

Text.Append(", MSAA %dx", Samples);

Text.Append(", FPS: %d", FPS);

Text.Append(" - %s", glGetString(GL\_RENDERER));

SetWindowText(hWnd, Text);

LastFPSTime = Time;

FPS = 0;

}

else

{

FPS++;

}

BYTE Keys = 0x00;

if(GetKeyState('W') & 0x80) Keys |= 0x01;

if(GetKeyState('S') & 0x80) Keys |= 0x02;

if(GetKeyState('A') & 0x80) Keys |= 0x04;

if(GetKeyState('D') & 0x80) Keys |= 0x08;

if(GetKeyState('R') & 0x80) Keys |= 0x10;

if(GetKeyState('F') & 0x80) Keys |= 0x20;

if(GetKeyState(VK\_SHIFT) & 0x80) Keys |= 0x40;

if(GetKeyState(VK\_CONTROL) & 0x80) Keys |= 0x80;

if(Keys & 0x3F)

{

Camera.Move(Camera.OnKeys(Keys, FrameTime));

}

OpenGLRenderer.Render(FrameTime);

SwapBuffers(hDC);

EndPaint(hWnd, &ps);

InvalidateRect(hWnd, NULL, FALSE);

}

void COpenGLView::OnRButtonDown(int X, int Y)

{

LastX = X;

LastY = Y;

}

void COpenGLView::OnSize(int Width, int Height)

{

this->Width = Width;

this->Height = Height;

OpenGLRenderer.Resize(Width, Height);

}

// ----------------------------------------------------------------------------------------------------------------------------

COpenGLView OpenGLView;

// ----------------------------------------------------------------------------------------------------------------------------

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)

{

switch(uiMsg)

{

case WM\_CLOSE:

PostQuitMessage(0);

break;

case WM\_KEYDOWN:

OpenGLView.OnKeyDown((UINT)wParam);

break;

case WM\_LBUTTONDOWN:

OpenGLView.OnLButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_MOUSEMOVE:

OpenGLView.OnMouseMove(LOWORD(lParam), HIWORD(lParam));

break;

case 0x020A: // WM\_MOUSWHEEL

OpenGLView.OnMouseWheel(HIWORD(wParam));

break;

case WM\_PAINT:

OpenGLView.OnPaint();

break;

case WM\_RBUTTONDOWN:

OpenGLView.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));

break;

case WM\_SIZE:

OpenGLView.OnSize(LOWORD(lParam), HIWORD(lParam));

break;

default:

return DefWindowProc(hWnd, uiMsg, wParam, lParam);

}

return 0;

}

// ----------------------------------------------------------------------------------------------------------------------------

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine, int iShow)

{

char \*AppName = "Water waves GPU algorithm";

if(OpenGLView.Init(hInstance, AppName, 800, 600, 0))

{

OpenGLView.Show();

OpenGLView.MessageLoop();

}

else

{

MessageBox(NULL, ErrorLog, AppName, MB\_OK | MB\_ICONERROR);

}

OpenGLView.Destroy();

return 0;

}

**Comments (6)**

|  |
| --- |
| **zxx43**, December 19, 2014, 04:27 PM |
| In waternormalmap.fs ODWNMR is 1.0 / 256.0 and WMSDWNMRM2 is 4.0 / 256.0, why WMSDWNMRM2 is not ODWNMR \* 2.0? |

|  |
| --- |
| **Admin**, December 19, 2014, 04:57 PM |
| ODWNMR is TEXTURE SPACE distance between 2 neighboring pixels (or a pixel size) of the water normal map texture and is used to fetch heights from the water height map texture - this is a trick - it also fetches heights between pixels of the height map texture, because the resolution of the normal map is 2x higher. WMSDWNMRM2 is WORLD SPACE doubled distance between two neghboring pixels (or a pixel size) of the water normal map texture and is used to calculate normals of the water mesh. Heights and normals are stored in the water height and normal map textures in WORLD SPACE. WORLD SPACE size of the water mesh is 2.0 x 2.0. TEXTURE SPACE size of any texture is 1.0 x 1.0. |

|  |
| --- |
| **zxx43**, December 22, 2014, 09:52 AM |
| Why do we need to check in water.fs if the refract function returns vec3(0.0) only when the camera is under water? |

|  |
| --- |
| **Admin**, December 22, 2014, 10:15 AM |
| http://en.wikipedia.org/wiki/Refraction http://en.wikipedia.org/wiki/Refraction#mediaviewer/File:RefractionReflextion.svg  https://www.opengl.org/registry/doc/glspec21.20061201.pdf https://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf  The refract function is described in the chapter 8.4 of the OpenGL shading language 1.20.8.  genType refract(genType I, genType N, float eta) {  k = 1.0 - eta \* eta \* (1.0 - dot(N, I) \* dot(N, I));   if(k < 0.0)  {  return genType(0.0);  }  else  {  return eta \* I - (eta \* dot(N, I) + sqrt(k)) \* N;  } }  In case of total internal reflection (k < 0.0) the refract function returns genType(0.0). The condition k < 0.0 can be true only when eta > 1.0, only when the camera is under water. |

|  |
| --- |
| **zxx43**, December 26, 2014, 11:04 AM |
| How is it possible that waves are spreading, which equations did you use and which informations are stored in the components of the water height map texture? |

|  |
| --- |
| **Admin**, December 26, 2014, 12:03 PM |
| Everything is in the code - in the "update water height map" part of the COpenGLRenderer::Render function and in the waterheightmap.fs fragment shader. You have to be able to understand it on your own - analyze it. Don't look for wave equations in this tutorial, we don't use and we don't need any. To create the effect of spreading waves we just use couple of very simple rules, which might unveil the deeper truth about waves. Very interesting consequences of those simple rules are wave magnitude and length reduction, interference of waves, reflection of waves at the edges of the water mesh and spreading of waves. The vertical force applied to a vertex of the water mesh is stored in the red component of the water height map texture, the actual height of a vertex of the water mesh is stored in the green component of the water height map texture and the blue and alpha components of the water height map textures are not used. The vertical force applied to a vertex of the water mesh is calculated as sum of weighted averages of the height differences of a vertex and it's surrounding vertices, this force changes the height and the height is then slightly reduced to attenuate waves by time - these are those simple rules that do the trick and they're obvious. |