目录

第一章 地形编辑器

第一节:Shader 源代码

```
<1>Map.vs
#version 120
void main()
{
	gl_TexCoord[0] = gl_MultiTexCoord0;
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
Map.fs
#version 120
uniform sampler2D HeightMapTexture;
uniform float MaxHeightD2;
void main()
{
	float Height = texture2D(HeightMapTexture, gl_TexCoord[0].st).y;
	vec3 Color;
	if(Height < 0.0)
	{
		Color = vec3(0.5, 0.75f, 1.0f);
	}
	else if(Height >= 0.0 && Height < MaxHeightD2)
	{
		Color = mix(vec3(0.5, 1.0, 0.25), vec3(1.0, 0.5, 0.25), Height / MaxHeightD2);
	}
	else if(Height >= MaxHeightD2)
	{
		Color = mix(vec3(1.0, 0.5, 0.25), vec3(1.0, 1.0, 1.0), Height / MaxHeightD2 - 1.0);
	}
	gl_FragColor = vec4(Color, 1.0);
}
<2>Terrain.vs
#version 120
uniform mat4x4 ShadowMatrix;
varying vec3 Normal;
void main()
{
	gl_FrontColor = gl_Color;
	gl_TexCoord[0] = ShadowMatrix * gl_Vertex;
	Normal = gl_Normal;
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Terrain.fs

```
#version 120
uniform sampler2D ShadowMap, RotationTexture;
uniform vec3 LightDirection;
uniform float Scale, Radius;
varying vec3 Normal;
vec2 PoissonDisk[16] = vec2[](
    vec2( -0.94201624, -0.39906216 ),
    vec2( 0.94558609, -0.76890725 ),
    vec2( -0.094184101, -0.92938870 ),
    vec2( 0.34495938, 0.29387760 ),
    vec2( -0.91588581, 0.45771432 ),
    vec2( -0.81544232, -0.87912464 ),
    vec2( -0.38277543, 0.27676845 ),
    vec2( 0.97484398, 0.75648379 ),
    vec2( 0.44323325, -0.97511554 ),
    vec2( 0.53742981, -0.47373420 ),
    vec2( -0.26496911, -0.41893023 ),
    vec2( 0.79197514, 0.19090188 ),
    vec2( -0.24188840, 0.99706507 ),
    vec2( -0.81409955, 0.91437590 ),
    vec2( 0.19984126, 0.78641367 ),
    vec2( 0.14383161, -0.14100790 )
);

void main()
{
        vec3 ShadowTexCoord = gl_TexCoord[0].xyz / gl_TexCoord[0].w;
        ShadowTexCoord.z -= 0.005;
        vec4 srv = (texture2D(RotationTexture, gl_FragCoord.st * Scale) * 2.0 - 1.0) * Radius;
        mat2x2 srm = mat2x2(srv.xy, srv.zw);
        float Shadow = 0.0;
        for(int i = 0; i < 16; i++)
        {
                float Depth = texture2D(ShadowMap, ShadowTexCoord.st + srm *PoissonDisk[i]).r;
                if(ShadowTexCoord.z < Depth)
                {
                        Shadow += 1.0;
                }
        }

        Shadow /= 16.0;
        gl_FragColor = vec4(gl_Color.rgb * (0.25 + 0.75 * max(0.0, dot(normalize(Normal),
LightDirection)) * Shadow), 1.0);
}
<3>Water.vs
#version 120
void main()
{
	gl_FrontColor = gl_Color;
	gl_TexCoord[0] = gl_MultiTexCoord0;
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
Water.fs
#version 120
uniform sampler2D HeightMapTexture;
void main()
{
	float Height = texture2D(HeightMapTexture, gl_TexCoord[0].st).y;
	if(Height >= 0.0)
	{
		discard;
	}
	gl_FragColor = gl_Color;
}
```

第二节　Source Code Header

```
class CFrustum
{
protected:
	vec4 V[8];
	vec3 N[6];
	float D[6];

public:
	CFrustum();
	~CFrustum();

	void Set(mat4x4 &ViewMatrixInverse, mat4x4 &ProjectionMatrixInverse);
	void Set(int AX, int AY, int BX, int BY, int WidthM1, int HeightM1, mat4x4 
&ViewMatrixInverse, mat4x4 &ProjectionMatrixInverse);
	void Set(float AX, float AY, float BX, float BY, mat4x4 &ViewMatrixInverse, mat4x4 
&ProjectionMatrixInverse);
	bool VertexInside(vec3 &Vertex);
};

// ----------------------------------------------------------------------------------------------------------------
```

```
----------------------------------------------

class CCamera
{
public:
	vec3 X, Y, Z, Position, Reference;
	mat4x4 ViewMatrix, ViewMatrixInverse, ProjectionMatrix, ProjectionMatrixInverse;
	CFrustum Frustum;

public:
	CCamera();
	~CCamera();

	void Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference
= false);
	void Move(const vec3 &Movement);
	vec3 OnKeys(BYTE Keys, float FrameTime);
	void OnMouseMove(int dx, int dy);
	void OnMouseWheel(float zDelta);
	void SetPerspectiveProjection(float fovy, float aspect, float n, float f);

private:
	void CalculateViewMatrix();
};

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

class CSelectedVerticesList
{
protected:
	int *VerticesIndices;
	BYTE *VerticesIndicesFlags;
	int VerticesCount, VerticesIndicesCount;

public:
	CSelectedVerticesList();
	~CSelectedVerticesList();

	void AddVertexIndex(int VertexIndex);
	void Create(int VerticesCount);
	void Destroy();
	void Empty();
	int GetVertexIndex(int Index);
	int GetVerticesIndicesCount();
};

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

class CTerrain
{
protected:
	int Size, SizeP1;
	float SizeD2, MSizeD2, ODSizeD2, OMODSizeD2, MinHeight, MaxHeight;
	int VerticesCount;
	vec3 *Vertices, *Normals;
	int LinesIndicesCount, QuadsIndicesCount;
	int *LinesIndices, *QuadsIndices;
	GLuint HeightMapTexture;
	CSelectedVerticesList SelectedVerticesIndices;

public:
	CTerrain();
	~CTerrain();

	bool Create(int Size);
	bool Load(char *FileName);
	bool LoadHeightMapTexture(char *FileName, float ScaleHeight = 1.0f);
	bool Save(char *FileName);
	void RenderLines();
	void RenderQuads(bool NormalArray);
	void Destroy();

	GLuint GetHeightMapTexture();
	float GetMinHeight();
	float GetMaxHeight();
	int GetSize();
	float GetSizeD2();
	float GetMSizeD2();
	float GetODSizeD2();
	float GetOMODSizeD2();

	void CalculateMinAndMaxHeights();
	void CalculateNormals();
	void CopyVerticesToHeightMapTexture();
	void Displace(float Displacement);
	float GetHeight(int X, int Z);
	float GetHeight(float X, float Z);
	void GetMinMax(mat4x4 &ViewMatrix, vec3 &Min, vec3 &Max);
	int GetVertexIndex(int X, int Z);
	void Randomize();
	void RenderSelectionBoxAtVertex(int VertexIndex, mat4x4 &ViewMatrix);
	void RenderSelectionBoxesAtSelectedVertices(mat4x4 &ViewMatrix);
	void SelectVertices(CFrustum Frustum);
	void Smooth();
	void UnselectAllVertices();
};
// ---------------------------------------------------------------------------
---------------------------------------------

#define SHADOW_MAP_SIZE 4096
// ---------------------------------------------------------------------------
---------------------------------------------

class COpenGLRenderer
{
protected:
	int Width, Height, WidthM1, HeightM1;
	mat3x3 NormalMatrix;
	mat4x4 ModelMatrix, OrthoMatrix;

protected:
	CShaderProgram TerrainShader, WaterShader, MapShader;
	CTerrain Terrain;
	GLuint ShadowMap, RotationTexture, FBO;
	int ShadowMapSize;
	float LightAngle;
	vec3 LightPosition, LightDirection;
	mat4x4 LightViewMatrix, LightProjectionMatrix, ShadowMatrix;
	int AX, AY, BX, BY;
	bool RenderLines, RenderWater, DisplayMap, DisplayShadowMap;

public:
	CString Text;

public:
	COpenGLRenderer();
	~COpenGLRenderer();

	bool Init();
	void Render(float FrameTime);
```

```
    void Resize(int Width, int Height);
    void Destroy();

    void SetText();

    void CalculateShadowMatrix();
    void CheckCameraTerrainPosition();
    void MoveLight(float Angle);
    void RenderShadowMap();

    void OnKeyDown(UINT Key);
    void OnLButtonDown(int X, int Y);
    void OnLButtonUp(int X, int Y);
    void OnMouseMove(int X, int Y);
};
```

第三节 Source Code Cpp

```
CFrustum::CFrustum()
{
}

CFrustum::~CFrustum()
{
}

void CFrustum::Set(mat4x4 &ViewMatrixInverse, mat4x4 &ProjectionMatrixInverse)
{
    Set(-1.0f, -1.0f, 1.0f, 1.0f, ViewMatrixInverse, ProjectionMatrixInverse);
}

void CFrustum::Set(int AX, int AY, int BX, int BY, int WidthM1, int HeightM1, mat4x4
&ViewMatrixInverse, mat4x4 &ProjectionMatrixInverse)
{
    AY = HeightM1 - AY;
    BY = HeightM1 - BY;

    if(BX < AX)
    {
        int temp = AX;
        AX = BX;
        BX = temp;
    }

    if(BY < AY)
```

```
	{
		int temp = AY;
		AY = BY;
		BY = temp;
	}

	float ax = (float)AX / (float)WidthM1 * 2.0f - 1.0f;
	float ay = (float)AY / (float)HeightM1 * 2.0f - 1.0f;
	float bx = (float)BX / (float)WidthM1 * 2.0f - 1.0f;
	float by = (float)BY / (float)HeightM1 * 2.0f - 1.0f;

	Set(ax, ay, bx, by, ViewMatrixInverse, ProjectionMatrixInverse);
}

void CFrustum::Set(float AX, float AY, float BX, float BY, mat4x4 &ViewMatrixInverse, mat4x4
&ProjectionMatrixInverse)
{
	V[0] = vec4(AX, AY, -1.0f, 1.0f);
	V[1] = vec4(BX, AY, -1.0f, 1.0f);
	V[2] = vec4(BX, BY, -1.0f, 1.0f);
	V[3] = vec4(AX, BY, -1.0f, 1.0f);
	V[4] = vec4(AX, AY, 1.0f, 1.0f);
	V[5] = vec4(BX, AY, 1.0f, 1.0f);
	V[6] = vec4(BX, BY, 1.0f, 1.0f);
	V[7] = vec4(AX, BY, 1.0f, 1.0f);

	for(int i = 0; i < 8; i++)
	{
		V[i] = ViewMatrixInverse * (ProjectionMatrixInverse * V[i]);
		V[i] /= V[i].w;
	}

	N[0] = normalize(cross(*(vec3*)&V[2] - *(vec3*)&V[1], *(vec3*)&V[5] - *(vec3*)&V[1]));
	D[0] = -dot(N[0], *(vec3*)&V[1]);

	N[1] = normalize(cross(*(vec3*)&V[4] - *(vec3*)&V[0], *(vec3*)&V[3] - *(vec3*)&V[0]));
	D[1] = -dot(N[1], *(vec3*)&V[0]);

	N[2] = normalize(cross(*(vec3*)&V[1] - *(vec3*)&V[0], *(vec3*)&V[4] - *(vec3*)&V[0]));
	D[2] = -dot(N[2], *(vec3*)&V[0]);

	N[3] = normalize(cross(*(vec3*)&V[3] - *(vec3*)&V[2], *(vec3*)&V[6] - *(vec3*)&V[2]));
	D[3] = -dot(N[3], *(vec3*)&V[2]);
```

```
        N[4] = normalize(cross(*(vec3*)&V[3] - *(vec3*)&V[0], *(vec3*)&V[1] - *(vec3*)&V[0]));
        D[4] = -dot(N[4], *(vec3*)&V[0]);

        N[5] = normalize(cross(*(vec3*)&V[6] - *(vec3*)&V[5], *(vec3*)&V[4] - *(vec3*)&V[6]));
        D[5] = -dot(N[5], *(vec3*)&V[5]);
}

bool CFrustum::VertexInside(vec3 &Vertex)
{
        for(int i = 0; i < 6; i++)
        {
                if(dot(N[i], Vertex) + D[i] < 0.0f) return false;
        }

        return true;
}

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

CCamera::CCamera()
{
        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);

        CalculateViewMatrix();
}

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::CalculateViewMatrix()
{
	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);
	ViewMatrixInverse = inverse(ViewMatrix);

	Frustum.Set(ViewMatrixInverse, ProjectionMatrixInverse);
}

void CCamera::SetPerspectiveProjection(float Fovy, float Aspect, float N, float F)
{
	ProjectionMatrix = perspective(Fovy, Aspect, N, F);
	ProjectionMatrixInverse = inverse(ProjectionMatrix);

	Frustum.Set(ViewMatrixInverse, ProjectionMatrixInverse);
}

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

CCamera Camera;

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

CSelectedVerticesList::CSelectedVerticesList()
{
}

CSelectedVerticesList::~CSelectedVerticesList()
{
}

void CSelectedVerticesList::AddVertexIndex(int VertexIndex)
{
	if(VerticesIndicesFlags[VertexIndex] == 0)
	{
		VerticesIndices[VerticesIndicesCount++] = VertexIndex;
		VerticesIndicesFlags[VertexIndex] = 1;
	}
}

void CSelectedVerticesList::Create(int VerticesCount)
{
	VerticesIndices = new int[VerticesCount];
	VerticesIndicesFlags = new BYTE[VerticesCount];

	memset(VerticesIndicesFlags, 0, VerticesCount);

	this->VerticesCount = VerticesCount;

	VerticesIndicesCount = 0;
}

void CSelectedVerticesList::Destroy()
{
	delete [] VerticesIndices;
	delete [] VerticesIndicesFlags;
}

void CSelectedVerticesList::Empty()
{
	memset(VerticesIndicesFlags, 0, VerticesCount);
	VerticesIndicesCount = 0;
}

int CSelectedVerticesList::GetVertexIndex(int Index)
{
	return VerticesIndices[Index];
}

int CSelectedVerticesList::GetVerticesIndicesCount()
{
	return VerticesIndicesCount;
}
```

```
// ---------------------------------------------------------------------------
----------------------------------------------

CTerrain::CTerrain()
{
}

CTerrain::~CTerrain()
{
}

bool CTerrain::Create(int Size)
{
    this->Size = Size;

    SizeP1 = Size + 1;
    SizeD2 = (float)Size / 2.0f;
    MSizeD2 = -SizeD2;
    ODSizeD2 = 1.0f / (float)Size / 2.0f;
    OMODSizeD2 = 1.0f - ODSizeD2;

    VerticesCount = SizeP1 * SizeP1;
    LinesIndicesCount = Size * Size * 8;
    QuadsIndicesCount = Size * Size * 4;

    Vertices = new vec3[VerticesCount];
    Normals = new vec3[VerticesCount];
    LinesIndices = new int[LinesIndicesCount];
    QuadsIndices = new int[QuadsIndicesCount];

    int i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            Vertices[i].x = (float)x - SizeD2;
            Vertices[i].y = 0.0f;
            Vertices[i].z = SizeD2 - (float)z;

            Normals[i].x = 0.0f;
            Normals[i].y = 1.0f;
            Normals[i].z = 0.0f;
            i++;
        }
    }

    i = 0;

    for(int z = 0; z < Size; z++)
    {
        for(int x = 0; x < Size; x++)
        {
            LinesIndices[i++] = GetVertexIndex(x, z);
            LinesIndices[i++] = GetVertexIndex(x + 1, z);
            LinesIndices[i++] = GetVertexIndex(x + 1, z);
            LinesIndices[i++] = GetVertexIndex(x + 1, z + 1);
            LinesIndices[i++] = GetVertexIndex(x + 1, z + 1);
            LinesIndices[i++] = GetVertexIndex(x, z + 1);
            LinesIndices[i++] = GetVertexIndex(x, z + 1);
            LinesIndices[i++] = GetVertexIndex(x, z);
        }
    }

    i = 0;

    for(int z = 0; z < Size; z++)
    {
        for(int x = 0; x < Size; x++)
        {
            QuadsIndices[i++] = GetVertexIndex(x, z);
            QuadsIndices[i++] = GetVertexIndex(x + 1, z);
            QuadsIndices[i++] = GetVertexIndex(x + 1, z + 1);
            QuadsIndices[i++] = GetVertexIndex(x, z + 1);
        }
    }

    glGenTextures(1, &HeightMapTexture);

    CopyVerticesToHeightMapTexture();

    SelectedVerticesIndices.Create(VerticesCount);

    return true;
}
```

```
bool CTerrain::Load(char *FileName)
{
    CString DirectoryFileName = ModuleDirectory + FileName;

    FILE *File;

    if(fopen_s(&File, DirectoryFileName, "rb") != 0)
    {
        return false;
    }

    int Size;

    fread(&Size, 4, 1, File);

    Create(Size);

    fread(Vertices, 12, VerticesCount, File);

    fclose(File);

    CalculateMinAndMaxHeights();
    CalculateNormals();
    CopyVerticesToHeightMapTexture();

    return true;
}

bool CTerrain::LoadHeightMapTexture(char *FileName, float ScaleHeight)
{
    CTexture HeightMapTexture;

    if(!HeightMapTexture.LoadTexture2D(FileName))
    {
        return false;
    }

    if(HeightMapTexture.Width != HeightMapTexture.Height)
    {
        HeightMapTexture.Destroy();
        ErrorLog.Append("Width and height of the height map texture must be
equal!\r\n");
        return false;
    }
```

```
	Create(HeightMapTexture.Width - 1);

	vec3 *HeightMapTextureData = new vec3[VerticesCount];

	glBindTexture(GL_TEXTURE_2D, HeightMapTexture);
	glGetTexImage(GL_TEXTURE_2D, 0, GL_RGB, GL_FLOAT, HeightMapTextureData);
	glBindTexture(GL_TEXTURE_2D, 0);

	HeightMapTexture.Destroy();

	float MinY = HeightMapTextureData[0].y;

	for(int i = 1; i < VerticesCount; i++)
	{
		if(HeightMapTextureData[i].y < MinY) MinY = HeightMapTextureData[i].y;
	}

	for(int i = 0; i < VerticesCount; i++)
	{
		Vertices[i].y = (HeightMapTextureData[i].y - MinY) * 256.0f * ScaleHeight;
	}

	delete [] HeightMapTextureData;

	CalculateMinAndMaxHeights();
	CalculateNormals();
	CopyVerticesToHeightMapTexture();

	return true;
}

bool CTerrain::Save(char *FileName)
{
	CString DirectoryFileName = ModuleDirectory + FileName;

	FILE *File;

	if(fopen_s(&File, DirectoryFileName, "wb+") != 0)
	{
		return false;
	}

	fwrite(&Size, 4, 1, File);
	fwrite(Vertices, 12, VerticesCount, File);

	fclose(File);

	return true;
}

void CTerrain::RenderLines()
{
	glEnableClientState(GL_VERTEX_ARRAY);
	glVertexPointer(3, GL_FLOAT, 12, Vertices);

	glDrawElements(GL_LINES, LinesIndicesCount, GL_UNSIGNED_INT, LinesIndices);

	glDisableClientState(GL_VERTEX_ARRAY);
}

void CTerrain::RenderQuads(bool NormalArray)
{
	glEnableClientState(GL_VERTEX_ARRAY);
	glVertexPointer(3, GL_FLOAT, 12, Vertices);

	if(NormalArray)
	{
		glEnableClientState(GL_NORMAL_ARRAY);
		glNormalPointer(GL_FLOAT, 12, Normals);
	}

	glDrawElements(GL_QUADS, QuadsIndicesCount, GL_UNSIGNED_INT, QuadsIndices);

	if(NormalArray)
	{
		glDisableClientState(GL_NORMAL_ARRAY);
	}

	glDisableClientState(GL_VERTEX_ARRAY);
}

void CTerrain::Destroy()
{
	delete [] Vertices;
	delete [] Normals;
	delete [] LinesIndices;
	delete [] QuadsIndices;

	glDeleteTextures(1, &HeightMapTexture);

	SelectedVerticesIndices.Destroy();
}

GLuint CTerrain::GetHeightMapTexture()
{
	return HeightMapTexture;
}

float CTerrain::GetMinHeight()
{
	return MinHeight;
}

float CTerrain::GetMaxHeight()
{
	return MaxHeight;
}

int CTerrain::GetSize()
{
	return Size;
}

float CTerrain::GetSizeD2()
{
	return SizeD2;
}

float CTerrain::GetMSizeD2()
{
	return MSizeD2;
}

float CTerrain::GetODSizeD2()
{
	return ODSizeD2;
}

float CTerrain::GetOMODSizeD2()
{
    return OMODSizeD2;
}

void CTerrain::CalculateMinAndMaxHeights()
{
    MinHeight = MaxHeight = Vertices[0].y;

    for(int i = 1; i < VerticesCount; i++)
    {
        if(Vertices[i].y < MinHeight) MinHeight = Vertices[i].y;
        if(Vertices[i].y > MaxHeight) MaxHeight = Vertices[i].y;
    }
}

void CTerrain::CalculateNormals()
{
    int i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            Normals[i++] = normalize(vec3(GetHeight(x - 1, z) - GetHeight(x + 1, z), 2.0f, GetHeight(x, z + 1) - GetHeight(x, z - 1)));
        }
    }
}

void CTerrain::CopyVerticesToHeightMapTexture()
{
    glBindTexture(GL_TEXTURE_2D, HeightMapTexture);
    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_RGBA16F, SizeP1, SizeP1, 0, GL_RGB, GL_FLOAT, Vertices);
    glBindTexture(GL_TEXTURE_2D, 0);
}

void CTerrain::Displace(float Displacement)
{
    if(SelectedVerticesIndices.GetVerticesIndicesCount() > 0)
    {
```

```
		for(int i = 0; i < SelectedVerticesIndices.GetVerticesIndicesCount(); i++)
		{
			Vertices[SelectedVerticesIndices.GetVertexIndex(i)].y += Displacement;
		}

		CalculateNormals();
	}
	else
	{
		for(int i = 0; i < VerticesCount; i++)
		{
			Vertices[i].y += Displacement;
		}
	}

	CalculateMinAndMaxHeights();
	CopyVerticesToHeightMapTexture();
}

float CTerrain::GetHeight(int X, int Z)
{
	return Vertices[GetVertexIndex(X < 0 ? 0 : X > Size ? Size : X, Z < 0 ? 0 : Z > Size ? Size :
Z)].y;
}

float CTerrain::GetHeight(float X, float Z)
{
	Z = -Z;

	X += SizeD2;
	Z += SizeD2;

	float Size = (float)this->Size;

	if(X < 0.0f) X = 0.0f;
	if(X > Size) X = Size;
	if(Z < 0.0f) Z = 0.0f;
	if(Z > Size) Z = Size;

	int ix = (int)X;
	int iz = (int)Z;

	float fx = X - (float)ix;
	float fz = Z - (float)iz;
	float a = GetHeight(ix, iz);
	float b = GetHeight(ix + 1, iz);
	float c = GetHeight(ix, iz + 1);
	float d = GetHeight(ix + 1, iz + 1);

	float ab = a + (b - a) * fx;
	float cd = c + (d - c) * fx;

	return ab + (cd - ab) * fz;
}

void CTerrain::GetMinMax(mat4x4 &ViewMatrix, vec3 &Min, vec3 &Max)
{
	vec4 Vertex = ViewMatrix * vec4(Vertices[0], 1.0);

	Min.x = Max.x = Vertex.x;
	Min.y = Max.y = Vertex.y;
	Min.z = Max.z = Vertex.z;

	for(int i = 1; i < VerticesCount; i++)
	{
		Vertex = ViewMatrix * vec4(Vertices[i], 1.0);

		if(Vertex.x < Min.x) Min.x = Vertex.x;
		if(Vertex.y < Min.y) Min.y = Vertex.y;
		if(Vertex.z < Min.z) Min.z = Vertex.z;

		if(Vertex.x > Max.x) Max.x = Vertex.x;
		if(Vertex.y > Max.y) Max.y = Vertex.y;
		if(Vertex.z > Max.z) Max.z = Vertex.z;
	}
}

int CTerrain::GetVertexIndex(int X, int Z)
{
	return SizeP1 * Z + X;
}

void CTerrain::Randomize()
{
	if(SelectedVerticesIndices.GetVerticesIndicesCount() > 0)
	{
		for(int i = 0; i < SelectedVerticesIndices.GetVerticesIndicesCount(); i++)
```

```
        {
            Vertices[SelectedVerticesIndices.GetVertexIndex(i)].y += (float)rand() /
(float)RAND_MAX - 0.5f;
        }
    }
    else
    {
        for(int i = 0; i < VerticesCount; i++)
        {
            Vertices[i].y += (float)rand() / (float)RAND_MAX - 0.5f;
        }
    }

    CalculateMinAndMaxHeights();
    CalculateNormals();
    CopyVerticesToHeightMapTexture();
}

float SelectionBoxVertices[] = {
    -0.125f, -0.125f,  0.125f,
     0.125f, -0.125f,  0.125f,
     0.125f,  0.125f,  0.125f,
    -0.125f,  0.125f,  0.125f,
    -0.125f, -0.125f, -0.125f,
     0.125f, -0.125f, -0.125f,
     0.125f,  0.125f, -0.125f,
    -0.125f,  0.125f, -0.125f
};

int SelectionBoxLinesIndices[] = {
    0, 1, 1, 2, 2, 3, 3, 0,
    4, 5, 5, 6, 6, 7, 7, 4,
    0, 4, 1, 5, 2, 6, 3, 7
};

void CTerrain::RenderSelectionBoxAtVertex(int VertexIndex, mat4x4 &ViewMatrix)
{
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 12, SelectionBoxVertices);

    glLoadMatrixf(&ViewMatrix);
    glTranslatef(Vertices[VertexIndex].x, Vertices[VertexIndex].y, Vertices[VertexIndex].z);

    glDrawElements(GL_LINES, 24, GL_UNSIGNED_INT, SelectionBoxLinesIndices);
    glDisableClientState(GL_VERTEX_ARRAY);
}

void CTerrain::RenderSelectionBoxesAtSelectedVertices(mat4x4 &ViewMatrix)
{
    if(SelectedVerticesIndices.GetVerticesIndicesCount() > 0)
    {
        glMatrixMode(GL_MODELVIEW);

        for(int i = 0; i < SelectedVerticesIndices.GetVerticesIndicesCount(); i++)
        {
            RenderSelectionBoxAtVertex(SelectedVerticesIndices.GetVertexIndex(i),
ViewMatrix);
        }

        glLoadMatrixf(&ViewMatrix);
    }
}

void CTerrain::SelectVertices(CFrustum Frustum)
{
    for(int i = 0; i < VerticesCount; i++)
    {
        if(Frustum.VertexInside(Vertices[i]))
        {
            SelectedVerticesIndices.AddVertexIndex(i);
        }
    }
}

void CTerrain::Smooth()
{
    if(SelectedVerticesIndices.GetVerticesIndicesCount() > 0)
    {
        float *NewHeights = new float[SelectedVerticesIndices.GetVerticesIndicesCount()];

        for(int i = 0; i < SelectedVerticesIndices.GetVerticesIndicesCount(); i++)
        {
            int VertexIndex = SelectedVerticesIndices.GetVertexIndex(i);

            int x = (int)(Vertices[VertexIndex].x + SizeD2);
            int z = (int)(SizeD2 - Vertices[VertexIndex].z);
            NewHeights[i] = 0.0f;

            NewHeights[i] += GetHeight(x - 1, z + 1) + GetHeight(x, z + 1) * 2 + GetHeight(x + 1, z + 1);
            NewHeights[i] += GetHeight(x - 1, z) * 2 + GetHeight(x, z) * 3 + GetHeight(x + 1, z) * 2;
            NewHeights[i] += GetHeight(x - 1, z - 1) + GetHeight(x, z - 1) * 2 + GetHeight(x + 1, z - 1);

            NewHeights[i] /= 15.0f;
        }

        for(int i = 0; i < SelectedVerticesIndices.GetVerticesIndicesCount(); i++)
        {
            Vertices[SelectedVerticesIndices.GetVertexIndex(i)].y = NewHeights[i];
        }

        delete [] NewHeights;
    }
    else
    {
        float *NewHeights = new float[VerticesCount];

        int i = 0;

        for(int z = 0; z <= Size; z++)
        {
            for(int x = 0; x <= Size; x++)
            {
                NewHeights[i] = 0.0f;

                NewHeights[i] += GetHeight(x - 1, z + 1) + GetHeight(x, z + 1) * 2 + GetHeight(x + 1, z + 1);
                NewHeights[i] += GetHeight(x - 1, z) * 2 + GetHeight(x, z) * 3 + GetHeight(x + 1, z) * 2;
                NewHeights[i] += GetHeight(x - 1, z - 1) + GetHeight(x, z - 1) * 2 + GetHeight(x + 1, z - 1);

                NewHeights[i] /= 15.0f;

                i++;
            }
        }
```

```
		for(int i = 0; i < VerticesCount; i++)
		{
			Vertices[i].y = NewHeights[i];
		}

		delete [] NewHeights;
	}

	CalculateMinAndMaxHeights();
	CalculateNormals();
	CopyVerticesToHeightMapTexture();
}

void CTerrain::UnselectAllVertices()
{
	SelectedVerticesIndices.Empty();
}

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

COpenGLRenderer::COpenGLRenderer()
{
	LightAngle = 0.0f;

	AX = AY = BX = BY = -1;

	RenderLines = true;
	RenderWater = true;
	DisplayMap = true;
	DisplayShadowMap = false;
}

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;
    }

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

    Error |= !TerrainShader.Load("terrain.vs", "terrain.fs");
    Error |= !WaterShader.Load("water.vs", "water.fs");
    Error |= !MapShader.Load("map.vs", "map.fs");

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

    int terrain = 2;

    switch(terrain)
    {
        case 0: Error = !Terrain.Load("terrain0.xyz"); break;
        case 1: Error = !Terrain.LoadHeightMapTexture("terrain1.jpg", 0.0625f); break;
        case 2: Error = !Terrain.LoadHeightMapTexture("terrain2.jpg", 0.5f); break;
        case 3: Error = !Terrain.LoadHeightMapTexture("terrain3.jpg", 0.5f); break;
    }

    // ------------------------------------------------------------------------
----------------------------------------------
```

```
    if(Error)
    {
        return false;
    }

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

    if(terrain == 1)
    {
        Terrain.Displace(-2.5f);
    }

    if(terrain == 2 || terrain == 3)
    {
        Terrain.Smooth();
        Terrain.Smooth();
    }

    if(terrain == 3)
    {
        Terrain.Displace(-7.0f);
    }

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

    TerrainShader.UniformLocations = new GLuint[2];
    TerrainShader.UniformLocations[0] = glGetUniformLocation(TerrainShader,
"ShadowMatrix");
    TerrainShader.UniformLocations[1] = glGetUniformLocation(TerrainShader,
"LightDirection");

    MapShader.UniformLocations = new GLuint[1];
    MapShader.UniformLocations[0] = glGetUniformLocation(MapShader, "MaxHeightD2");

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

    glUseProgram(TerrainShader);
    glUniform1i(glGetUniformLocation(TerrainShader, "ShadowMap"), 0);
    glUniform1i(glGetUniformLocation(TerrainShader, "RotationTexture"), 1);
    glUniform1f(glGetUniformLocation(TerrainShader, "Scale"), 1.0f / 64.0f);
    glUniform1f(glGetUniformLocation(TerrainShader, "Radius"), 1.0f / 1024.0f);
```

```
	glUseProgram(0);

	glUseProgram(MapShader);
	glUniform1f(MapShader.UniformLocations[0], Terrain.GetMaxHeight() / 2.0f);
	glUseProgram(0);

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

	ShadowMapSize = SHADOW_MAP_SIZE > gl_max_texture_size ? gl_max_texture_size : SHADOW_MAP_SIZE;

	glGenTextures(1, &ShadowMap);
	glBindTexture(GL_TEXTURE_2D, ShadowMap);
	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, ShadowMapSize, ShadowMapSize, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
	glBindTexture(GL_TEXTURE_2D, 0);

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

	srand(GetTickCount());

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

	vec4 *RotationTextureData = new vec4[4096];

	float Angle = 3.14f * 2.0f * (float)rand() / (float)RAND_MAX;

	for(int i = 0; i < 4096; i++)
	{
		RotationTextureData[i].x = cos(Angle);
		RotationTextureData[i].y = sin(Angle);
		RotationTextureData[i].z = -RotationTextureData[i].y;
		RotationTextureData[i].w = RotationTextureData[i].x;

		RotationTextureData[i] *= 0.5f;
		RotationTextureData[i] += 0.5f;
```

```
        Angle += 3.14f * 2.0f * (float)rand() / (float)RAND_MAX;
    }

    glGenTextures(1, &RotationTexture);
    glBindTexture(GL_TEXTURE_2D, RotationTexture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 64, 64, 0, GL_RGBA, GL_FLOAT,
RotationTextureData);
    glBindTexture(GL_TEXTURE_2D, 0);

    delete [] RotationTextureData;

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

    glGenFramebuffersEXT(1, &FBO);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
    glDrawBuffers(0, NULL); glReadBuffer(GL_NONE);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
GL_TEXTURE_2D, ShadowMap, 0);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

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

    MoveLight(90.0f);

    CalculateShadowMatrix();

    RenderShadowMap();

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

    Camera.Look(vec3(0.0f, Terrain.GetSizeD2(), Terrain.GetSizeD2()), vec3(0.0f, 1.75f, 0.0f),
true);

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

    CheckCameraTerrainPosition();

    // ------------------------------------------------------------------------
```

```
-------------------------------------------

    SetText();

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

    return true;

    // -------------------------------------------------------------------
-------------------------------------------
}

void COpenGLRenderer::Render(float FrameTime)
{
    // -------------------------------------------------------------------
-------------------------------------------

    glViewport(0, 0, Width, Height);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glEnable(GL_DEPTH_TEST);

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(&Camera.ProjectionMatrix);

    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(&Camera.ViewMatrix);

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

    glColor3f(1.0f, 0.5f, 0.0f);

    glBegin(GL_POINTS);
        glVertex3fv(&LightPosition);
    glEnd();

    vec3 Reference = Camera.Reference - vec3(0.0f, 1.75f, 0.0f);

    glBegin(GL_LINES);
        glVertex3f(Reference.x + 0.125f, Reference.y, Reference.z);
        glVertex3f(Reference.x - 0.125f, Reference.y, Reference.z);
```

```
		glVertex3f(Reference.x, Reference.y - 0.125f, Reference.z);
		glVertex3f(Reference.x, Reference.y + 0.125f, Reference.z);
		glVertex3f(Reference.x, Reference.y, Reference.z - 0.125f);
		glVertex3f(Reference.x, Reference.y, Reference.z + 0.125f);
	glEnd();

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

	if(RenderLines)
	{
		glColor3f(0.125f, 0.125f, 0.125f);

		Terrain.RenderLines();
	}

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

	glEnable(GL_CULL_FACE);

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

	glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, ShadowMap);
	glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, RotationTexture);

	glUseProgram(TerrainShader);

	Terrain.RenderQuads(true);

	glUseProgram(0);

	glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, 0);
	glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, 0);

	glDisable(GL_CULL_FACE);

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

	glColor3f(0.0f, 1.0f, 0.0f);

	Terrain.RenderSelectionBoxesAtSelectedVertices(Camera.ViewMatrix);
	// ------------------------------------------------------------------------
-------------------------------------------

	if(RenderWater)
	{
		glEnable(GL_BLEND);

		glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

		glColor4f(0.5f, 0.75f, 1.0f, 0.25f);

		glBindTexture(GL_TEXTURE_2D, Terrain.GetHeightMapTexture());

		glUseProgram(WaterShader);

		glBegin(GL_QUADS);
			glTexCoord2f(Terrain.GetODSizeD2(), Terrain.GetODSizeD2());
glVertex3f(Terrain.GetMSizeD2(), 0.0f, Terrain.GetSizeD2());
			glTexCoord2f(Terrain.GetOMODSizeD2(), Terrain.GetODSizeD2());
glVertex3f(Terrain.GetSizeD2(), 0.0f, Terrain.GetSizeD2());
			glTexCoord2f(Terrain.GetOMODSizeD2(), Terrain.GetOMODSizeD2());
glVertex3f(Terrain.GetSizeD2(), 0.0f, Terrain.GetMSizeD2());
			glTexCoord2f(Terrain.GetODSizeD2(), Terrain.GetOMODSizeD2());
glVertex3f(Terrain.GetMSizeD2(), 0.0f, Terrain.GetMSizeD2());
		glEnd();

		glUseProgram(0);

		glBindTexture(GL_TEXTURE_2D, 0);

		glDisable(GL_BLEND);
	}

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

	glDisable(GL_DEPTH_TEST);

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

	if(AX >= 0 && AX <= WidthM1 && AY >= 0 && AY <= HeightM1 && BX >= 0 && BX
<= WidthM1 && BY >= 0 && BY <= HeightM1)
	{
        glMatrixMode(GL_PROJECTION);
        glLoadMatrixf(&OrthoMatrix);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        float ax = (float)AX;
        float ay = (float)AY;
        float bx = (float)BX;
        float by = (float)BY;

        glColor3f(0.0f, 1.0f, 0.0f);

        glBegin(GL_LINES);
            glVertex2f(ax, ay); glVertex2f(bx, ay);
            glVertex2f(bx, ay); glVertex2f(bx, by);
            glVertex2f(bx, by); glVertex2f(ax, by);
            glVertex2f(ax, by); glVertex2f(ax, ay);
        glEnd();
    }

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

    if(DisplayMap)
    {
        glViewport(16, 16, 256, 256);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

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

        glEnable(GL_TEXTURE_2D);

        glBindTexture(GL_TEXTURE_2D, Terrain.GetHeightMapTexture());

        glUseProgram(MapShader);

        glBegin(GL_QUADS);
            glTexCoord2f(Terrain.GetODSizeD2(), Terrain.GetODSizeD2()); glVertex2f(-
1.0f, -1.0f);
            glTexCoord2f(Terrain.GetOMODSizeD2(), Terrain.GetODSizeD2());
glVertex2f(1.0f, -1.0f);
            glTexCoord2f(Terrain.GetOMODSizeD2(), Terrain.GetOMODSizeD2());
glVertex2f(1.0f, 1.0f);
            glTexCoord2f(Terrain.GetODSizeD2(), Terrain.GetOMODSizeD2()); glVertex2f(-
1.0f, 1.0f);
        glEnd();

        glUseProgram(0);

        glBindTexture(GL_TEXTURE_2D, 0);

        glDisable(GL_TEXTURE_2D);

        vec2 Position = (vec2(Camera.Reference.x, -Camera.Reference.z) +
Terrain.GetSizeD2()) / (float)Terrain.GetSize() * 2.0f - 1.0f;

        glColor3f(0.0f, 0.0f, 0.0f);

        glBegin(GL_LINES);
            glVertex2f(Position.x - 0.03125f, Position.y); glVertex2f(Position.x + 0.03125f,
Position.y);
            glVertex2f(Position.x, Position.y - 0.03125f); glVertex2f(Position.x, Position.y +
0.03125f);
        glEnd();
    }

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

    if(DisplayShadowMap)
    {
        glViewport(Width - 1 - 16 - 256, 16, 256, 256);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

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

        glEnable(GL_TEXTURE_2D);
```

```
		glBindTexture(GL_TEXTURE_2D, ShadowMap);

		glBegin(GL_QUADS);
			glTexCoord2f(0.0f, 0.0f); glVertex2f(-1.0f, -1.0f);
			glTexCoord2f(1.0f, 0.0f); glVertex2f(1.0f, -1.0f);
			glTexCoord2f(1.0f, 1.0f); glVertex2f(1.0f, 1.0f);
			glTexCoord2f(0.0f, 1.0f); glVertex2f(-1.0f, 1.0f);
		glEnd();

		glBindTexture(GL_TEXTURE_2D, 0);

		glDisable(GL_TEXTURE_2D);
	}

	// -----------------------------------------------------------------------
----------------------------------------------
}

void COpenGLRenderer::Resize(int Width, int Height)
{
	this->Width = Width;
	this->Height = Height;

	WidthM1 = Width - 1;
	HeightM1 = Height - 1;

	Camera.SetPerspectiveProjection(45.0f, (float)Width / (float)Height, 0.125f, 1024.0f);

	OrthoMatrix = ortho(0.0f, (float)WidthM1, (float)HeightM1, 0.0f, 0.0f, 1.0f);
}

void COpenGLRenderer::Destroy()
{
	TerrainShader.Destroy();
	WaterShader.Destroy();
	MapShader.Destroy();

	Terrain.Destroy();

	glDeleteTextures(1, &ShadowMap);
	glDeleteTextures(1, &RotationTexture);

	if(GLEW_EXT_framebuffer_object)
```

```
    {
        glDeleteFramebuffersEXT(1, &FBO);
    }
}

void COpenGLRenderer::SetText()
{
    Text.Set("MinHeight = %f, MaxHeight = %f", Terrain.GetMinHeight(),
Terrain.GetMaxHeight());
}

void COpenGLRenderer::CalculateShadowMatrix()
{
    LightViewMatrix = look(LightPosition, vec3(0.0), vec3(0.0f, 1.0f, 0.0f));

    vec3 Min, Max;

    Terrain.GetMinMax(LightViewMatrix, Min, Max);

    LightProjectionMatrix = ortho(Min.x, Max.x, Min.y, Max.y, -Max.z, -Min.z);

    ShadowMatrix = BiasMatrix * LightProjectionMatrix * LightViewMatrix;

    glUseProgram(TerrainShader);
    glUniformMatrix4fv(TerrainShader.UniformLocations[0], 1, GL_FALSE, &ShadowMatrix);
    glUseProgram(0);
}

void COpenGLRenderer::CheckCameraTerrainPosition()
{
    float TerrainSizeD2 = Terrain.GetSizeD2();

    if(Camera.Reference.x < -TerrainSizeD2) Camera.Move(vec3(-TerrainSizeD2 -
Camera.Reference.x, 0.0f, 0.0f));
    if(Camera.Reference.x > TerrainSizeD2) Camera.Move(vec3(TerrainSizeD2 -
Camera.Reference.x, 0.0f, 0.0f));
    if(Camera.Reference.z < -TerrainSizeD2) Camera.Move(vec3(0.0f, 0.0f, -TerrainSizeD2 -
Camera.Reference.z));
    if(Camera.Reference.z > TerrainSizeD2) Camera.Move(vec3(0.0f, 0.0f, TerrainSizeD2 -
Camera.Reference.z));

    Camera.Move(vec3(0.0f, Terrain.GetHeight(Camera.Reference.x, Camera.Reference.z) +
1.75f - Camera.Reference.y, 0.0f));
}
```

```
void COpenGLRenderer::MoveLight(float Angle)
{
	LightAngle += Angle;

	LightPosition = rotate(vec3((float)Terrain.GetSize(), 0.0f, 0.0f), -LightAngle, vec3(0.0f,
1.0f, -1.0f));

	LightDirection = normalize(LightPosition);

	glUseProgram(TerrainShader);
	glUniform3fv(TerrainShader.UniformLocations[1], 1, &LightDirection);
	glUseProgram(0);
}

void COpenGLRenderer::RenderShadowMap()
{
	glViewport(0, 0, ShadowMapSize, ShadowMapSize);

	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);

	glClear(GL_DEPTH_BUFFER_BIT);

	glMatrixMode(GL_PROJECTION);
	glLoadMatrixf(&LightProjectionMatrix);

	glMatrixMode(GL_MODELVIEW);
	glLoadMatrixf(&LightViewMatrix);

	glEnable(GL_DEPTH_TEST);

	Terrain.RenderQuads(false);

	glDisable(GL_DEPTH_TEST);

	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}
void COpenGLRenderer::OnKeyDown(UINT Key)
{
	bool TerrainChanged = false, LightChanged = false;

	switch(Key)
	{
		case VK_F1:
    Terrain.Smooth();
    TerrainChanged = true;
    break;

case VK_F2:
    Terrain.Randomize();
    TerrainChanged = true;
    break;

case VK_F3:
    DisplayMap = !DisplayMap;
    break;

case VK_F4:
    DisplayShadowMap = !DisplayShadowMap;
    break;

case VK_F5:
    Terrain.Save("terrain-saved.xyz");
    break;

case '1':
    RenderLines = !RenderLines;
    break;

case '2':
    RenderWater = !RenderWater;
    break;

case 'R':
    Terrain.Displace(1.0f);
    TerrainChanged = true;
    break;

case 'F':
    Terrain.Displace(-1.0f);
    TerrainChanged = true;
    break;

case 'N':
    if(GetKeyState(VK_CONTROL) & 0x80)
    {
        Terrain.Destroy();
        Terrain.Create(128);
```

```
					TerrainChanged = true;
				}
				break;

			case VK_ADD:
				MoveLight(7.5f);
				LightChanged = true;
				break;

			case VK_SUBTRACT:
				MoveLight(-7.5f);
				LightChanged = true;
				break;
		}

		if(TerrainChanged)
		{
			glUseProgram(MapShader);
			glUniform1f(MapShader.UniformLocations[0], Terrain.GetMaxHeight() / 2.0f);
			glUseProgram(0);

			CalculateShadowMatrix();
			RenderShadowMap();
			CheckCameraTerrainPosition();
			SetText();
		}

		if(LightChanged)
		{
			CalculateShadowMatrix();
			RenderShadowMap();
		}
	}

	void COpenGLRenderer::OnLButtonDown(int X, int Y)
	{
		AX = BX = X;
		AY = BY = Y;
	}

	void COpenGLRenderer::OnLButtonUp(int X, int Y)
	{
		if(!(GetKeyState(VK_CONTROL) & 0x80))
		{
```

```
		Terrain.UnselectAllVertices();
	}

	if(AX != BX && AY != BY)
	{
		CFrustum Frustum;

		Frustum.Set(AX, AY, BX, BY, WidthM1, HeightM1, Camera.ViewMatrixInverse, 
Camera.ProjectionMatrixInverse);

		Terrain.SelectVertices(Frustum);
	}
	AX = AY = BX = BY = -1;
}

void COpenGLRenderer::OnMouseMove(int X, int Y)
{
	if(GetKeyState(VK_LBUTTON) & 0x80)
	{
		BX = X;
		BY = Y;
	}
}
```

第二章　屏幕空间环境光遮蔽

第一节:Shader Source

<1>:defferedlighting.vs

```
#version 120
void main()
{
    gl_TexCoord[0] = gl_Vertex;
    gl_Position = gl_Vertex * 2.0 - 1.0;
}
```

Deferredlighting.fs

```
#version 120
uniform sampler2D ColorBuffer, NormalBuffer, DepthBuffer, SSAOBuffer;
uniform mat4x4 ProjectionBiasMatrixInverse;
uniform bool ShowPositionBuffer, ShowNormalBuffer, ShowDepthBuffer;
void main()
{
    if(ShowPositionBuffer)
    {
        float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;

        if(Depth < 1.0)
        {
            vec4 Position = ProjectionBiasMatrixInverse *
vec4(gl_TexCoord[0].st, Depth, 1.0);
            Position.xyz /= Position.w;

            gl_FragColor = vec4(Position.xyz, 1.0);
        }
        else
        {
            gl_FragColor = vec4(vec3(0.0), 1.0);
        }
    }
    else if(ShowNormalBuffer)
    {
        gl_FragColor = texture2D(NormalBuffer, gl_TexCoord[0].st);
    }
    else if(ShowDepthBuffer)
    {
        float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;
        gl_FragColor = vec4(vec3(Depth), 1.0);
    }
    else
    {
        gl_FragColor = texture2D(ColorBuffer, gl_TexCoord[0].st);

        float SSAO = texture2D(SSAOBuffer, gl_TexCoord[0].st).r;

        gl_FragColor.rgb *= SSAO;
    }
}
```

<2>:preprocess.vs

```
#version 120
varying vec3 Normal;

void main()
{
    gl_FrontColor = gl_Color;
    Normal = gl_NormalMatrix * gl_Normal;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Preprocess,fs

```
#version 120
varying vec3 Normal;
void main()
{
    gl_FragData[0] = gl_Color;
    gl_FragData[1] = vec4(normalize(Normal) * 0.5 + 0.5, 1.0);
}
```

<3>:SSAO.vs

```
#version 120
uniform vec2 Scale;

void main()
{
    gl_TexCoord[0] = gl_Vertex;
    gl_TexCoord[1] = vec4(gl_Vertex.xy * Scale, gl_Vertex.zw);
    gl_Position = gl_Vertex * 2.0 - 1.0;
}
```

SSAO.fs

```
#version 120
uniform sampler2D NormalBuffer, DepthBuffer, RotationTexture;
uniform mat4x4 ProjectionBiasMatrixInverse;
uniform vec2 Samples[16];
uniform float Radius, Strength, ConstantAttenuation, LinearAttenuation,
QuadraticAttenuation;
```

```
void main()
{
    float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;
    if(Depth < 1.0)
    {
        vec3 Normal = normalize(texture2D(NormalBuffer, gl_TexCoord[0].st).rgb
* 2.0 - 1.0);
        vec4 Position = ProjectionBiasMatrixInverse * vec4(gl_TexCoord[0].st,
Depth, 1.0);
        Position.xyz /= Position.w;

        if(dot(Normal, Position.xyz) > 0.0)
        {
            Normal = -Normal;
        }
        vec4 ScaleRotationVector = normalize(texture2D(RotationTexture,
gl_TexCoord[1].st) * 2.0 - 1.0) * Radius;

        mat2x2 ScaleRotationMatrix = mat2x2(ScaleRotationVector.xy,
ScaleRotationVector.zw);
        float SSAO = 0.0;
        for(int i = 0; i < 16; i++)
        {
            vec2 TexCoord = clamp(gl_TexCoord[0].st + ScaleRotationMatrix *
Samples[i], 0.0, 0.999999);

            float SampleDepth = texture2D(DepthBuffer, TexCoord).r;

            vec4 SamplePosition = ProjectionBiasMatrixInverse * vec4(TexCoord,
SampleDepth, 1.0);
            SamplePosition.xyz /= SamplePosition.w;

            vec3 P2SP = SamplePosition.xyz - Position.xyz;
            float Distance2 = dot(P2SP, P2SP);
            float Distance = sqrt(Distance2);
            float NdotP2SP = dot(Normal, P2SP) / Distance;
            if(NdotP2SP > 0.342)
            {
                SSAO += NdotP2SP / (ConstantAttenuation + Distance *
LinearAttenuation + Distance2 * QuadraticAttenuation);
            }
        }
        gl_FragColor = vec4(vec3(1.0 - SSAO * 0.0625 * Strength), 1.0);
    }
    else
    {
        gl_FragColor = vec4(vec3(0.0), 1.0);
    }
}
```

<4>:SSAOFilter.vs

```
#version 120
void main()
{
    gl_TexCoord[0] = gl_Vertex;
    gl_Position = gl_Vertex * 2.0 - 1.0;
}
```

SSAOFilterV.fs

```
#version 120
uniform sampler2D SSAOBuffer, DepthBuffer;
uniform float PixelSizeY, fs, fd;
float Offsets[8] = float[](-4.0, -3.0, -2.0, -1.0, 1.0, 2.0, 3.0, 4.0);
float BlurWeights[8] = float[](1.0, 2.0, 3.0, 4.0, 4.0, 3.0, 2.0, 1.0);
void main()
{
    float BlurWeightsSum = 5.0;

    float SSAO = texture2D(SSAOBuffer, gl_TexCoord[0].st).r * BlurWeightsSum;
    float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;

    float Factor = fs - fd * Depth;
    for(int i = 0; i < 8; i++)
    {
        vec2 TexCoord = vec2(gl_TexCoord[0].s, gl_TexCoord[0].t + Offsets[i] * PixelSizeY);

        float DepthDifference = abs(Depth - texture2D(DepthBuffer, TexCoord).r);
        if(DepthDifference < Factor)
        {
            SSAO += texture2D(SSAOBuffer, TexCoord).r * BlurWeights[i];
            BlurWeightsSum += BlurWeights[i];
        }
    }
    gl_FragColor = vec4(vec3(SSAO / BlurWeightsSum), 1.0);
}
```

SSAOFilterH.fs

```
#version 120
```

```
uniform sampler2D SSAOBuffer, DepthBuffer;
uniform float PixelSizeX, fs, fd;

float Offsets[8] = float[](-4.0, -3.0, -2.0, -1.0, 1.0, 2.0, 3.0, 4.0);
float BlurWeights[8] = float[](1.0, 2.0, 3.0, 4.0, 4.0, 3.0, 2.0, 1.0);

void main()
{
    float BlurWeightsSum = 5.0;

    float SSAO = texture2D(SSAOBuffer, gl_TexCoord[0].st).r * BlurWeightsSum;
    float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;

    float Factor = fs - fd * Depth;

    for(int i = 0; i < 8; i++)
    {
        vec2 TexCoord = vec2(gl_TexCoord[0].s + Offsets[i] * PixelSizeX,
gl_TexCoord[0].t);

        float DepthDifference = abs(Depth - texture2D(DepthBuffer,
TexCoord).r);

        if(DepthDifference < Factor)
        {
            SSAO += texture2D(SSAOBuffer, TexCoord).r * BlurWeights[i];
            BlurWeightsSum += BlurWeights[i];
        }
    }

    gl_FragColor = vec4(vec3(SSAO / BlurWeightsSum), 1.0);
}
```

第二节: Source Code Header

```
class CCamera
{
public:
    vec3 X, Y, Z, Position, Reference;

public:
    mat4x4 ViewMatrix, ViewMatrixInverse, ProjectionMatrix,
ProjectionMatrixInverse, ViewProjectionMatrix, ViewProjectionMatrixInverse;

public:
    CCamera();
    ~CCamera();

public:
    void Look(const vec3 &Position, const vec3 &Reference, bool
RotateAroundReference = false);
    void Move(const vec3 &Movement);
    vec3 OnKeys(BYTE Keys, float FrameTime);
    void OnMouseMove(int dx, int dy);
    void OnMouseWheel(float zDelta);
    void SetPerspective(float fovy, float aspect, float n, float f);

private:
    void CalculateViewMatrix();
};

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

class CScene
{
private:
    vec3 *Vertices;
    int VerticesCount;

private:
    GLuint VertexBufferObject;

public:
    CScene();
    ~CScene();

private:
    void SetDefaults();

public:
    bool LoadBinary(const char *FileName);
    void Render();
    void Destroy();
};

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

class COpenGLRenderer
```

```
{
private:
    int LastX, LastY, LastClickedX, LastClickedY;

private:
    int Width, Height;

private:
    CCamera Camera;

private:
    CShaderProgram Preprocess, SSAO, SSAOFilterH, SSAOFilterV,
DeferredLighting, FXAA;
    GLuint RotationTexture, ColorBuffer, NormalBuffer, DepthBuffer, SSAOBuffer,
SSAOFilterBuffer, FXAABuffer, FBO;

private:
    CScene Scene;

private:
    bool RenderGLUTObjects, ShowPositionBuffer, ShowNormalBuffer,
ShowDepthBuffer, ApplySSAOFilter, ApplyFXAA;

public:
    CString Text;

public:
    COpenGLRenderer();
    ~COpenGLRenderer();

public:
    bool Init();
    void Render();
    void Animate(float FrameTime);
    void Resize(int Width, int Height);
    void Destroy();

public:
    void CheckCameraKeys(float FrameTime);

public:
    void OnKeyDown(UINT Key);
    void OnLButtonDown(int X, int Y);
    void OnLButtonUp(int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseWheel(short zDelta);
    void OnRButtonDown(int X, int Y);
    void OnRButtonUp(int X, int Y);
};
```

第三节: Source Code Cpp

```
CScene::CScene()
{
    SetDefaults();
}

CScene::~CScene()
{
}

void CScene::SetDefaults()
{
    Vertices = NULL;
    VerticesCount = 0;

    VertexBufferObject = 0;
}

bool CScene::LoadBinary(const char *FileName)
{
    CString DirectoryFileName = ModuleDirectory + FileName;

    FILE *File;

    if(fopen_s(&File, DirectoryFileName, "rb") != 0)
    {
        ErrorLog.Append("Error opening file " + DirectoryFileName + "!\r\n");
        return false;
    }

    Destroy();

    if(fread(&VerticesCount, sizeof(int), 1, File) != 1)
    {
        ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
        fclose(File);
        return false;
    }
    if(VerticesCount > 0)
    {
        Vertices = new vec3[VerticesCount];

        if(fread(Vertices, sizeof(vec3), VerticesCount, File) != VerticesCount)
        {
            ErrorLog.Append("Error reading file " + DirectoryFileName +
"!\r\n");
            fclose(File);
            Destroy();
            return false;
        }

        vec3 *VertexBufferData = new vec3[VerticesCount * 2];

        for(int i = 0; i < VerticesCount; i += 3)
        {
            vec3 VertexA = Vertices[i + 0];
            vec3 VertexB = Vertices[i + 1];
            vec3 VertexC = Vertices[i + 2];

            vec3 Normal = normalize(cross(VertexB - VertexA, VertexC -
VertexA));

            VertexBufferData[i * 2 + 0] = VertexA;
            VertexBufferData[i * 2 + 1] = Normal;
            VertexBufferData[i * 2 + 2] = VertexB;
            VertexBufferData[i * 2 + 3] = Normal;
            VertexBufferData[i * 2 + 4] = VertexC;
            VertexBufferData[i * 2 + 5] = Normal;
        }

        glGenBuffers(1, &VertexBufferObject);

        glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
        glBufferData(GL_ARRAY_BUFFER, VerticesCount * 2 * 12, VertexBufferData,
GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        delete [] VertexBufferData;
    }

    fclose(File);
    return true;
}

void CScene::Render()
{
    glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 24, (void*)0);

    glEnableClientState(GL_NORMAL_ARRAY);
    glNormalPointer(GL_FLOAT, 24, (void*)12);

    glDrawArrays(GL_TRIANGLES, 0, VerticesCount);

    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

void CScene::Destroy()
{
    if(Vertices != NULL)
    {
        delete [] Vertices;
    }

    if(VertexBufferObject != 0)
    {
        glDeleteBuffers(1, &VertexBufferObject);
    }

    SetDefaults();
}

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

COpenGLRenderer::COpenGLRenderer()
{
    RenderGLUTObjects = true;
    ShowPositionBuffer = false;
    ShowNormalBuffer = false;
    ShowDepthBuffer = false;
    ApplySSAOFilter = true;
    ApplyFXAA = true;
}

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 |= !Preprocess.Load("preprocess.vs", "preprocess.fs");
    Error |= !SSAO.Load("ssao.vs", "ssao.fs");
    Error |= !SSAOFilterH.Load("ssaofilter.vs", "ssaofilterh.fs");
    Error |= !SSAOFilterV.Load("ssaofilter.vs", "ssaofilterv.fs");
    Error |= !DeferredLighting.Load("deferredlighting.vs",
"deferredlighting.fs");
    Error |= !FXAA.Load("FXAA.vert", "FXAA_Extreme_Quality.frag");

    Error |= !Scene.LoadBinary("scene.bin");

    if(Error)
    {
        return false;
    }

    SSAO.UniformLocations = new GLuint[2];
    SSAO.UniformLocations[0] = glGetUniformLocation(SSAO, "Scale");
    SSAO.UniformLocations[1] = glGetUniformLocation(SSAO,
"ProjectionBiasMatrixInverse");

    SSAOFilterH.UniformLocations = new GLuint[1];
    SSAOFilterH.UniformLocations[0] = glGetUniformLocation(SSAOFilterH,
"PixelSizeX");

    SSAOFilterV.UniformLocations = new GLuint[1];
    SSAOFilterV.UniformLocations[0] = glGetUniformLocation(SSAOFilterV,
"PixelSizeY");

    DeferredLighting.UniformLocations = new GLuint[4];
    DeferredLighting.UniformLocations[0] =
glGetUniformLocation(DeferredLighting, "ProjectionBiasMatrixInverse");
    DeferredLighting.UniformLocations[1] =
glGetUniformLocation(DeferredLighting, "ShowPositionBuffer");
    DeferredLighting.UniformLocations[2] =
glGetUniformLocation(DeferredLighting, "ShowNormalBuffer");
    DeferredLighting.UniformLocations[3] =
glGetUniformLocation(DeferredLighting, "ShowDepthBuffer");

    FXAA.UniformLocations = new GLuint[1];
    FXAA.UniformLocations[0] = glGetUniformLocation(FXAA, "RCPFrame");

    glUseProgram(SSAO);
    glUniform1i(glGetUniformLocation(SSAO, "NormalBuffer"), 0);
    glUniform1i(glGetUniformLocation(SSAO, "DepthBuffer"), 1);
    glUniform1i(glGetUniformLocation(SSAO, "RotationTexture"), 2);
    glUniform1f(glGetUniformLocation(SSAO, "Radius"), 0.125f);
    glUniform1f(glGetUniformLocation(SSAO, "Strength"), 2.0f);
    glUniform1f(glGetUniformLocation(SSAO, "ConstantAttenuation"), 1.0f);
    glUniform1f(glGetUniformLocation(SSAO, "LinearAttenuation"), 1.0f);
    glUniform1f(glGetUniformLocation(SSAO, "QuadraticAttenuation"), 0.0f);
    glUseProgram(0);

    float s = 128.0f, e = 131070.0f, fs = 1.0f / s, fe = 1.0f / e, fd = fs -
fe;

    glUseProgram(SSAOFilterH);
    glUniform1i(glGetUniformLocation(SSAOFilterH, "SSAOBuffer"), 0);
```

```
glUniform1i(glGetUniformLocation(SSAOFilterH, "DepthBuffer"), 1);
glUniform1f(glGetUniformLocation(SSAOFilterH, "fs"), fs);
glUniform1f(glGetUniformLocation(SSAOFilterH, "fd"), fd);
glUseProgram(0);

glUseProgram(SSAOFilterV);
glUniform1i(glGetUniformLocation(SSAOFilterV, "SSAOBuffer"), 0);
glUniform1i(glGetUniformLocation(SSAOFilterV, "DepthBuffer"), 1);
glUniform1f(glGetUniformLocation(SSAOFilterV, "fs"), fs);
glUniform1f(glGetUniformLocation(SSAOFilterV, "fd"), fd);
glUseProgram(0);

glUseProgram(DeferredLighting);
glUniform1i(glGetUniformLocation(DeferredLighting, "ColorBuffer"), 0);
glUniform1i(glGetUniformLocation(DeferredLighting, "NormalBuffer"), 1);
glUniform1i(glGetUniformLocation(DeferredLighting, "DepthBuffer"), 2);
glUniform1i(glGetUniformLocation(DeferredLighting, "SSAOBuffer"), 3);
glUseProgram(0);

srand(GetTickCount());

vec2 *Samples = new vec2[16];
float Angle = (float)M_PI_4;

for(int i = 0; i < 16; i++)
{
    Samples[i].x = cos(Angle) * (float)(i + 1) / 16.0f;
    Samples[i].y = sin(Angle) * (float)(i + 1) / 16.0f;

    Angle += (float)M_PI_2;

    if(((i + 1) % 4) == 0) Angle += (float)M_PI_4;
}

glUseProgram(SSAO);
glUniform2fv(glGetUniformLocation(SSAO, "Samples"), 16, (float*)Samples);
glUseProgram(0);

delete [] Samples;

vec4 *RotationTextureData = new vec4[64 * 64];
float RandomAngle = (float)rand() / (float)RAND_MAX * (float)M_PI * 2.0f;

for(int i = 0; i < 64 * 64; i++)
```

```
    {
        RotationTextureData[i].x = cos(RandomAngle) * 0.5f + 0.5f;
        RotationTextureData[i].y = sin(RandomAngle) * 0.5f + 0.5f;
        RotationTextureData[i].z = -sin(RandomAngle) * 0.5f + 0.5f;
        RotationTextureData[i].w = cos(RandomAngle) * 0.5f + 0.5f;

        RandomAngle += (float)rand() / (float)RAND_MAX * (float)M_PI * 2.0f;
    }

    glGenTextures(1, &RotationTexture);
    glBindTexture(GL_TEXTURE_2D, RotationTexture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 64, 64, 0, GL_RGBA, GL_FLOAT,
(float*)RotationTextureData);
    glBindTexture(GL_TEXTURE_2D, 0);

    delete [] RotationTextureData;

    glGenTextures(1, &ColorBuffer);
    glGenTextures(1, &NormalBuffer);
    glGenTextures(1, &DepthBuffer);
    glGenTextures(1, &SSAOBuffer);
    glGenTextures(1, &SSAOFilterBuffer);
    glGenTextures(1, &FXAABuffer);

    glGenFramebuffersEXT(1, &FBO);

    Camera.Look(vec3(-1.0f, 1.75, 1.0f), vec3(0.0f, 1.75, 0.0f));

    return true;
}

void COpenGLRenderer::Render()
{
    GLenum Buffers[] = {GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT};

    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(&Camera.ViewMatrix);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
    glDrawBuffers(2, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
GL_TEXTURE_2D, ColorBuffer, 0);
```

```
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,
GL_TEXTURE_2D, NormalBuffer, 0);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
GL_TEXTURE_2D, DepthBuffer, 0);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);

    glUseProgram(Preprocess);

    Scene.Render();

    glDisable(GL_CULL_FACE);

    if(RenderGLUTObjects)
    {
        glLoadMatrixf(&Camera.ViewMatrix);
        glTranslatef(2.5f, 1.185f, -2.0f);
        glRotatef(33.0f, 0.0f, 1.0f, 0.0f);
        glutSolidTeapot(0.25f);

        glLoadMatrixf(&Camera.ViewMatrix);
        glTranslatef(2.5f, 1.185f, -2.5f);
        glRotatef(180.0f, 0.0f, 1.0f, 0.0f);
        glutSolidTeapot(0.25f);

        glLoadMatrixf(&Camera.ViewMatrix);
        glTranslatef(2.5f, 1.185f, -3.0f);
        glRotatef(-33.0f, 0.0f, 1.0f, 0.0f);
        glutSolidTeapot(0.25f);

        glLoadMatrixf(&Camera.ViewMatrix);
        glTranslatef(-2.5f, 0.25f, -1.0f);
        glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
        glutSolidTorus(0.25f, 0.5f, 64, 64);

        glLoadMatrixf(&Camera.ViewMatrix);
        glTranslatef(-2.5f, 0.25f, 0.0f);
        glutSolidSphere(0.25f, 32, 32);

        glLoadMatrixf(&Camera.ViewMatrix);
        glTranslatef(-2.5f, 0.125f, 0.365f);
        glutSolidSphere(0.125f, 32, 32);
    }

    glUseProgram(0);

    glDisable(GL_DEPTH_TEST);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

    if(!ShowPositionBuffer && !ShowNormalBuffer && !ShowDepthBuffer)
    {
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
        glDrawBuffers(1, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
GL_TEXTURE_2D, SSAOBuffer, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,
GL_TEXTURE_2D, 0, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
GL_TEXTURE_2D, 0, 0);

        glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D,
NormalBuffer);
        glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D,
DepthBuffer);
        glActiveTexture(GL_TEXTURE2); glBindTexture(GL_TEXTURE_2D,
RotationTexture);
        glUseProgram(SSAO);
        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);

        if(ApplySSAOFilter)
        {
            glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
            glDrawBuffers(1, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, SSAOFilterBuffer, 0);
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, 0, 0);
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_DEPTH_ATTACHMENT_EXT, GL_TEXTURE_2D, 0, 0);

            glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D,
SSAOBuffer);
            glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D,
DepthBuffer);
            glUseProgram(SSAOFilterH);
            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);

            glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

            glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
            glDrawBuffers(1, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, SSAOBuffer, 0);
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, 0, 0);
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_DEPTH_ATTACHMENT_EXT, GL_TEXTURE_2D, 0, 0);

            glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D,
SSAOFilterBuffer);
            glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D,
DepthBuffer);
            glUseProgram(SSAOFilterV);
            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);

            glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
        }
    }

    if(ApplyFXAA)
    {
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
        glDrawBuffers(1, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
GL_TEXTURE_2D, FXAABuffer, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,
GL_TEXTURE_2D, 0, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
GL_TEXTURE_2D, 0, 0);
    }

    glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, ColorBuffer);
    glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, NormalBuffer);
    glActiveTexture(GL_TEXTURE2); glBindTexture(GL_TEXTURE_2D, DepthBuffer);
    glActiveTexture(GL_TEXTURE3); glBindTexture(GL_TEXTURE_2D, SSAOBuffer);
    glUseProgram(DeferredLighting);
    glUniform1i(DeferredLighting.UniformLocations[1], ShowPositionBuffer);
    glUniform1i(DeferredLighting.UniformLocations[2], ShowNormalBuffer);
    glUniform1i(DeferredLighting.UniformLocations[3], ShowDepthBuffer);
    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);

    if(ApplyFXAA)
    {
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
    }
    if(ApplyFXAA)
    {
        glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, FXAABuffer);
        glUseProgram(FXAA);
        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_TEXTURE0); glBindTexture(GL_TEXTURE_2D, 0);
    }
}

void COpenGLRenderer::Animate(float FrameTime)
{
}

void COpenGLRenderer::Resize(int Width, int Height)
{
    this->Width = Width;
    this->Height = Height;

    glViewport(0, 0, Width, Height);

    Camera.SetPerspective(45.0f, (float)Width / (float)Height, 0.125f, 512.0f);

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(&Camera.ProjectionMatrix);

    mat4x4 ProjectionBiasMatrixInverse = Camera.ProjectionMatrixInverse *
BiasMatrixInverse;

    glBindTexture(GL_TEXTURE_2D, ColorBuffer);
    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, NormalBuffer);
    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, DepthBuffer);
    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);

    glBindTexture(GL_TEXTURE_2D, SSAOBuffer);
    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_RGBA8, Width, Height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
    glBindTexture(GL_TEXTURE_2D, 0);

    glBindTexture(GL_TEXTURE_2D, SSAOFilterBuffer);
    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_RGBA8, Width, Height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
    glBindTexture(GL_TEXTURE_2D, 0);

    glBindTexture(GL_TEXTURE_2D, FXAABuffer);
    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, Width, Height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
    glBindTexture(GL_TEXTURE_2D, 0);
    glUseProgram(SSAO);
    glUniform2f(SSAO.UniformLocations[0], (float)Width / 64.0f, (float)Height / 
64.0f);
    glUniformMatrix4fv(SSAO.UniformLocations[1], 1, GL_FALSE, 
&ProjectionBiasMatrixInverse);
    glUseProgram(0);

    glUseProgram(SSAOFilterH);
    glUniform1f(SSAOFilterH.UniformLocations[0], 1.0f / (float)Width);
    glUseProgram(SSAOFilterV);
    glUniform1f(SSAOFilterV.UniformLocations[0], 1.0f / (float)Height);
    glUseProgram(0);

    glUseProgram(DeferredLighting);
    glUniformMatrix4fv(DeferredLighting.UniformLocations[0], 1, GL_FALSE, 
&ProjectionBiasMatrixInverse);
    glUseProgram(0);

    glUseProgram(FXAA);
    glUniform2f(FXAA.UniformLocations[0], 1.0f / (float)Width, 1.0f / 
(float)Height);
    glUseProgram(0);
}

void COpenGLRenderer::Destroy()
{
    Preprocess.Destroy();
    SSAO.Destroy();
    SSAOFilterH.Destroy();
    SSAOFilterV.Destroy();
    DeferredLighting.Destroy();
    FXAA.Destroy();

    Scene.Destroy();

    glDeleteTextures(1, &RotationTexture);

    glDeleteTextures(1, &ColorBuffer);
    glDeleteTextures(1, &NormalBuffer);
    glDeleteTextures(1, &DepthBuffer);
    glDeleteTextures(1, &SSAOBuffer);
    glDeleteTextures(1, &SSAOFilterBuffer);
    glDeleteTextures(1, &FXAABuffer);
    if(GLEW_EXT_framebuffer_object)
    {
        glDeleteFramebuffersEXT(1, &FBO);
    }
}

void COpenGLRenderer::CheckCameraKeys(float FrameTime)
{
    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));
    }
}

void COpenGLRenderer::OnKeyDown(UINT Key)
{
    switch(Key)
    {
        case VK_F1:
            RenderGLUTObjects = !RenderGLUTObjects;
            break;

        case VK_F2:
            if(!ShowPositionBuffer && !ShowNormalBuffer && !ShowDepthBuffer)
            {
                ShowPositionBuffer = true;
                ShowNormalBuffer = false;
                ShowDepthBuffer = false;
            }
            else if(ShowPositionBuffer && !ShowNormalBuffer&&!ShowDepthBuffer)
            {
                ShowPositionBuffer = false;
```

```
                 ShowNormalBuffer = true;
                ShowDepthBuffer = false;
            }
            else if(!ShowPositionBuffer && ShowNormalBuffer&& !ShowDepthBuffer)
            {
                ShowPositionBuffer = false;
                ShowNormalBuffer = false;
                 ShowDepthBuffer = true; }
            else if(!ShowPositionBuffer && !ShowNormalBuffer &&ShowDepthBuffer)
            {
                ShowPositionBuffer = false;
                ShowNormalBuffer = false;
                ShowDepthBuffer = false;
            }
            break;

        case VK_F3:
            ApplySSAOFilter = !ApplySSAOFilter;
            break;

        case VK_F4:
            ApplyFXAA = !ApplyFXAA;
            break;
    }
}

void COpenGLRenderer::OnLButtonDown(int X, int Y)
{
    LastClickedX = X;
    LastClickedY = Y;
}

void COpenGLRenderer::OnLButtonUp(int X, int Y)
{
    if(X == LastClickedX && Y == LastClickedY)
    {
    }
}

void COpenGLRenderer::OnMouseMove(int X, int Y)
{
    if(GetKeyState(VK_RBUTTON) & 0x80)
    {
        Camera.OnMouseMove(LastX - X, LastY - Y);
    }

    LastX = X;
    LastY = Y;
}

void COpenGLRenderer::OnMouseWheel(short zDelta)
{
    Camera.OnMouseWheel(zDelta);
}

void COpenGLRenderer::OnRButtonDown(int X, int Y)
{
    LastClickedX = X;
    LastClickedY = Y;
}

void COpenGLRenderer::OnRButtonUp(int X, int Y)
{
    if(X == LastClickedX && Y == LastClickedY)
    {
    }
}
```

第三章 **First person camera , collision detection , gravity , jump, crouch**

第一节 Shader Source

<1>defferedlighting.vs

```
#version 120
void main()
{
	gl_TexCoord[0] = gl_Vertex;
	gl_Position = gl_Vertex * 2.0 - 1.0;
}
```

Defferedlighting.fs

```
#version 120

uniform sampler2D ColorBuffer, NormalBuffer, DepthBuffer, SSAOBuffer;
uniform mat4x4 ProjectionBiasMatrixInverse;
uniform bool Lighting, ApplySSAO;
void main()
{
	gl_FragColor = texture2D(ColorBuffer, gl_TexCoord[0].st);

	float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;

	if(Depth < 1.0)
	{
		vec3 Normal = normalize(texture2D(NormalBuffer, gl_TexCoord[0].st).rgb * 2.0 - 1.0);

		vec4 Position = ProjectionBiasMatrixInverse * vec4(gl_TexCoord[0].st, Depth, 1.0);
		Position /= Position.w;

		float SSAO = ApplySSAO ? texture2D(SSAOBuffer, gl_TexCoord[0].st).r : 1.0;
		if(Lighting)
		{
			vec3 LightDirection = normalize(vec3(0.0) - Position.xyz);
			float NdotLD = max(dot(Normal, LightDirection), 0.0);
			gl_FragColor.rgb *= 0.5 * SSAO + 0.5 * NdotLD;
		}
		else
		{
			gl_FragColor.rgb *= SSAO;
		}
	}
}
```

<2>:预处理 preprocess.vs

```
#version 120
```

```
varying vec3 Normal;
void main()
{
    gl_FrontColor = gl_Color;
    gl_TexCoord[0] = gl_MultiTexCoord0;
    Normal = gl_NormalMatrix * gl_Normal;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
#version 12
preprocess.fs
uniform sampler2D Texture;
uniform bool Texturing;

varying vec3 Normal;

void main()
{
    gl_FragData[0] = gl_Color;
    if(Texturing) gl_FragData[0] *= texture2D(Texture, gl_TexCoord[0].st);
    gl_FragData[1] = vec4(normalize(Normal) * 0.5 + 0.5, 1.0);
}
<2>SSAO.vs
#version 120
uniform vec2 Scale;
void main()
{
    gl_TexCoord[0] = gl_Vertex;
    gl_TexCoord[1] = vec4(gl_Vertex.xy * Scale, gl_Vertex.zw);
    gl_Position = gl_Vertex * 2.0 - 1.0;
}
SSAO.fs
#version 120
uniform sampler2D NormalBuffer, DepthBuffer, RotationTexture;
uniform mat4x4 ProjectionBiasMatrixInverse;
uniform vec2 Samples[16];
uniform float Radius, Strength, ConstantAttenuation, LinearAttenuation,
QuadraticAttenuation;
void main()
{
    float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;
    if(Depth < 1.0)
    {
        vec3 Normal = normalize(texture2D(NormalBuffer, gl_TexCoord[0].st).rgb * 2.0 -
1.0);
        vec4 Position = ProjectionBiasMatrixInverse * vec4(gl_TexCoord[0].st, Depth, 1.0);
        Position.xyz /= Position.w;

        if(dot(Normal, Position.xyz) > 0.0)
        {
            Normal = -Normal;
        }
        vec4 ScaleRotationVector = normalize(texture2D(RotationTexture,
gl_TexCoord[1].st) * 2.0 - 1.0) * Radius;

        mat2x2 ScaleRotationMatrix = mat2x2(ScaleRotationVector.xy,
ScaleRotationVector.zw);
        float SSAO = 0.0;
        for(int i = 0; i < 16; i++)
        {
            vec2 TexCoord = clamp(gl_TexCoord[0].st + ScaleRotationMatrix * Samples[i],
0.0, 0.999999);
            float SampleDepth = texture2D(DepthBuffer, TexCoord).r;

            vec4 SamplePosition = ProjectionBiasMatrixInverse * vec4(TexCoord,
SampleDepth, 1.0);
            SamplePosition.xyz /= SamplePosition.w;

            vec3 P2SP = SamplePosition.xyz - Position.xyz;
            float Distance2 = dot(P2SP, P2SP);
            float Distance = sqrt(Distance2);
            float NdotP2SP = dot(Normal, P2SP) / Distance;
            if(NdotP2SP > 0.342)
            {
                SSAO += NdotP2SP / (ConstantAttenuation + Distance *
LinearAttenuation + Distance2 * QuadraticAttenuation);
            }
        }
        gl_FragColor = vec4(vec3(1.0 - SSAO * 0.0625 * Strength), 1.0);
    }
    else
    {
        gl_FragColor = vec4(vec3(0.0), 1.0);
    }
}
```

SSAOFilter.vs

```
#version 120
void main()
```

```
{
    gl_TexCoord[0] = gl_Vertex;
    gl_Position = gl_Vertex * 2.0 - 1.0;
}
SSAOFilterV.fs
#version 120
uniform sampler2D SSAOBuffer, DepthBuffer;
uniform float PixelSizeY, fs, fd;
float Offsets[8] = float[](-4.0, -3.0, -2.0, -1.0, 1.0, 2.0, 3.0, 4.0);
float BlurWeights[8] = float[](1.0, 2.0, 3.0, 4.0, 4.0, 3.0, 2.0, 1.0);
void main()
{
    float BlurWeightsSum = 5.0;
    float SSAO = texture2D(SSAOBuffer, gl_TexCoord[0].st).r * BlurWeightsSum;
    float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;
    float Factor = fs - fd * Depth;
    for(int i = 0; i < 8; i++)
    {
        vec2 TexCoord = vec2(gl_TexCoord[0].s, gl_TexCoord[0].t + Offsets[i] * PixelSizeY);
        float DepthDifference = abs(Depth - texture2D(DepthBuffer, TexCoord).r);
        if(DepthDifference < Factor)
        {
            SSAO += texture2D(SSAOBuffer, TexCoord).r * BlurWeights[i];
            BlurWeightsSum += BlurWeights[i];
        }
    }
    gl_FragColor = vec4(vec3(SSAO / BlurWeightsSum), 1.0);
}
SSAOFilterH.fs
#version 120
uniform sampler2D SSAOBuffer, DepthBuffer;
uniform float PixelSizeX, fs, fd;
float Offsets[8] = float[](-4.0, -3.0, -2.0, -1.0, 1.0, 2.0, 3.0, 4.0);
float BlurWeights[8] = float[](1.0, 2.0, 3.0, 4.0, 4.0, 3.0, 2.0, 1.0);
void main()
{
    float BlurWeightsSum = 5.0;

    float SSAO = texture2D(SSAOBuffer, gl_TexCoord[0].st).r * BlurWeightsSum;
    float Depth = texture2D(DepthBuffer, gl_TexCoord[0].st).r;

    float Factor = fs - fd * Depth;

    for(int i = 0; i < 8; i++)
```

```
	{
		vec2 TexCoord = vec2(gl_TexCoord[0].s + Offsets[i] * PixelSizeX, gl_TexCoord[0].t);
		float DepthDifference = abs(Depth - texture2D(DepthBuffer, TexCoord).r);
		if(DepthDifference < Factor)
		{
			SSAO += texture2D(SSAOBuffer, TexCoord).r * BlurWeights[i];
			BlurWeightsSum += BlurWeights[i];
		}
	}
	gl_FragColor = vec4(vec3(SSAO / BlurWeightsSum), 1.0);
}
```

第二节 Source Code Header

```
class CTriangle
{
public:
	vec3 A, B, C, M;
	vec3 AB, BC, CA;
	float LAB, LBC, LCA;
	vec3 N, NH, N1, N2, N3;
	float D, NdotNH, D1, D2, D3;
	vec3 HPNAB, HPNBC, HPNCA;
	float HPDAB, HPDBC, HPDCA;
	vec3 VPNAB, VPNBC, VPNCA;
	float VPDAB, VPDBC, VPDCA;

public:
	CTriangle();
	CTriangle(const vec3 &A, const vec3 &B, const vec3 &C);
	~CTriangle();

public:
	void Set(const vec3 &A, const vec3 &B, const vec3 &C);

public:
	bool Inside(const vec3 &Point);
	bool RayTriangleIntersectionTest(const vec3 &RayOrigin, const vec3 &RayDirection, float &MinDistance, vec3 &IntersectionPoint);
	bool GetHeightAbove(const vec3 &EyePosition, float &MinDistance, float &Height);
	bool GetHeightUnder(const vec3 &EyePosition, float EyeKneeDistance, float &MinDistance, float &Height);
	bool IntersectionTest(const vec3 &EyePositionA, const vec3 &EyePositionB, const vec3 &Direction, float EyeKneeDistance, float ClosestDistance, const vec3 &PN, float PD, float &MinDistance, vec3 &Compensation);
	bool DistanceTest(const vec3 &EyePositionB, float EyeKneeDistance, float
ClosestDistance, float &MinDistance, vec3 &Compensation);
};

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

class CCollisionDetector
{
private:
	CTriangle *Triangles;
	int TrianglesCount;

private:
	float EyeHeight, EyeKneeDistance, ClosestDistance;

private:
	float EH, EHD2, EKD, EKDD2;

private:
	float FallSpeed;
	int CrouchState;

public:
	CCollisionDetector();
	~CCollisionDetector();

private:
	void SetDefaults();

public:
	void Init(vec3 *Vertices, int VerticesCount, float EyeHeight, float EyeKneeDistance, float
ClosestDistance);
	void Destroy();

public:
	void Jump();
	void Crouch();

private:
	bool GetHeightAbove(const vec3 &EyePosition, float &MinDistance, float &Height);
	bool GetHeightUnder(const vec3 &EyePosition, float EyeKneeDistance, float
&MinDistance, float &Height);
	bool IntersectionTest(const vec3 &EyePositionA, const vec3 &EyePositionB, const vec3
&Direction, float EyeKneeDistance, float ClosestDistance, const vec3 &PN, float PD, float
&MinDistance, vec3 &Compensation);
	bool DistanceTest(const vec3 &EyePositionB, float EyeKneeDistance, float
ClosestDistance, float &MinDistance, vec3 &Compensation);

public:
	void CheckHorizontalCollision(const vec3 &EyePosition, vec3 &Movement);
	void CheckVerticalCollision(const vec3 &EyePosition, float FrameTime, vec3
&Movement);
};

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

class CScene
{
private:
	vec3 *Vertices;
	int VerticesCount;

private:
	GLuint VertexBufferObject;

private:
	CTexture Texture;

public:
	CScene();
	~CScene();

private:
	void SetDefaults();

public:
	bool LoadBinary(const char *FileName);
	void Render();
	void Destroy();

public:
	vec3 *GetVertices();
	int GetVerticesCount();
};

// ----------------------------------------------------------------------------
```

```
-----------------------------------------------

class COpenGLRenderer
{
private:
	int LastX, LastY, LastClickedX, LastClickedY;

private:
	int Width, Height;

private:
	CCamera Camera;

private:
	CCollisionDetector CollisionDetector;

private:
	CShaderProgram Preprocess, SSAO, SSAOFilterH, SSAOFilterV, DeferredLighting, FXAA;
	GLuint RotationTexture, ColorBuffer, NormalBuffer, DepthBuffer, SSAOBuffer,
SSAOFilterBuffer, FXAABuffer, FBO;

private:
	CScene Scene;

private:
	bool Texturing, Lighting, ApplySSAO, ApplyFXAA;

public:
	CString Text;

public:
	COpenGLRenderer();
	~COpenGLRenderer();

public:
	bool Init();
	void Render();
	void Animate(float FrameTime);
	void Resize(int Width, int Height);
	void Destroy();

public:
	void CheckCameraKeys(float FrameTime);
public:
	void OnKeyDown(UINT Key);
	void OnLButtonDown(int X, int Y);
	void OnLButtonUp(int X, int Y);
	void OnMouseMove(int X, int Y);
	void OnMouseWheel(short zDelta);
	void OnRButtonDown(int X, int Y);
	void OnRButtonUp(int X, int Y);
};
```

第三节 Source Code Cpp

```
CTriangle::CTriangle()
{
}

CTriangle::CTriangle(const vec3 &A, const vec3 &B, const vec3 &C)
{
	Set(A, B, C);
}

CTriangle::~CTriangle()
{
}

void CTriangle::Set(const vec3 &A, const vec3 &B, const vec3 &C)
{
	this->A = A;
	this->B = B;
	this->C = C;

	M = (A + B + C) / 3.0f;

	AB = B - A;
	BC = C - B;
	CA = A - C;

	LAB = length(AB);
	LBC = length(BC);
	LCA = length(CA);

	AB /= LAB;
	BC /= LBC;
	CA /= LCA;
```

```
    N = normalize(cross(AB, -CA));
    D = -dot(N, A);

    NH = (N.y > -1.0f && N.y < 1.0f) ? normalize(vec3(N.x, 0.0f, N.z)) : vec3(0.0f);
    NdotNH = dot(N, NH);

    N1 = normalize(cross(N, AB));
    D1 = -dot(N1, A);

    N2 = normalize(cross(N, BC));
    D2 = -dot(N2, B);

    N3 = normalize(cross(N, CA));
    D3 = -dot(N3, C);

    HPNAB = (AB.y > -1.0f && AB.y < 1.0f) ? normalize(vec3(0.0f, 1.0f, 0.0f) - AB * AB.y) : vec3(0.0f);
    HPDAB = -dot(A, HPNAB);
    VPNAB = cross(AB, HPNAB);
    VPDAB = -dot(A, VPNAB);

    HPNBC = (BC.y > -1.0f && BC.y < 1.0f) ? normalize(vec3(0.0f, 1.0f, 0.0f) - BC * BC.y) : vec3(0.0f);
    HPDBC = -dot(B, HPNBC);
    VPNBC = cross(BC, HPNBC);
    VPDBC = -dot(B, VPNBC);

    HPNCA = (CA.y > -1.0f && CA.y < 1.0f) ? normalize(vec3(0.0f, 1.0f, 0.0f) - CA * CA.y) : vec3(0.0f);
    HPDCA = -dot(C, HPNCA);
    VPNCA = cross(CA, HPNCA);
    VPDCA = -dot(C, VPNCA);
}

bool CTriangle::Inside(const vec3 &Point)
{
    if(dot(N1, Point) + D1 < 0.0f) return false;
    if(dot(N2, Point) + D2 < 0.0f) return false;
    if(dot(N3, Point) + D3 < 0.0f) return false;

    return true;
}
```

```
bool CTriangle::RayTriangleIntersectionTest(const vec3 &RayOrigin, const vec3
&RayDirection, float &MinDistance, vec3 &IntersectionPoint)
{
    float NdotRD = -dot(N, RayDirection);

    if(NdotRD > 0.0f)
    {
        float DistanceFromPlane = (dot(N, RayOrigin) + D) / NdotRD;

        if(DistanceFromPlane > 0.0f && DistanceFromPlane < MinDistance)
        {
            vec3 PointOnPlane = RayOrigin + RayDirection * DistanceFromPlane;

            if(Inside(PointOnPlane))
            {
                MinDistance = DistanceFromPlane;
                IntersectionPoint = PointOnPlane;

                return true;
            }
        }
    }

    return false;
}

bool CTriangle::GetHeightAbove(const vec3 &EyePosition, float &MinDistance, float
&Height)
{
    float NdotRD = -N.y;

    if(NdotRD > 0.0f)
    {
        float DistanceFromPlane = (dot(N, EyePosition) + D) / NdotRD;

        if(DistanceFromPlane > 0.0f && DistanceFromPlane < MinDistance)
        {
            vec3 PointOnPlane = vec3(EyePosition.x, EyePosition.y + DistanceFromPlane,
EyePosition.z);

            if(Inside(PointOnPlane))
            {
                MinDistance = DistanceFromPlane;
                Height = PointOnPlane.y;
                return true;
            }
        }
    }

    return false;
}

bool CTriangle::GetHeightUnder(const vec3 &EyePosition, float EyeKneeDistance, float
&MinDistance, float &Height)
{
    float NdotRD = N.y;

    if(NdotRD > 0.0f)
    {
        float DistanceFromPlane = (dot(N, EyePosition) + D) / NdotRD;

        if(DistanceFromPlane > EyeKneeDistance && DistanceFromPlane < MinDistance)
        {
            vec3 PointOnPlane = vec3(EyePosition.x, EyePosition.y - DistanceFromPlane,
EyePosition.z);

            if(Inside(PointOnPlane))
            {
                MinDistance = DistanceFromPlane;
                Height = PointOnPlane.y;

                return true;
            }
        }
    }

    return false;
}

bool CTriangle::IntersectionTest(const vec3 &EyePositionA, const vec3 &EyePositionB, const
vec3 &Direction, float EyeKneeDistance, float ClosestDistance, const vec3 &PN, float PD,
float &MinDistance, vec3 &Compensation)
{
    bool IntersectionTestPassed = false;

    if(NdotNH > 0.0f)
    {
        float NdotD = -dot(N, Direction);

        if(NdotD > 0.0f)
        {
            float DistanceFromPlane = (dot(N, EyePositionA) + D) / NdotD;

            if(DistanceFromPlane > 0.0f && DistanceFromPlane < MinDistance)
            {
                vec3 PointOnPlane = EyePositionA + Direction * DistanceFromPlane;

                if(Inside(PointOnPlane))
                {
                    IntersectionTestPassed = true;
                    MinDistance = DistanceFromPlane;
                    Compensation = PointOnPlane - EyePositionB + NH *
(ClosestDistance / NdotNH);
                }
            }
        }
    }

    vec3 *Vertices = (vec3*)&A;
    vec3 *Edges = (vec3*)&AB;
    float *EdgesLengths = &LAB;
    vec3 *VPNs = (vec3*)&VPNAB;

    for(int i = 0; i < 3; i++)
    {
        float PNdotE = -dot(PN, Edges[i]);

        if(PNdotE != 0.0f)
        {
            float DistanceFromPlane = (dot(PN, Vertices[i]) + PD) / PNdotE;

            if(DistanceFromPlane > 0.0f && DistanceFromPlane < EdgesLengths[i])
            {
                vec3 PointOnPlane = Vertices[i] + Edges[i] * DistanceFromPlane;

                vec3 EPAPOP = PointOnPlane - EyePositionA;

                float DistanceV = -EPAPOP.y;

                if(DistanceV > 0.0f && DistanceV < EyeKneeDistance)
                {
                    float DistanceH = dot(Direction, EPAPOP);

                    if(DistanceH > 0.0f && DistanceH < MinDistance)
                    {
                        IntersectionTestPassed = true;
                        MinDistance = DistanceH;
                        Compensation = vec3(PointOnPlane.x - EyePositionB.x, 0.0f,
PointOnPlane.z - EyePositionB.z);
                        float VPNdotD = -dot(VPNs[i], Direction);
                        if(VPNdotD > 0.0f) Compensation += VPNs[i] * ClosestDistance;
                        if(VPNdotD < 0.0f) Compensation -= VPNs[i] * ClosestDistance;
                    }
                }
            }
        }
    }

    return IntersectionTestPassed;
}

bool CTriangle::DistanceTest(const vec3 &EyePositionB, float EyeKneeDistance, float
ClosestDistance, float &MinDistance, vec3 &Compensation)
{
    bool DistanceTestFailed = false;

    if(NdotNH > 0.0f)
    {
        float DistanceFromPlane = dot(N, EyePositionB) + D;

        if(DistanceFromPlane > 0.0f && DistanceFromPlane < MinDistance)
        {
            if(Inside(EyePositionB))
            {
                DistanceTestFailed = true;
                MinDistance = DistanceFromPlane;
                Compensation = NH * ((ClosestDistance - DistanceFromPlane) /
NdotNH);
            }
        }
    }

    vec3 *Vertices = (vec3*)&A;
    vec3 *Edges = (vec3*)&AB;
    float *EdgesLengths = &LAB;
    for(int i = 0; i < 3; i++)
    {
        vec3 EPBD = EyePositionB - Vertices[i];

        float EdotEPBD = dot(Edges[i], EPBD);

        if(EdotEPBD > 0.0f && EdotEPBD < EdgesLengths[i])
        {
            vec3 N = EPBD - Edges[i] * EdotEPBD;

            if(N.x != 0.0f || N.z != 0.0f)
            {
                float DistanceFromEdge = length(N);

                if(DistanceFromEdge > 0.0f && DistanceFromEdge < MinDistance)
                {
                    DistanceTestFailed = true;
                    MinDistance = DistanceFromEdge;
                    N /= DistanceFromEdge;
                    vec3 NH = normalize(vec3(N.x, 0.0f, N.z));
                    float NdotNH = dot(N, NH);
                    Compensation = NH * ((ClosestDistance - DistanceFromEdge) /
NdotNH);
                }
            }
        }
    }

    for(int i = 0; i < 3; i++)
    {
        vec3 N = EyePositionB - Vertices[i];

        if(N.x != 0.0f || N.z != 0.0f)
        {
            float DistanceFromVertex = length(N);

            if(DistanceFromVertex > 0.0f && DistanceFromVertex < MinDistance)
            {
                DistanceTestFailed = true;
                MinDistance = DistanceFromVertex;
                N /= DistanceFromVertex;
                vec3 NH = normalize(vec3(N.x, 0.0f, N.z));
                float NdotNH = dot(N, NH);
                Compensation = NH * ((ClosestDistance - DistanceFromVertex) /
NdotNH);
            }
        }
    }

    vec3 *HPNs = (vec3*)&HPNAB;
    float *HPDs = &HPDAB;
    vec3 *VPNs = (vec3*)&VPNAB;
    float *VPDs = &VPDAB;

    for(int i = 0; i < 3; i++)
    {
        if(HPNs[i].y > 0.0f)
        {
            float DistanceFromHorizontalPlane = (dot(HPNs[i], EyePositionB) + HPDs[i]) /
HPNs[i].y;

            if(DistanceFromHorizontalPlane > 0.0f && DistanceFromHorizontalPlane <
EyeKneeDistance)
            {
                float DistanceFromVerticalPlane = dot(VPNs[i], EyePositionB) + VPDs[i];

                if(DistanceFromVerticalPlane > 0.0f && DistanceFromVerticalPlane <
MinDistance)
                {
                    vec3 PointOnHorizontalPlane = vec3(EyePositionB.x, EyePositionB.y
- DistanceFromHorizontalPlane, EyePositionB.z);

                    float EdotPOHPD = dot(Edges[i], PointOnHorizontalPlane -
Vertices[i]);

                    if(EdotPOHPD > 0.0f && EdotPOHPD < EdgesLengths[i])
                    {
                        DistanceTestFailed = true;
                        MinDistance = DistanceFromVerticalPlane;
                        Compensation = VPNs[i] * (ClosestDistance -
DistanceFromVerticalPlane);
                    }
                }
            }
        }
    }
    for(int i = 0; i < 3; i++)
    {
        vec3 EPBD = Vertices[i] - EyePositionB;

        float EdotEPBD = -EPBD.y;

        if(EdotEPBD > 0.0f && EdotEPBD < EyeKneeDistance)
        {
            vec3 N = vec3(EPBD.x, EPBD.y + EdotEPBD, EPBD.z);

            float DistanceFromVertex = length(N);

            if(DistanceFromVertex > 0.0f && DistanceFromVertex < MinDistance)
            {
                DistanceTestFailed = true;
                MinDistance = DistanceFromVertex;
                N /= DistanceFromVertex;
                Compensation = N * (DistanceFromVertex - ClosestDistance);
            }
        }
    }

    return DistanceTestFailed;
}

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

CCollisionDetector::CCollisionDetector()
{
    SetDefaults();
}

CCollisionDetector::~CCollisionDetector()
{
}

void CCollisionDetector::SetDefaults()
{
    Triangles = NULL;
    TrianglesCount = 0;

    EyeHeight = 0.0f;
    EyeKneeDistance = 0.0f;
```

```
	ClosestDistance = 0.0f;

	EH = 0.0f;
	EHD2 = 0.0f;
	EKD = 0.0f;
	EKDD2 = 0.0f;

	FallSpeed = 0.0f;
	CrouchState = 0;
}

void CCollisionDetector::Init(vec3 *Vertices, int VerticesCount, float EyeHeight, float
EyeKneeDistance, float ClosestDistance)
{
	Destroy();

	this->EyeHeight = EyeHeight;
	this->EyeKneeDistance = EyeKneeDistance;
	this->ClosestDistance = ClosestDistance;

	EH = EyeHeight;
	EHD2 = EyeHeight / 2.0f;
	EKD = EyeKneeDistance;
	EKDD2 = EyeKneeDistance / 2.0f;

	if(Vertices != NULL && VerticesCount > 0)
	{
		TrianglesCount = VerticesCount / 3;

		Triangles = new CTriangle[TrianglesCount];

		for(int i = 0; i < TrianglesCount; i++)
		{
			Triangles[i].Set(Vertices[i * 3 + 0], Vertices[i * 3 + 1], Vertices[i * 3 + 2]);
		}
	}
}

void CCollisionDetector::Destroy()
{
	if(Triangles != NULL)
	{
		delete [] Triangles;
	}
	SetDefaults();
}

void CCollisionDetector::Jump()
{
	if(CrouchState == 0)
	{
		if(FallSpeed == 0.0f)
		{
			FallSpeed = -9.82f / 3.0f;
		}
	}
	else
	{
		CrouchState = 2;
	}
}

void CCollisionDetector::Crouch()
{
	if(CrouchState == 0)
	{
		EyeHeight = EHD2;
		EyeKneeDistance = EKDD2;
		CrouchState = 1;
	}
	else if(FallSpeed < 0.0f)
	{
		if(CrouchState == 1)
		{
			EyeHeight = EH;
			EyeKneeDistance = EKD;
			CrouchState = 0;
		}
	}
	else
	{
		if(CrouchState == 1)
		{
			CrouchState = 2;
		}
		else if(CrouchState == 2)
		{
```

```
            EyeHeight = EHD2;
            EyeKneeDistance = EKDD2;
            CrouchState = 1;
        }
    }
}

bool CCollisionDetector::GetHeightAbove(const vec3 &EyePositionA, float &MinDistance,
float &Height)
{
    bool HeightFound = false;

    for(int i = 0; i < TrianglesCount; i++)
    {
        HeightFound |= Triangles[i].GetHeightAbove(EyePositionA, MinDistance, Height);
    }

    return HeightFound;
}

bool CCollisionDetector::GetHeightUnder(const vec3 &EyePositionA, float EyeKneeDistance,
float &MinDistance, float &Height)
{
    bool HeightFound = false;

    for(int i = 0; i < TrianglesCount; i++)
    {
        HeightFound |= Triangles[i].GetHeightUnder(EyePositionA, EyeKneeDistance,
MinDistance, Height);
    }

    return HeightFound;
}

bool CCollisionDetector::IntersectionTest(const vec3 &EyePositionA, const vec3
&EyePositionB, const vec3 &Direction, float EyeKneeDistance, float ClosestDistance, const
vec3 &PN, float PD, float &MinDistance, vec3 &Compensation)
{
    bool IntersectionTestPassed = false;

    for(int i = 0; i < TrianglesCount; i++)
    {
        IntersectionTestPassed |= Triangles[i].IntersectionTest(EyePositionA, EyePositionB,
Direction, EyeKneeDistance, ClosestDistance, PN, PD, MinDistance, Compensation);
```

```
	}

	return IntersectionTestPassed;
}

bool CCollisionDetector::DistanceTest(const vec3 &EyePositionB, float EyeKneeDistance,
float ClosestDistance, float &MinDistance, vec3 &Compensation)
{
	bool DistanceTestFailed = false;

	for(int i = 0; i < TrianglesCount; i++)
	{
		DistanceTestFailed |= Triangles[i].DistanceTest(EyePositionB, EyeKneeDistance,
ClosestDistance, MinDistance, Compensation);
	}

	return DistanceTestFailed;
}

void CCollisionDetector::CheckHorizontalCollision(const vec3 &EyePosition, vec3
&Movement)
{
	if(CrouchState != 0)
	{
		Movement *= 0.5f;
	}

	int Depth = 0;

	TestAgain:

	if(Depth < 16)
	{
		vec3 EyePositionA = EyePosition;
		float Length = length(Movement);
		vec3 Direction = Movement / Length;
		vec3 EyePositionB = EyePositionA + Movement;

		if(Length > ClosestDistance)
		{
			vec3 PN = cross(Direction, vec3(0.0f, -1.0f, 0.0f));
			float PD = -dot(PN, EyePositionA);
			float Distance = Length;
			vec3 Compensation;
```

```
            if(IntersectionTest(EyePositionA, EyePositionB, Direction, EyeKneeDistance,
ClosestDistance, PN, PD, Distance, Compensation))
            {
                Movement += Compensation;

                Depth++;

                goto TestAgain;
            }
        }

        float Distance = ClosestDistance;
        vec3 Compensation;

        if(DistanceTest(EyePositionB, EyeKneeDistance, ClosestDistance, Distance,
Compensation))
        {
            Movement += Compensation;

            Depth++;

            goto TestAgain;
        }
    }
}

void CCollisionDetector::CheckVerticalCollision(const vec3 &EyePosition, float FrameTime,
vec3 &Movement)
{
    if(CrouchState == 2)
    {
        float DistanceAbove = EH - EyeHeight + ClosestDistance, HeightAbove;

        if(!GetHeightAbove(EyePosition, DistanceAbove, HeightAbove))
        {
            EyeHeight += EH * 2.0f * FrameTime;
            EyeKneeDistance = EyeHeight * EKD / EH;

            if(EyeHeight >= EH)
            {
                EyeHeight = EH;
                EyeKneeDistance = EKD;
                CrouchState = 0;
            }
        }
    }

    float DistanceUnder = 1048576.0f, HeightUnder = 0.0f;

    GetHeightUnder(EyePosition, EyeKneeDistance, DistanceUnder, HeightUnder);

    float EPYMEH = EyePosition.y - EyeHeight;

    if(HeightUnder < EPYMEH || FallSpeed < 0.0f)
    {
        FallSpeed += 9.82f * FrameTime;

        float Distance = FallSpeed * FrameTime;

        if(FallSpeed < 0.0f)
        {
            float DistanceAbove = ClosestDistance - Distance, HeightAbove;

            if(GetHeightAbove(EyePosition, DistanceAbove, HeightAbove))
            {
                Distance = DistanceAbove - ClosestDistance;
                FallSpeed = 0.0f;
            }
        }

        float EPYMEHMHU = EPYMEH - HeightUnder;

        if(Distance > EPYMEHMHU)
        {
            Distance = EPYMEHMHU;
        }

        Movement = vec3(0.0f, -Distance, 0.0f);
    }
    else
    {
        FallSpeed = 0.0f;

        float HUMEPYMEH = HeightUnder - EPYMEH;

        if(HUMEPYMEH < EyeHeight - EyeKneeDistance)
        {
            Movement = vec3(0.0f, HUMEPYMEH, 0.0f);
        }
    }

    if(Movement.y != 0.0f)
    {
        int Depth = 0;

        TestAgain:

        if(Depth < 16)
        {
            float Distance = ClosestDistance;
            vec3 Compensation;

            if(DistanceTest(EyePosition + Movement, EyeKneeDistance, ClosestDistance,
Distance, Compensation))
            {
                Movement += Compensation;

                Depth++;

                goto TestAgain;
            }
        }
    }
}

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

CScene::CScene()
{
    SetDefaults();
}

CScene::~CScene()
{
}

void CScene::SetDefaults()
{
    Vertices = NULL;
    VerticesCount = 0;
```

```
	VertexBufferObject = 0;
}

bool CScene::LoadBinary(const char *FileName)
{
	CString DirectoryFileName = ModuleDirectory + FileName;

	FILE *File;

	if(fopen_s(&File, DirectoryFileName, "rb") != 0)
	{
		ErrorLog.Append("Error opening file " + DirectoryFileName + "!\r\n");
		return false;
	}

	Destroy();

	if(fread(&VerticesCount, sizeof(int), 1, File) != 1)
	{
		ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
		fclose(File);
		return false;
	}

	if(VerticesCount > 0)
	{
		Vertices = new vec3[VerticesCount];

		if(fread(Vertices, sizeof(vec3), VerticesCount, File) != VerticesCount)
		{
			ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
			fclose(File);
			Destroy();
			return false;
		}

		vec3 *VertexBufferData = new vec3[VerticesCount * 3];

		for(int i = 0; i < VerticesCount; i += 3)
		{
			vec3 VertexA = Vertices[i + 0];
			vec3 VertexB = Vertices[i + 1];
			vec3 VertexC = Vertices[i + 2];
```

```
            vec3 Normal = normalize(cross(VertexB - VertexA, VertexC - VertexA));

            mat3x3 TBN = GetTBNMatrix(Normal);

            vec3 TexCoordA = TBN * VertexA;
            vec3 TexCoordB = TBN * VertexB;
            vec3 TexCoordC = TBN * VertexC;

            VertexBufferData[i * 3 + 0] = VertexA;
            VertexBufferData[i * 3 + 1] = Normal;
            VertexBufferData[i * 3 + 2] = TexCoordA;
            VertexBufferData[i * 3 + 3] = VertexB;
            VertexBufferData[i * 3 + 4] = Normal;
            VertexBufferData[i * 3 + 5] = TexCoordB;
            VertexBufferData[i * 3 + 6] = VertexC;
            VertexBufferData[i * 3 + 7] = Normal;
            VertexBufferData[i * 3 + 8] = TexCoordC;
        }

        glGenBuffers(1, &VertexBufferObject);

        glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
        glBufferData(GL_ARRAY_BUFFER, VerticesCount * 3 * 12, VertexBufferData,
GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        delete [] VertexBufferData;
    }

    fclose(File);

    if(!Texture.LoadTexture2D("concrete.jpg"))
    {
        Destroy();
        return false;
    }

    return true;
}

void CScene::Render()
{
    glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 36, (void*)0);

    glEnableClientState(GL_NORMAL_ARRAY);
    glNormalPointer(GL_FLOAT, 36, (void*)12);

    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    glTexCoordPointer(3, GL_FLOAT, 36, (void*)24);

    glBindTexture(GL_TEXTURE_2D, Texture);

    glDrawArrays(GL_TRIANGLES, 0, VerticesCount);

    glBindTexture(GL_TEXTURE_2D, 0);

    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

void CScene::Destroy()
{
    if(Vertices != NULL)
    {
        delete [] Vertices;
    }

    if(VertexBufferObject != 0)
    {
        glDeleteBuffers(1, &VertexBufferObject);
    }

    Texture.Destroy();

    SetDefaults();
}

vec3 *CScene::GetVertices()
{
    return Vertices;
}
```

```
int CScene::GetVerticesCount()
{
	return VerticesCount;
}

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

COpenGLRenderer::COpenGLRenderer()
{
	Texturing = true;
	Lighting = true;
	ApplySSAO = true;
	ApplyFXAA = true;
}

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("GLEW_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 |= !Preprocess.Load("preprocess.vs", "preprocess.fs");
	Error |= !SSAO.Load("ssao.vs", "ssao.fs");
	Error |= !SSAOFilterH.Load("ssaofilter.vs", "ssaofilterh.fs");
	Error |= !SSAOFilterV.Load("ssaofilter.vs", "ssaofilterv.fs");
	Error |= !DeferredLighting.Load("deferredlighting.vs", "deferredlighting.fs");
	Error |= !FXAA.Load("FXAA.vert", "FXAA_Extreme_Quality.frag");

	Error |= !Scene.LoadBinary("scene.bin");

	if(Error)
	{
		return false;
	}

	Preprocess.UniformLocations = new GLuint[1];
	Preprocess.UniformLocations[0] = glGetUniformLocation(Preprocess, "Texturing");

	SSAO.UniformLocations = new GLuint[2];
	SSAO.UniformLocations[0] = glGetUniformLocation(SSAO, "Scale");
	SSAO.UniformLocations[1] = glGetUniformLocation(SSAO,
"ProjectionBiasMatrixInverse");

	SSAOFilterH.UniformLocations = new GLuint[1];
	SSAOFilterH.UniformLocations[0] = glGetUniformLocation(SSAOFilterH, "PixelSizeX");

	SSAOFilterV.UniformLocations = new GLuint[1];
	SSAOFilterV.UniformLocations[0] = glGetUniformLocation(SSAOFilterV, "PixelSizeY");

	DeferredLighting.UniformLocations = new GLuint[3];
	DeferredLighting.UniformLocations[0] = glGetUniformLocation(DeferredLighting,
"ProjectionBiasMatrixInverse");
	DeferredLighting.UniformLocations[1] = glGetUniformLocation(DeferredLighting,
"Lighting");
	DeferredLighting.UniformLocations[2] = glGetUniformLocation(DeferredLighting,
"ApplySSAO");

	FXAA.UniformLocations = new GLuint[1];
	FXAA.UniformLocations[0] = glGetUniformLocation(FXAA, "RCPFrame");

	glUseProgram(SSAO);
	glUniform1i(glGetUniformLocation(SSAO, "NormalBuffer"), 0);
	glUniform1i(glGetUniformLocation(SSAO, "DepthBuffer"), 1);
	glUniform1i(glGetUniformLocation(SSAO, "RotationTexture"), 2);
	glUniform1f(glGetUniformLocation(SSAO, "Radius"), 0.125f);
	glUniform1f(glGetUniformLocation(SSAO, "Strength"), 2.0f);
```

```
glUniform1f(glGetUniformLocation(SSAO, "ConstantAttenuation"), 1.0f);
glUniform1f(glGetUniformLocation(SSAO, "LinearAttenuation"), 1.0f);
glUniform1f(glGetUniformLocation(SSAO, "QuadraticAttenuation"), 0.0f);
glUseProgram(0);

float s = 128.0f, e = 131070.0f, fs = 1.0f / s, fe = 1.0f / e, fd = fs - fe;

glUseProgram(SSAOFilterH);
glUniform1i(glGetUniformLocation(SSAOFilterH, "SSAOBuffer"), 0);
glUniform1i(glGetUniformLocation(SSAOFilterH, "DepthBuffer"), 1);
glUniform1f(glGetUniformLocation(SSAOFilterH, "fs"), fs);
glUniform1f(glGetUniformLocation(SSAOFilterH, "fd"), fd);
glUseProgram(0);

glUseProgram(SSAOFilterV);
glUniform1i(glGetUniformLocation(SSAOFilterV, "SSAOBuffer"), 0);
glUniform1i(glGetUniformLocation(SSAOFilterV, "DepthBuffer"), 1);
glUniform1f(glGetUniformLocation(SSAOFilterV, "fs"), fs);
glUniform1f(glGetUniformLocation(SSAOFilterV, "fd"), fd);
glUseProgram(0);

glUseProgram(DeferredLighting);
glUniform1i(glGetUniformLocation(DeferredLighting, "ColorBuffer"), 0);
glUniform1i(glGetUniformLocation(DeferredLighting, "NormalBuffer"), 1);
glUniform1i(glGetUniformLocation(DeferredLighting, "DepthBuffer"), 2);
glUniform1i(glGetUniformLocation(DeferredLighting, "SSAOBuffer"), 3);
glUseProgram(0);

srand(GetTickCount());

vec2 *Samples = new vec2[16];
float Angle = (float)M_PI_4;

for(int i = 0; i < 16; i++)
{
	Samples[i].x = cos(Angle) * (float)(i + 1) / 16.0f;
	Samples[i].y = sin(Angle) * (float)(i + 1) / 16.0f;

	Angle += (float)M_PI_2;

	if(((i + 1) % 4) == 0) Angle += (float)M_PI_4;
}

glUseProgram(SSAO);
```

```
	glUniform2fv(glGetUniformLocation(SSAO, "Samples"), 16, (float*)Samples);
	glUseProgram(0);

	delete [] Samples;

	vec4 *RotationTextureData = new vec4[64 * 64];
	float RandomAngle = (float)rand() / (float)RAND_MAX * (float)M_PI * 2.0f;

	for(int i = 0; i < 64 * 64; i++)
	{
		RotationTextureData[i].x = cos(RandomAngle) * 0.5f + 0.5f;
		RotationTextureData[i].y = sin(RandomAngle) * 0.5f + 0.5f;
		RotationTextureData[i].z = -sin(RandomAngle) * 0.5f + 0.5f;
		RotationTextureData[i].w = cos(RandomAngle) * 0.5f + 0.5f;

		RandomAngle += (float)rand() / (float)RAND_MAX * (float)M_PI * 2.0f;
	}

	glGenTextures(1, &RotationTexture);
	glBindTexture(GL_TEXTURE_2D, RotationTexture);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 64, 64, 0, GL_RGBA, GL_FLOAT,
(float*)RotationTextureData);
	glBindTexture(GL_TEXTURE_2D, 0);

	delete [] RotationTextureData;

	glGenTextures(1, &ColorBuffer);
	glGenTextures(1, &NormalBuffer);
	glGenTextures(1, &DepthBuffer);
	glGenTextures(1, &SSAOBuffer);
	glGenTextures(1, &SSAOFilterBuffer);
	glGenTextures(1, &FXAABuffer);

	glGenFramebuffersEXT(1, &FBO);

	Camera.Look(vec3(0.0f, 1.75f, 7.0f), vec3(0.0f, 1.75f, 0.0f));

	CollisionDetector.Init(Scene.GetVertices(), Scene.GetVerticesCount(), 1.75f, 1.25f,
0.125f);

	return true;
}
```

```
void COpenGLRenderer::Render()
{
    GLenum Buffers[] = {GL_COLOR_ATTACHMENT0_EXT,
GL_COLOR_ATTACHMENT1_EXT};

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
    glDrawBuffers(2, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
GL_TEXTURE_2D, ColorBuffer, 0);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,
GL_TEXTURE_2D, NormalBuffer, 0);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
GL_TEXTURE_2D, DepthBuffer, 0);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);

    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(&Camera.ViewMatrix);

    glUseProgram(Preprocess);
    glUniform1i(Preprocess.UniformLocations[0], Texturing);

    Scene.Render();

    glUseProgram(0);

    glDisable(GL_CULL_FACE);
    glDisable(GL_DEPTH_TEST);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

    if(ApplySSAO)
    {
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
        glDrawBuffers(1, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, SSAOBuffer, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, 0, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_DEPTH_ATTACHMENT_EXT, GL_TEXTURE_2D, 0, 0);

        glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, NormalBuffer);
        glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, DepthBuffer);
        glActiveTexture(GL_TEXTURE2); glBindTexture(GL_TEXTURE_2D, RotationTexture);
        glUseProgram(SSAO);
        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);

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
        glDrawBuffers(1, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, SSAOFilterBuffer, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, 0, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_DEPTH_ATTACHMENT_EXT, GL_TEXTURE_2D, 0, 0);

        glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, SSAOBuffer);
        glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, DepthBuffer);
        glUseProgram(SSAOFilterH);
        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);

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
        glDrawBuffers(1, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, SSAOBuffer, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, 0, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_DEPTH_ATTACHMENT_EXT, GL_TEXTURE_2D, 0, 0);

        glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, SSAOFilterBuffer);
        glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, DepthBuffer);
        glUseProgram(SSAOFilterV);
        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);

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
    }

    if(ApplyFXAA)
    {
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
        glDrawBuffers(1, Buffers); glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, FXAABuffer, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, 0, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_DEPTH_ATTACHMENT_EXT, GL_TEXTURE_2D, 0, 0);
    }

    glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, ColorBuffer);
    glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, NormalBuffer);
    glActiveTexture(GL_TEXTURE2); glBindTexture(GL_TEXTURE_2D, DepthBuffer);
    glActiveTexture(GL_TEXTURE3); glBindTexture(GL_TEXTURE_2D, SSAOBuffer);
    glUseProgram(DeferredLighting);
    glUniform1i(DeferredLighting.UniformLocations[1], Lighting);
    glUniform1i(DeferredLighting.UniformLocations[2], ApplySSAO);
    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);

    if(ApplyFXAA)
    {
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
    }

    if(ApplyFXAA)
    {
        glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, FXAABuffer);
        glUseProgram(FXAA);
        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_TEXTURE0); glBindTexture(GL_TEXTURE_2D, 0);
    }
}

void COpenGLRenderer::Animate(float FrameTime)
{
}

void COpenGLRenderer::Resize(int Width, int Height)
{
    this->Width = Width;
    this->Height = Height;

    glViewport(0, 0, Width, Height);

    Camera.SetPerspective(45.0f, (float)Width / (float)Height, 0.125f, 512.0f);

    glMatrixMode(GL_PROJECTION);
```

```
        glLoadMatrixf(&Camera.ProjectionMatrix);

        mat4x4 ProjectionBiasMatrixInverse = Camera.ProjectionMatrixInverse *
BiasMatrixInverse;

        glBindTexture(GL_TEXTURE_2D, ColorBuffer);
        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_RGBA8, Width, Height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
        glBindTexture(GL_TEXTURE_2D, 0);

        glBindTexture(GL_TEXTURE_2D, NormalBuffer);
        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_RGBA8, Width, Height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
        glBindTexture(GL_TEXTURE_2D, 0);

        glBindTexture(GL_TEXTURE_2D, DepthBuffer);
        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);

        glBindTexture(GL_TEXTURE_2D, SSAOBuffer);
        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_RGBA8, Width, Height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
        glBindTexture(GL_TEXTURE_2D, 0);

        glBindTexture(GL_TEXTURE_2D, SSAOFilterBuffer);
        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_RGBA8, Width, Height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
	glBindTexture(GL_TEXTURE_2D, 0);

	glBindTexture(GL_TEXTURE_2D, FXAABuffer);
	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);

	glUseProgram(SSAO);
	glUniform2f(SSAO.UniformLocations[0], (float)Width / 64.0f, (float)Height / 64.0f);
	glUniformMatrix4fv(SSAO.UniformLocations[1], 1, GL_FALSE,
&ProjectionBiasMatrixInverse);
	glUseProgram(0);

	glUseProgram(SSAOFilterH);
	glUniform1f(SSAOFilterH.UniformLocations[0], 1.0f / (float)Width);
	glUseProgram(SSAOFilterV);
	glUniform1f(SSAOFilterV.UniformLocations[0], 1.0f / (float)Height);
	glUseProgram(0);

	glUseProgram(DeferredLighting);
	glUniformMatrix4fv(DeferredLighting.UniformLocations[0], 1, GL_FALSE,
&ProjectionBiasMatrixInverse);
	glUseProgram(0);

	glUseProgram(FXAA);
	glUniform2f(FXAA.UniformLocations[0], 1.0f / (float)Width, 1.0f / (float)Height);
	glUseProgram(0);
}

void COpenGLRenderer::Destroy()
{
	Preprocess.Destroy();
	SSAO.Destroy();
	SSAOFilterH.Destroy();
	SSAOFilterV.Destroy();
	DeferredLighting.Destroy();
	FXAA.Destroy();

	Scene.Destroy();

	glDeleteTextures(1, &RotationTexture);

	glDeleteTextures(1, &ColorBuffer);
	glDeleteTextures(1, &NormalBuffer);
	glDeleteTextures(1, &DepthBuffer);
	glDeleteTextures(1, &SSAOBuffer);
	glDeleteTextures(1, &SSAOFilterBuffer);
	glDeleteTextures(1, &FXAABuffer);

	if(GLEW_EXT_framebuffer_object)
	{
		glDeleteFramebuffersEXT(1, &FBO);
	}

	CollisionDetector.Destroy();
}

void COpenGLRenderer::CheckCameraKeys(float FrameTime)
{
	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)
	{
		vec3 Movement = Camera.OnKeys(Keys, FrameTime * 0.5f);

		CollisionDetector.CheckHorizontalCollision(Camera.Reference, Movement);

		if(length(Movement) > 0.0f)
		{
			Camera.Move(Movement);
		}
	}

	vec3 Movement;

	CollisionDetector.CheckVerticalCollision(Camera.Reference, FrameTime, Movement);

	if(length(Movement) > 0.0f)
	{
		Camera.Move(Movement);
	}
}

void COpenGLRenderer::OnKeyDown(UINT Key)
{
	switch(Key)
	{
		case VK_F1:
			Texturing = !Texturing;
			break;

		case VK_F2:
			Lighting = !Lighting;
			break;

		case VK_F3:
			ApplySSAO = !ApplySSAO;
			break;

		case VK_F4:
			ApplyFXAA = !ApplyFXAA;
			break;

		case 'C':
			CollisionDetector.Crouch();
			break;

		case VK_SPACE:
			CollisionDetector.Jump();
			break;
	}
}

void COpenGLRenderer::OnLButtonDown(int X, int Y)
```

```
{
    LastClickedX = X;
    LastClickedY = Y;
}

void COpenGLRenderer::OnLButtonUp(int X, int Y)
{
    if(X == LastClickedX && Y == LastClickedY)
    {
    }
}

void COpenGLRenderer::OnMouseMove(int X, int Y)
{
    if(GetKeyState(VK_RBUTTON) & 0x80)
    {
        Camera.OnMouseMove(LastX - X, LastY - Y);
    }

    LastX = X;
    LastY = Y;
}

void COpenGLRenderer::OnMouseWheel(short zDelta)
{
    // Camera.OnMouseWheel(zDelta);
}

void COpenGLRenderer::OnRButtonDown(int X, int Y)
{
    LastClickedX = X;
    LastClickedY = Y;
}

void COpenGLRenderer::OnRButtonUp(int X, int Y)
{
    if(X == LastClickedX && Y == LastClickedY)
    {
    }
}
```

第四章地形的渲染

第一节 Shader Source

```
<1>glsl120shader.vs
#version 120
uniform vec3 CameraPosition;
varying vec3 var_Normal, var_LightDirection;
void main()
{
  gl_FrontColor = gl_Color;
  var_Normal = gl_Normal;
  var_LightDirection = CameraPosition - gl_Vertex.xyz;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
glsl120shader.fs
#version 120
varying vec3 var_Normal, var_LightDirection;
void main()
{
  gl_FragColor = gl_Color;
  float NdotLD = dot(normalize(var_Normal), normalize(var_LightDirection));
  gl_FragColor.rgb *= 0.5 + 0.5 * NdotLD;
}
```

第二节 Source Code Header

```
class CVertex
{
public:
  vec3 Position;
  vec3 Normal;
};

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

class CTerrain
{
private:
  int Size, SizeP1;
  float SizeD2;

private:
  vec3 Min, Max;
```

```
private:
  float *Heights;

private:
  int VerticesCount, IndicesCount;

private:
  GLuint VertexBufferObject, IndexBufferObject;

public:
  CTerrain();
  ~CTerrain();

private:
  void SetDefaults();

public:
  bool LoadTexture2D(char *FileName, float Scale = 256.0f, float Offset = -128.0f);
  bool LoadBinary(char *FileName);
  bool SaveBinary(char *FileName);
  void Render();
  void Destroy();

public:
  vec3 GetMin();
  vec3 GetMax();

private:
  int GetIndex(int X, int Z);
  float GetHeight(int X, int Z);

public:
  float GetHeight(float X, float Z);

private:
  float GetHeight(float *Heights, int Size, float X, float Z);
};

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

class COpenGLRenderer
{
```

```
private:
  int LastX, LastY, LastClickedX, LastClickedY;

private:
  int Width, Height;

private:
  CCamera Camera;

private:
  CShaderProgram Shader;

private:
  CTerrain Terrain;

private:
  bool Wireframe;

public:
  CString Text;

public:
  COpenGLRenderer();
  ~COpenGLRenderer();

public:
  bool Init();
  void Render();
  void Animate(float FrameTime);
  void Resize(int Width, int Height);
  void Destroy();

private:
  void CheckCameraTerrainPosition(vec3 &Movement);

public:
  void CheckCameraKeys(float FrameTime);

public:
  void OnKeyDown(UINT Key);
  void OnLButtonDown(int X, int Y);
  void OnLButtonUp(int X, int Y);
  void OnMouseMove(int X, int Y);
  void OnMouseWheel(short zDelta);
```

```
    void OnRButtonDown(int X, int Y);
    void OnRButtonUp(int X, int Y);
  };
```

第三节 Source Code Cpp

```
CTerrain::CTerrain()
{
    SetDefaults();
}

CTerrain::~CTerrain()
{
}

void CTerrain::SetDefaults()
{
    Size = 0;
    SizeP1 = 0;
    SizeD2 = 0.0f;

    Min = Max = vec3(0.0f);

    Heights = NULL;

    VerticesCount = 0;
    IndicesCount = 0;

    VertexBufferObject = 0;
    IndexBufferObject = 0;
}

bool CTerrain::LoadTexture2D(char *FileName, float Scale, float Offset)
{
    CTexture Texture;

    if(!Texture.LoadTexture2D(FileName))
    {
        return false;
    }

    if(Texture.GetWidth() != Texture.GetHeight())
    {
        ErrorLog.Append("Unsupported texture dimensions (%s)!\r\n", FileName);
        Texture.Destroy();
```

```
	return false;
}

Destroy();

Size = Texture.GetWidth();
SizeP1 = Size + 1;
SizeD2 = (float)Size / 2.0f;

VerticesCount = SizeP1 * SizeP1;

float *TextureHeights = new float[Size * Size];

glBindTexture(GL_TEXTURE_2D, Texture);
glGetTexImage(GL_TEXTURE_2D, 0, GL_GREEN, GL_FLOAT, TextureHeights);
glBindTexture(GL_TEXTURE_2D, 0);

Texture.Destroy();

for(int i = 0; i < Size * Size; i++)
{
	TextureHeights[i] = TextureHeights[i] * Scale + Offset;
}

Heights = new float[VerticesCount];

int i = 0;

for(int z = 0; z <= Size; z++)
{
	for(int x = 0; x <= Size; x++)
	{
		Heights[i++] = GetHeight(TextureHeights, Size, (float)x - 0.5f, (float)z - 0.5f);
	}
}

delete [] TextureHeights;

float *SmoothedHeights = new float[VerticesCount];

i = 0;

for(int z = 0; z <= Size; z++)
{
        for(int x = 0; x <= Size; x++)
        {
            SmoothedHeights[i] = 0.0f;

            SmoothedHeights[i] += GetHeight(x - 1, z + 1) + GetHeight(x, z + 1) * 2 + GetHeight(x + 1, z + 1);
            SmoothedHeights[i] += GetHeight(x - 1, z) * 2 + GetHeight(x, z) * 3 + GetHeight(x + 1, z) * 2;
            SmoothedHeights[i] += GetHeight(x - 1, z - 1) + GetHeight(x, z - 1) * 2 + GetHeight(x + 1, z - 1);

            SmoothedHeights[i] /= 15.0f;

            i++;
        }
    }

    delete [] Heights;

    Heights = SmoothedHeights;

    Min.x = Min.z = -SizeD2;
    Max.x = Max.z = SizeD2;

    Min.y = Max.y = Heights[0];

    for(int i = 1; i < VerticesCount; i++)
    {
        if(Heights[i] < Min.y) Min.y = Heights[i];
        if(Heights[i] > Max.y) Max.y = Heights[i];
    }

    CVertex *Vertices = new CVertex[VerticesCount];

    i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            Vertices[i].Position = vec3((float)x - SizeD2, Heights[i], SizeD2 - (float)z);
            Vertices[i].Normal = normalize(vec3(GetHeight(x - 1, z) - GetHeight(x + 1, z), 2.0f, GetHeight(x, z + 1) - GetHeight(x, z - 1)));
            i++;
        }
    }

    glGenBuffers(1, &VertexBufferObject);

    glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
    glBufferData(GL_ARRAY_BUFFER, VerticesCount * sizeof(CVertex), Vertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    delete [] Vertices;

    IndicesCount = Size * Size * 2 * 3;

    int *Indices = new int[IndicesCount];

    i = 0;

    for(int z = 0; z < Size; z++)
    {
        for(int x = 0; x < Size; x++)
        {
            Indices[i++] = GetIndex(x, z);
            Indices[i++] = GetIndex(x + 1, z);
            Indices[i++] = GetIndex(x + 1, z + 1);

            Indices[i++] = GetIndex(x + 1, z + 1);
            Indices[i++] = GetIndex(x, z + 1);
            Indices[i++] = GetIndex(x, z);
        }
    }

    glGenBuffers(1, &IndexBufferObject);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndicesCount * sizeof(int), Indices, GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

    delete [] Indices;

    return true;
}
```

```
bool CTerrain::LoadBinary(char *FileName)
{
    CString DirectoryFileName = ModuleDirectory + FileName;

    FILE *File;

    if(fopen_s(&File, DirectoryFileName, "rb") != 0)
    {
        ErrorLog.Append("Error opening file " + DirectoryFileName + "!\r\n");
        return false;
    }

    int Size;

    if(fread(&Size, sizeof(int), 1, File) != 1 || Size <= 0)
    {
        ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
        fclose(File);
        return false;
    }

    Destroy();

    this->Size = Size;
    SizeP1 = Size + 1;
    SizeD2 = (float)Size / 2.0f;

    VerticesCount = SizeP1 * SizeP1;

    Heights = new float[VerticesCount];

    if(fread(Heights, sizeof(float), VerticesCount, File) != VerticesCount)
    {
        ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
        fclose(File);
        Destroy();
        return false;
    }

    fclose(File);

    Min.x = Min.z = -SizeD2;
    Max.x = Max.z = SizeD2;
```

```
	Min.y = Max.y = Heights[0];

	for(int i = 1; i < VerticesCount; i++)
	{
		if(Heights[i] < Min.y) Min.y = Heights[i];
		if(Heights[i] > Max.y) Max.y = Heights[i];
	}

	CVertex *Vertices = new CVertex[VerticesCount];

	int i = 0;

	for(int z = 0; z <= Size; z++)
	{
		for(int x = 0; x <= Size; x++)
		{
			Vertices[i].Position = vec3((float)x - SizeD2, Heights[i], SizeD2 - (float)z);
			Vertices[i].Normal = normalize(vec3(GetHeight(x - 1, z) - GetHeight(x + 1, z),
2.0f, GetHeight(x, z + 1) - GetHeight(x, z - 1)));

			i++;
		}
	}

	glGenBuffers(1, &VertexBufferObject);

	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
	glBufferData(GL_ARRAY_BUFFER, VerticesCount * sizeof(CVertex), Vertices,
GL_STATIC_DRAW);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	delete [] Vertices;

	IndicesCount = Size * Size * 2 * 3;

	int *Indices = new int[IndicesCount];

	i = 0;

	for(int z = 0; z < Size; z++)
	{
		for(int x = 0; x < Size; x++)
		{
```

```
            Indices[i++] = GetIndex(x, z);
            Indices[i++] = GetIndex(x + 1, z);
            Indices[i++] = GetIndex(x + 1, z + 1);

            Indices[i++] = GetIndex(x + 1, z + 1);
            Indices[i++] = GetIndex(x, z + 1);
            Indices[i++] = GetIndex(x, z);
        }
    }

    glGenBuffers(1, &IndexBufferObject);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndicesCount * sizeof(int), Indices,
GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

    delete [] Indices;

    return true;
}

bool CTerrain::SaveBinary(char *FileName)
{
    CString DirectoryFileName = ModuleDirectory + FileName;

    FILE *File;

    if(fopen_s(&File, DirectoryFileName, "wb+") != 0)
    {
        return false;
    }

    fwrite(&Size, sizeof(int), 1, File);

    fwrite(Heights, sizeof(float), VerticesCount, File);

    fclose(File);

    return true;
}

void CTerrain::Render()
{
	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);

	glEnableClientState(GL_VERTEX_ARRAY);
	glVertexPointer(3, GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 0));

	glEnableClientState(GL_NORMAL_ARRAY);
	glNormalPointer(GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 1));

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);

	glDrawElements(GL_TRIANGLES, IndicesCount, GL_UNSIGNED_INT, NULL);

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

	glDisableClientState(GL_NORMAL_ARRAY);
	glDisableClientState(GL_VERTEX_ARRAY);

	glBindBuffer(GL_ARRAY_BUFFER, 0);
}

void CTerrain::Destroy()
{
	if(Heights != NULL)
	{
		delete [] Heights;
	}

	if(VertexBufferObject != 0)
	{
		glDeleteBuffers(1, &VertexBufferObject);
	}

	if(IndexBufferObject != 0)
	{
		glDeleteBuffers(1, &IndexBufferObject);
	}

	SetDefaults();
}

vec3 CTerrain::GetMin()
{
	return Min;
}
```

```
vec3 CTerrain::GetMax()
{
	return Max;
}

int CTerrain::GetIndex(int X, int Z)
{
	return SizeP1 * Z + X;
}

float CTerrain::GetHeight(int X, int Z)
{
	return Heights[GetIndex(X < 0 ? 0 : X > Size ? Size : X, Z < 0 ? 0 : Z > Size ? Size : Z)];
}

float CTerrain::GetHeight(float X, float Z)
{
	Z = -Z;

	X += SizeD2;
	Z += SizeD2;

	float Size = (float)this->Size;

	if(X < 0.0f) X = 0.0f;
	if(X > Size) X = Size;
	if(Z < 0.0f) Z = 0.0f;
	if(Z > Size) Z = Size;

	int ix = (int)X, ixp1 = ix + 1;
	int iz = (int)Z, izp1 = iz + 1;

	float fx = X - (float)ix;
	float fz = Z - (float)iz;

	float a = GetHeight(ix, iz);
	float b = GetHeight(ixp1, iz);
	float c = GetHeight(ix, izp1);
	float d = GetHeight(ixp1, izp1);

	float ab = a + (b - a) * fx;
	float cd = c + (d - c) * fx;
	return ab + (cd - ab) * fz;
}

float CTerrain::GetHeight(float *Heights, int Size, float X, float Z)
{
	float SizeM1F = (float)Size - 1.0f;

	if(X < 0.0f) X = 0.0f;
	if(X > SizeM1F) X = SizeM1F;
	if(Z < 0.0f) Z = 0.0f;
	if(Z > SizeM1F) Z = SizeM1F;

	int ix = (int)X, ixp1 = ix + 1;
	int iz = (int)Z, izp1 = iz + 1;

	int SizeM1 = Size - 1;

	if(ixp1 > SizeM1) ixp1 = SizeM1;
	if(izp1 > SizeM1) izp1 = SizeM1;

	float fx = X - (float)ix;
	float fz = Z - (float)iz;

	int izMSize = iz * Size, izp1MSize = izp1 * Size;

	float a = Heights[izMSize + ix];
	float b = Heights[izMSize + ixp1];
	float c = Heights[izp1MSize + ix];
	float d = Heights[izp1MSize + ixp1];

	float ab = a + (b - a) * fx;
	float cd = c + (d - c) * fx;

	return ab + (cd - ab) * fz;
}

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

COpenGLRenderer::COpenGLRenderer()
{
	Wireframe = false;
}
```

```
COpenGLRenderer::~COpenGLRenderer()
{
}

bool COpenGLRenderer::Init()
{
	bool Error = false;

	Error |= !Shader.Load("glsl120shader.vs", "glsl120shader.fs");

	Error |= !Terrain.LoadBinary("terrain1.bin");

	if(Error)
	{
		return false;
	}

	Shader.UniformLocations = new GLuint[1];
	Shader.UniformLocations[0] = glGetUniformLocation(Shader, "CameraPosition");

	float Height = Terrain.GetHeight(0.0f, 0.0f);

	Camera.Look(vec3(0.0f, Height + 1.75f, 0.0f), vec3(0.0f, Height + 1.75f, -1.0f));

	return true;
}

void COpenGLRenderer::Render()
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	glEnable(GL_DEPTH_TEST);
	glEnable(GL_CULL_FACE);

	glMatrixMode(GL_MODELVIEW);
	glLoadMatrixf(&Camera.ViewMatrix);

	if(Wireframe)
	{
		glColor3f(0.0f, 0.0f, 0.0f);

		glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

		Terrain.Render();
		glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
	}

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

	glUseProgram(Shader);
	glUniform3fv(Shader.UniformLocations[0], 1, &Camera.Position);

	Terrain.Render();

	glUseProgram(0);

	glDisable(GL_CULL_FACE);
	glDisable(GL_DEPTH_TEST);
}

void COpenGLRenderer::Animate(float FrameTime)
{
}

void COpenGLRenderer::Resize(int Width, int Height)
{
	this->Width = Width;
	this->Height = Height;

	glViewport(0, 0, Width, Height);

	Camera.SetPerspective(45.0f, (float)Width / (float)Height, 0.125f, 1024.0f);

	glMatrixMode(GL_PROJECTION);
	glLoadMatrixf(&Camera.ProjectionMatrix);
}

void COpenGLRenderer::Destroy()
{
	Shader.Destroy();

	Terrain.Destroy();
}

void COpenGLRenderer::CheckCameraTerrainPosition(vec3 &Movement)
{
    vec3 CameraPosition = Camera.Reference + Movement, Min = Terrain.GetMin(), Max = Terrain.GetMax();

    if(CameraPosition.x < Min.x) Movement += vec3(Min.x - CameraPosition.x, 0.0f, 0.0f);
    if(CameraPosition.x > Max.x) Movement += vec3(Max.x - CameraPosition.x, 0.0f, 0.0f);
    if(CameraPosition.z < Min.z) Movement += vec3(0.0f, 0.0f, Min.z - CameraPosition.z);
    if(CameraPosition.z > Max.z) Movement += vec3(0.0f, 0.0f, Max.z - CameraPosition.z);

    CameraPosition = Camera.Reference + Movement;

    float Height = Terrain.GetHeight(CameraPosition.x, CameraPosition.z);

    Movement += vec3(0.0f, Height + 1.75f - Camera.Reference.y, 0.0f);
}

void COpenGLRenderer::CheckCameraKeys(float FrameTime)
{
    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)
    {
        vec3 Movement = Camera.OnKeys(Keys, FrameTime * 0.5f);

        CheckCameraTerrainPosition(Movement);

        Camera.Move(Movement);
    }
}

void COpenGLRenderer::OnKeyDown(UINT Key)
{
    switch(Key)
    {
        case VK_F1:
```

```
			Wireframe = !Wireframe;
			break;

		case VK_F5:
			Terrain.SaveBinary("terrain-saved.bin");
			break;

		case '1':
			if(Terrain.LoadBinary("terrain1.bin")) { vec3 Movement;
CheckCameraTerrainPosition(Movement); Camera.Move(Movement); }
			break;

		case '2':
			if(Terrain.LoadTexture2D("terrain2.jpg", 32.0f, -16.0f)) { vec3 Movement;
CheckCameraTerrainPosition(Movement); Camera.Move(Movement); }
			break;

		case '3':
			if(Terrain.LoadTexture2D("terrain3.jpg", 128.0f, -64.0f)) { vec3 Movement;
CheckCameraTerrainPosition(Movement); Camera.Move(Movement); }
			break;

		case '4':
			if(Terrain.LoadTexture2D("terrain4.jpg", 128.0f, -64.0f)) { vec3 Movement;
CheckCameraTerrainPosition(Movement); Camera.Move(Movement); }
			break;
	}
}

void COpenGLRenderer::OnLButtonDown(int X, int Y)
{
	LastClickedX = X;
	LastClickedY = Y;
}

void COpenGLRenderer::OnLButtonUp(int X, int Y)
{
	if(X == LastClickedX && Y == LastClickedY)
	{
	}
}

void COpenGLRenderer::OnMouseMove(int X, int Y)
{
    if(GetKeyState(VK_RBUTTON) & 0x80)
    {
        Camera.OnMouseMove(LastX - X, LastY - Y);
    }

    LastX = X;
    LastY = Y;
}

void COpenGLRenderer::OnMouseWheel(short zDelta)
{
    Camera.OnMouseWheel(zDelta);
}

void COpenGLRenderer::OnRButtonDown(int X, int Y)
{
    LastClickedX = X;
    LastClickedY = Y;
}

void COpenGLRenderer::OnRButtonUp(int X, int Y)
{
    if(X == LastClickedX && Y == LastClickedY)
    {
    }
}
```

第五章 二叉场景分割树

第一节 Shader Source

```
<1>glsl120shader.vs
#version 120
uniform vec3 CameraPosition;
varying vec3 var_Normal, var_LightDirection;
void main()
{
	gl_FrontColor = gl_Color;
	var_Normal = gl_Normal;
	var_LightDirection = CameraPosition - gl_Vertex.xyz;
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
glsl120shader.fs
#version 120
varying vec3 var_Normal, var_LightDirection;
void main()
{
	gl_FragColor = gl_Color;
	float NdotLD = dot(normalize(var_Normal), normalize(var_LightDirection));
	gl_FragColor.rgb *= 0.5 + 0.5 * NdotLD;
}
```

第二节 Source Code Header

```
class CPlane
{
private:
	vec3 N;
	float ND;
	int O;

public:
	CPlane();
	~CPlane();

public:
	void Set(const vec3 &A, const vec3 &B, const vec3 &C);
	bool AABBBehind(const vec3 *AABBVertices);
	float AABBDistance(const vec3 *AABBVertices);
};
```

```
// ----------------------------------------------------------------------------
----------------------------------------------

class CFrustum
{
private:
	vec3 Vertices[8];

private:
	CPlane Planes[6];

public:
	CFrustum();
	~CFrustum();

public:
	void Set(const mat4x4 &ViewProjectionMatrixInverse);
	bool AABBVisible(const vec3 *AABBVertices);
	float AABBDistance(const vec3 *AABBVertices);
	void Render();
};

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

class CCamera
{
public:
	vec3 X, Y, Z, Position, Reference;

public:
	mat4x4 ViewMatrix, ViewMatrixInverse, ProjectionMatrix, ProjectionMatrixInverse,
ViewProjectionMatrix, ViewProjectionMatrixInverse;

public:
	CFrustum Frustum;

public:
	CCamera();
	~CCamera();

public:
	void Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference
= false);
```

```
	void Move(const vec3 &Movement);
	vec3 OnKeys(BYTE Keys, float FrameTime);
	void OnMouseMove(int dx, int dy);
	void OnMouseWheel(float zDelta);
	void SetPerspective(float fovy, float aspect, float n, float f);

private:
	void CalculateViewMatrix();
};

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

class CVertex
{
public:
	vec3 Position;
	vec3 Normal;
};

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

class CAABB
{
private:
	vec3 Vertices[8];

public:
	CAABB();
	~CAABB();

public:
	void Set(const vec3 &Min, const vec3 &Max);
	bool PointInside(const vec3 &Point);
	bool Visible(CFrustum &Frustum);
	float Distance(CFrustum &Frustum);
	void Render();
};

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

class CBSPTreeNode
```

```
{
private:
	vec3 Min, Max;

private:
	int Depth;

private:
	CAABB AABB;

private:
	bool Visible;
	float Distance;

private:
	int *Indices;

private:
	int IndicesCount;

private:
	GLuint IndexBufferObject;

private:
	CBSPTreeNode *Children[2];

public:
	CBSPTreeNode();
	~CBSPTreeNode();

private:
	void SetDefaults();

public:
	void InitAABB(const vec3 &Min, const vec3 &Max, int Depth, float MinAABBSize);
	bool CheckTriangle(CVertex *Vertices, int *Indices, int A, int B, int C);
	void AllocateMemory();
	bool AddTriangle(CVertex *Vertices, int *Indices, int A, int B, int C);
	void ResetAABB(CVertex *Vertices);
	int InitIndexBufferObject();
	int CheckVisibility(CFrustum &Frustum, CBSPTreeNode **VisibleGeometryNodes, int
&VisibleGeometryNodesCount);
	float GetDistance();
	void Render();
	void RenderAABB(int Depth);
	void Destroy();
};

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

class CBSPTree
{
private:
	CBSPTreeNode *Root;

private:
	CBSPTreeNode **VisibleGeometryNodes;
	int VisibleGeometryNodesCount;

public:
	CBSPTree();
	~CBSPTree();

private:
	void SetDefaults();

public:
	void Init(CVertex *Vertices, int *Indices, int IndicesCount, const vec3 &Min, const vec3
&Max, float MinAABBSize = 16.0f);
	void QuickSortVisibleGeometryNodes(int Left, int Right);
	int CheckVisibility(CFrustum &Frustum, bool SortVisibleGeometryNodes);
	void Render(bool VisualizeRenderingOrder);
	void RenderAABB(int Depth);
	void Destroy();
};

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

class CTerrain
{
private:
	int Size, SizeP1;
	float SizeD2;

private:
	vec3 Min, Max;
```

```
private:
	float *Heights;

private:
	int VerticesCount;

private:
	GLuint VertexBufferObject;

public:
	CBSPTree BSPTree;

public:
	CTerrain();
	~CTerrain();

private:
	void SetDefaults();

public:
	bool LoadTexture2D(char *FileName, float Scale = 256.0f, float Offset = -128.0f);
	bool LoadBinary(char *FileName);
	bool SaveBinary(char *FileName);
	int CheckVisibility(CFrustum &Frustum, bool SortVisibleGeometryNodes = true);
	void Render(bool VisualizeRenderingOrder = false);
	void RenderAABB(int Depth = -1);
	void Destroy();

public:
	vec3 GetMin();
	vec3 GetMax();

private:
	int GetIndex(int X, int Z);
	float GetHeight(int X, int Z);

public:
	float GetHeight(float X, float Z);

private:
	float GetHeight(float *Heights, int Size, float X, float Z);
};
```

```
// -------------------------------------------------------------------------------------------------------------------------------

class COpenGLRenderer
{
private:
	int LastX, LastY, LastClickedX, LastClickedY;

private:
	int Width, Height;

private:
	mat4x4 ViewMatrix, ProjectionMatrix;

private:
	CCamera Camera;

private:
	CShaderProgram Shader;

private:
	CTerrain Terrain;

private:
	bool Wireframe, RenderAABB, RenderTree2D, VisualizeRenderingOrder, SortVisibleGeometryNodes, VisibilityCheckingPerformanceTest;
	int Depth;

public:
	CString Text;

public:
	COpenGLRenderer();
	~COpenGLRenderer();

public:
	bool Init();
	void Render();
	void Animate(float FrameTime);
	void Resize(int Width, int Height);
	void Destroy();

private:
	void CalculateProjectionMatrix();
```

```
    void CheckCameraTerrainPosition(vec3 &Movement);

public:
    void CheckCameraKeys(float FrameTime);

public:
    void OnKeyDown(UINT Key);
    void OnLButtonDown(int X, int Y);
    void OnLButtonUp(int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseWheel(short zDelta);
    void OnRButtonDown(int X, int Y);
    void OnRButtonUp(int X, int Y);
};
```

第三节 Source Code Cpp

```
CPlane::CPlane()
{
}

CPlane::~CPlane()
{
}

void CPlane::Set(const vec3 &A, const vec3 &B, const vec3 &C)
{
    N = normalize(cross(B - A, C - A));
    ND = dot(N, A);
    //计算视锥体的面,面的代表 0-6 分别为 X,-X,Y,-Y,Z,-Z
    //O = N.z < 0.0f ? (N.y < 0.0f ? dot1 : dot2) : (N.y < 0.0f ?dot3 : dot4);
    O = N.z < 0.0f ? (N.y < 0.0f ? (N.x < 0.0f ? 0 : 1) : (N.x < 0.0f ? 2 : 3)) : (N.y < 0.0f ? (N.x
< 0.0f ? 4 : 5) : (N.x < 0.0f ? 6 : 7));
}
/*
   *计算是否位于平面之下
  */
bool CPlane::AABBBehind(const vec3 *AABBVertices)
{
    return dot(N, AABBVertices[O]) < ND;
}
/*
   *计算顶点鱼平面的距离,在实际的实践中,应考虑末尾的平面方程的常数项
   */
float CPlane::AABBDistance(const vec3 *AABBVertices)
```

```
{
	return dot(N, AABBVertices[O]);
}

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

CFrustum::CFrustum()
{
}

CFrustum::~CFrustum()
{
}
/*
   *使用视图投影矩阵的逆来计算视锥体,参与计算的各个顶点是标准 NDC 坐标下的 8 个顶点
   */
void CFrustum::Set(const mat4x4 &ViewProjectionMatrixInverse)
{
	vec4 A = ViewProjectionMatrixInverse * vec4(-1.0f, -1.0f,  1.0f, 1.0f);
	vec4 B = ViewProjectionMatrixInverse * vec4( 1.0f, -1.0f,  1.0f, 1.0f);
	vec4 C = ViewProjectionMatrixInverse * vec4(-1.0f,  1.0f,  1.0f, 1.0f);
	vec4 D = ViewProjectionMatrixInverse * vec4( 1.0f,  1.0f,  1.0f, 1.0f);
	vec4 E = ViewProjectionMatrixInverse * vec4(-1.0f, -1.0f, -1.0f, 1.0f);
	vec4 F = ViewProjectionMatrixInverse * vec4( 1.0f, -1.0f, -1.0f, 1.0f);
	vec4 G = ViewProjectionMatrixInverse * vec4(-1.0f,  1.0f, -1.0f, 1.0f);
	vec4 H = ViewProjectionMatrixInverse * vec4( 1.0f,  1.0f, -1.0f, 1.0f);

	Vertices[0] = vec3(A.x / A.w, A.y / A.w, A.z / A.w);
	Vertices[1] = vec3(B.x / B.w, B.y / B.w, B.z / B.w);
	Vertices[2] = vec3(C.x / C.w, C.y / C.w, C.z / C.w);
	Vertices[3] = vec3(D.x / D.w, D.y / D.w, D.z / D.w);
	Vertices[4] = vec3(E.x / E.w, E.y / E.w, E.z / E.w);
	Vertices[5] = vec3(F.x / F.w, F.y / F.w, F.z / F.w);
	Vertices[6] = vec3(G.x / G.w, G.y / G.w, G.z / G.w);
	Vertices[7] = vec3(H.x / H.w, H.y / H.w, H.z / H.w);

	Planes[0].Set(Vertices[4], Vertices[0], Vertices[2]);
	Planes[1].Set(Vertices[1], Vertices[5], Vertices[7]);
	Planes[2].Set(Vertices[4], Vertices[5], Vertices[1]);
	Planes[3].Set(Vertices[2], Vertices[3], Vertices[7]);
	Planes[4].Set(Vertices[0], Vertices[1], Vertices[3]);
	Planes[5].Set(Vertices[5], Vertices[4], Vertices[6]);
}

bool CFrustum::AABBVisible(const vec3 *AABBVertices)
{
	for(int i = 0; i < 6; i++)
	{
		if(Planes[i].AABBBehind(AABBVertices))
		{
			return false;
		}
	}

	return true;
}

float CFrustum::AABBDistance(const vec3 *AABBVertices)
{
	return Planes[5].AABBDistance(AABBVertices);
}

void CFrustum::Render()
{
	glBegin(GL_LINES);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[1]);
	glVertex3fv(&Vertices[2]); glVertex3fv(&Vertices[3]);
	glVertex3fv(&Vertices[4]); glVertex3fv(&Vertices[5]);
	glVertex3fv(&Vertices[6]); glVertex3fv(&Vertices[7]);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[2]);
	glVertex3fv(&Vertices[1]); glVertex3fv(&Vertices[3]);
	glVertex3fv(&Vertices[4]); glVertex3fv(&Vertices[6]);
	glVertex3fv(&Vertices[5]); glVertex3fv(&Vertices[7]);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[4]);
	glVertex3fv(&Vertices[1]); glVertex3fv(&Vertices[5]);
	glVertex3fv(&Vertices[2]); glVertex3fv(&Vertices[6]);
	glVertex3fv(&Vertices[3]); glVertex3fv(&Vertices[7]);

	glEnd();
}

// ----------------------------------------------------------------------------------------------------------------------
```

```
CCamera::CCamera()
{
	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);

	CalculateViewMatrix();
}

CCamera::~CCamera()
{
}

void CCamera::Look(const vec3 &Position, const vec3 &Reference, bool
RotateAroundReference)
{
	this->Position = Position;
	this->Reference = Reference;

	Z = normalize(Position - Reference);

	GetXY(Z, X, Y);

	if(!RotateAroundReference)
	{
		this->Reference = 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::SetPerspective(float fovy, float aspect, float n, float f)
{
	ProjectionMatrix = perspective(fovy, aspect, n, f);
	ProjectionMatrixInverse = inverse(ProjectionMatrix);
	ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;
	ViewProjectionMatrixInverse = ViewMatrixInverse * ProjectionMatrixInverse;
	Frustum.Set(ViewProjectionMatrixInverse);
}

void CCamera::CalculateViewMatrix()
{
	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);
	ViewMatrixInverse = inverse(ViewMatrix);
	ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;
	ViewProjectionMatrixInverse = ViewMatrixInverse * ProjectionMatrixInverse;

	Frustum.Set(ViewProjectionMatrixInverse);
}

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

CAABB::CAABB()
{
}

CAABB::~CAABB()
{
}

void CAABB::Set(const vec3 &Min, const vec3 &Max)
{
	Vertices[0] = vec3(Min.x, Min.y, Min.z);
	Vertices[1] = vec3(Max.x, Min.y, Min.z);
	Vertices[2] = vec3(Min.x, Max.y, Min.z);
	Vertices[3] = vec3(Max.x, Max.y, Min.z);
	Vertices[4] = vec3(Min.x, Min.y, Max.z);
	Vertices[5] = vec3(Max.x, Min.y, Max.z);
	Vertices[6] = vec3(Min.x, Max.y, Max.z);
	Vertices[7] = vec3(Max.x, Max.y, Max.z);
}

bool CAABB::PointInside(const vec3 &Point)
{
	if(Point.x < Vertices[0].x) return false;
	if(Point.y < Vertices[0].y) return false;
	if(Point.z < Vertices[0].z) return false;

	if(Point.x > Vertices[7].x) return false;
	if(Point.y > Vertices[7].y) return false;
	if(Point.z > Vertices[7].z) return false;

	return true;
}

bool CAABB::Visible(CFrustum &Frustum)
{
	return Frustum.AABBVisible(Vertices);
}

float CAABB::Distance(CFrustum &Frustum)
{
	return Frustum.AABBDistance(Vertices);
}

void CAABB::Render()
{
	glBegin(GL_LINES);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[1]);
	glVertex3fv(&Vertices[2]); glVertex3fv(&Vertices[3]);
	glVertex3fv(&Vertices[4]); glVertex3fv(&Vertices[5]);
	glVertex3fv(&Vertices[6]); glVertex3fv(&Vertices[7]);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[2]);
	glVertex3fv(&Vertices[1]); glVertex3fv(&Vertices[3]);
	glVertex3fv(&Vertices[4]); glVertex3fv(&Vertices[6]);
	glVertex3fv(&Vertices[5]); glVertex3fv(&Vertices[7]);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[4]);
	glVertex3fv(&Vertices[1]); glVertex3fv(&Vertices[5]);
	glVertex3fv(&Vertices[2]); glVertex3fv(&Vertices[6]);
	glVertex3fv(&Vertices[3]); glVertex3fv(&Vertices[7]);

	glEnd();
}

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

CBSPTreeNode::CBSPTreeNode()
{
	SetDefaults();
```

```
}

CBSPTreeNode::~CBSPTreeNode()
{
}

void CBSPTreeNode::SetDefaults()
{
    Min = Max = vec3(0.0f);

    Depth = 0;

    Indices = NULL;

    IndicesCount = 0;

    IndexBufferObject = 0;

    Children[0] = NULL;
    Children[1] = NULL;
}

void CBSPTreeNode::InitAABB(const vec3 &Min, const vec3 &Max, int Depth, float
MinAABBSize)
{
    this->Min = Min;
    this->Max = Max;

    this->Depth = Depth;

    vec3 Mid = (Min + Max) / 2.0f;
    vec3 Size = Max - Min;

    AABB.Set(Min, Max);

    if(Size.x > MinAABBSize || Size.z > MinAABBSize)
    {
        Children[0] = new CBSPTreeNode();
        Children[1] = new CBSPTreeNode();

        if(Size.x >= Size.z)
        {
            Children[0]->InitAABB(vec3(Min.x, Min.y, Min.z), vec3(Mid.x, Max.y, Max.z),
Depth + 1, MinAABBSize);
```

```
            Children[1]->InitAABB(vec3(Mid.x, Min.y, Min.z), vec3(Max.x, Max.y, Max.z),
Depth + 1, MinAABBSize);
        }
        else
        {
            Children[0]->InitAABB(vec3(Min.x, Min.y, Min.z), vec3(Max.x, Max.y, Mid.z),
Depth + 1, MinAABBSize);
            Children[1]->InitAABB(vec3(Min.x, Min.y, Mid.z), vec3(Max.x, Max.y, Max.z),
Depth + 1, MinAABBSize);
        }
    }
}

bool CBSPTreeNode::CheckTriangle(CVertex *Vertices, int *Indices, int A, int B, int C)
{
    if(AABB.PointInside(Vertices[Indices[A]].Position))
    {
        if(AABB.PointInside(Vertices[Indices[B]].Position))
        {
            if(AABB.PointInside(Vertices[Indices[C]].Position))
            {
                bool BelongsToAChild = false;

                if(Children[0] != NULL)
                {
                    BelongsToAChild |= Children[0]->CheckTriangle(Vertices, Indices, A,
B, C);
                }

                if(Children[1] != NULL && !BelongsToAChild)
                {
                    BelongsToAChild |= Children[1]->CheckTriangle(Vertices, Indices, A,
B, C);
                }

                if(!BelongsToAChild)
                {
                    IndicesCount += 3;
                }

                return true;
            }
        }
    }
    return false;
}

void CBSPTreeNode::AllocateMemory()
{
    if(IndicesCount > 0)
    {
        Indices = new int[IndicesCount];
        IndicesCount = 0;
    }

    if(Children[0] != NULL)
    {
        Children[0]->AllocateMemory();
    }

    if(Children[1] != NULL)
    {
        Children[1]->AllocateMemory();
    }
}

bool CBSPTreeNode::AddTriangle(CVertex *Vertices, int *Indices, int A, int B, int C)
{
    if(AABB.PointInside(Vertices[Indices[A]].Position))
    {
        if(AABB.PointInside(Vertices[Indices[B]].Position))
        {
            if(AABB.PointInside(Vertices[Indices[C]].Position))
            {
                bool BelongsToAChild = false;

                if(Children[0] != NULL)
                {
                    BelongsToAChild |= Children[0]->AddTriangle(Vertices, Indices, A, B,
C);
                }

                if(Children[1] != NULL && !BelongsToAChild)
                {
                    BelongsToAChild |= Children[1]->AddTriangle(Vertices, Indices, A, B,
C);
                }
                if(!BelongsToAChild)
                {
                    this->Indices[IndicesCount++] = Indices[A];
                    this->Indices[IndicesCount++] = Indices[B];
                    this->Indices[IndicesCount++] = Indices[C];
                }

                return true;
            }
        }
    }

    return false;
}

void CBSPTreeNode::ResetAABB(CVertex *Vertices)
{
    float MinY = Min.y, MaxY = Max.y;

    Min.y = MaxY;
    Max.y = MinY;

    if(IndicesCount > 0)
    {
        for(int i = 0; i < IndicesCount; i++)
        {
            if(Vertices[Indices[i]].Position.y < Min.y) Min.y = Vertices[Indices[i]].Position.y;
            if(Vertices[Indices[i]].Position.y > Max.y) Max.y = Vertices[Indices[i]].Position.y;
        }
    }

    if(Children[0] != NULL)
    {
        Children[0]->ResetAABB(Vertices);

        if(Children[0]->Min.y < Min.y) Min.y = Children[0]->Min.y;
        if(Children[0]->Max.y > Max.y) Max.y = Children[0]->Max.y;
    }

    if(Children[1] != NULL)
    {
        Children[1]->ResetAABB(Vertices);
        if(Children[1]->Min.y < Min.y) Min.y = Children[1]->Min.y;
        if(Children[1]->Max.y > Max.y) Max.y = Children[1]->Max.y;
    }

    AABB.Set(Min, Max);
}

int CBSPTreeNode::InitIndexBufferObject()
{
    int GeometryNodesCount = 0;

    if(IndicesCount > 0)
    {
        glGenBuffers(1, &IndexBufferObject);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndicesCount * sizeof(int), Indices,
GL_STATIC_DRAW);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

        delete [] Indices;
        Indices = NULL;

        GeometryNodesCount++;
    }

    if(Children[0] != NULL)
    {
        GeometryNodesCount += Children[0]->InitIndexBufferObject();
    }

    if(Children[1] != NULL)
    {
        GeometryNodesCount += Children[1]->InitIndexBufferObject();
    }

    return GeometryNodesCount;
}

int CBSPTreeNode::CheckVisibility(CFrustum &Frustum, CBSPTreeNode
**VisibleGeometryNodes, int &VisibleGeometryNodesCount)
{
    int TrianglesRendered = 0;
    Visible = AABB.Visible(Frustum);

    if(Visible)
    {
        if(IndicesCount > 0)
        {
            Distance = AABB.Distance(Frustum);

            VisibleGeometryNodes[VisibleGeometryNodesCount++] = this;

            TrianglesRendered += IndicesCount / 3;
        }

        if(Children[0] != NULL)
        {
            TrianglesRendered += Children[0]->CheckVisibility(Frustum, VisibleGeometryNodes, VisibleGeometryNodesCount);
        }

        if(Children[1] != NULL)
        {
            TrianglesRendered += Children[1]->CheckVisibility(Frustum, VisibleGeometryNodes, VisibleGeometryNodesCount);
        }
    }

    return TrianglesRendered;
}

float CBSPTreeNode::GetDistance()
{
    return Distance;
}

void CBSPTreeNode::Render()
{
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);

    glDrawElements(GL_TRIANGLES, IndicesCount, GL_UNSIGNED_INT, NULL);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}

void CBSPTreeNode::RenderAABB(int Depth)
```

```
{
	if(Visible)
	{
		if(Depth == -1 || Depth == this->Depth)
		{
			AABB.Render();
		}

		if(Children[0] != NULL)
		{
			Children[0]->RenderAABB(Depth);
		}

		if(Children[1] != NULL)
		{
			Children[1]->RenderAABB(Depth);
		}
	}
}

void CBSPTreeNode::Destroy()
{
	if(Indices != NULL)
	{
		delete [] Indices;
	}

	if(IndexBufferObject != 0)
	{
		glDeleteBuffers(1, &IndexBufferObject);
	}

	if(Children[0] != NULL)
	{
		Children[0]->Destroy();
		delete Children[0];
	}

	if(Children[1] != NULL)
	{
		Children[1]->Destroy();
		delete Children[1];
	}
```

```
    SetDefaults();
}

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

CBSPTree::CBSPTree()
{
    SetDefaults();
}

CBSPTree::~CBSPTree()
{
}

void CBSPTree::SetDefaults()
{
    Root = NULL;

    VisibleGeometryNodes = NULL;
}

void CBSPTree::Init(CVertex *Vertices, int *Indices, int IndicesCount, const vec3 &Min, const
vec3 &Max, float MinAABBSize)
{
    Destroy();

    if(Vertices != NULL && Indices != NULL && IndicesCount > 0)
    {
        Root = new CBSPTreeNode();

        Root->InitAABB(Min, Max, 0, MinAABBSize);

        for(int i = 0; i < IndicesCount; i += 3)
        {
            Root->CheckTriangle(Vertices, Indices, i, i + 1, i + 2);
        }

        Root->AllocateMemory();

        for(int i = 0; i < IndicesCount; i += 3)
        {
            Root->AddTriangle(Vertices, Indices, i, i + 1, i + 2);
        }
        Root->ResetAABB(Vertices);

        int GeometryNodesCount = Root->InitIndexBufferObject();

        VisibleGeometryNodes = new CBSPTreeNode*[GeometryNodesCount];
    }
}

void CBSPTree::QuickSortVisibleGeometryNodes(int Left, int Right)
{
    float Pivot = VisibleGeometryNodes[(Left + Right) / 2]->GetDistance();
    int i = Left, j = Right;

    while(i <= j)
    {
        while(VisibleGeometryNodes[i]->GetDistance() < Pivot) i++;
        while(VisibleGeometryNodes[j]->GetDistance() > Pivot) j--;

        if(i <= j)
        {
            if(i != j)
            {
                CBSPTreeNode *Temp = VisibleGeometryNodes[i];
                VisibleGeometryNodes[i] = VisibleGeometryNodes[j];
                VisibleGeometryNodes[j] = Temp;
            }

            i++;
            j--;
        }
    }

    if(Left < j)
    {
        QuickSortVisibleGeometryNodes(Left, j);
    }

    if(i < Right)
    {
        QuickSortVisibleGeometryNodes(i, Right);
    }
}
```

```
int CBSPTree::CheckVisibility(CFrustum &Frustum, bool SortVisibleGeometryNodes)
{
    int TrianglesRendered = 0;

    VisibleGeometryNodesCount = 0;

    if(Root != NULL)
    {
        TrianglesRendered = Root->CheckVisibility(Frustum, VisibleGeometryNodes,
VisibleGeometryNodesCount);

        if(SortVisibleGeometryNodes)
        {
            if(VisibleGeometryNodesCount > 1)
            {
                QuickSortVisibleGeometryNodes(0, VisibleGeometryNodesCount - 1);
            }
        }
    }

    return TrianglesRendered;
}

void CBSPTree::Render(bool VisualizeRenderingOrder)
{
    if(VisibleGeometryNodesCount > 0)
    {
        if(!VisualizeRenderingOrder)
        {
            for(int i = 0; i < VisibleGeometryNodesCount; i++)
            {
                VisibleGeometryNodes[i]->Render();
            }
        }
        else
        {
            for(int i = 0; i < VisibleGeometryNodesCount; i++)
            {
                float Color = (float)i / (float)VisibleGeometryNodesCount;

                glColor3f(Color, Color, Color);

                VisibleGeometryNodes[i]->Render();
            }
```

```
		}
	}
}

void CBSPTree::RenderAABB(int Depth)
{
	if(Root != NULL)
	{
		Root->RenderAABB(Depth);
	}
}

void CBSPTree::Destroy()
{
	if(Root != NULL)
	{
		Root->Destroy();
		delete Root;
	}

	if(VisibleGeometryNodes != NULL)
	{
		delete [] VisibleGeometryNodes;
	}

	SetDefaults();
}

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

CTerrain::CTerrain()
{
	SetDefaults();
}

CTerrain::~CTerrain()
{
}

void CTerrain::SetDefaults()
{
	Size = 0;
	SizeP1 = 0;
	SizeD2 = 0.0f;

	Min = Max = vec3(0.0f);

	Heights = NULL;

	VerticesCount = 0;

	VertexBufferObject = 0;
}

bool CTerrain::LoadTexture2D(char *FileName, float Scale, float Offset)
{
	CTexture Texture;

	if(!Texture.LoadTexture2D(FileName))
	{
		return false;
	}

	if(Texture.GetWidth() != Texture.GetHeight())
	{
		ErrorLog.Append("Unsupported texture dimensions (%s)!\r\n", FileName);
		Texture.Destroy();
		return false;
	}

	Destroy();

	Size = Texture.GetWidth();
	SizeP1 = Size + 1;
	SizeD2 = (float)Size / 2.0f;

	VerticesCount = SizeP1 * SizeP1;

	float *TextureHeights = new float[Size * Size];

	glBindTexture(GL_TEXTURE_2D, Texture);
	glGetTexImage(GL_TEXTURE_2D, 0, GL_GREEN, GL_FLOAT, TextureHeights);
	glBindTexture(GL_TEXTURE_2D, 0);

	Texture.Destroy();

	for(int i = 0; i < Size * Size; i++)
```

```
    {
        TextureHeights[i] = TextureHeights[i] * Scale + Offset;
    }

    Heights = new float[VerticesCount];

    int i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            Heights[i++] = GetHeight(TextureHeights, Size, (float)x - 0.5f, (float)z - 0.5f);
        }
    }

    delete [] TextureHeights;

    float *SmoothedHeights = new float[VerticesCount];

    i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            SmoothedHeights[i] = 0.0f;

            SmoothedHeights[i] += GetHeight(x - 1, z + 1) + GetHeight(x, z + 1) * 2 +
GetHeight(x + 1, z + 1);
            SmoothedHeights[i] += GetHeight(x - 1, z) * 2 + GetHeight(x, z) * 3 +
GetHeight(x + 1, z) * 2;
            SmoothedHeights[i] += GetHeight(x - 1, z - 1) + GetHeight(x, z - 1) * 2 +
GetHeight(x + 1, z - 1);

            SmoothedHeights[i] /= 15.0f;

            i++;
        }
    }

    delete [] Heights;

    Heights = SmoothedHeights;
    Min.x = Min.z = -SizeD2;
    Max.x = Max.z = SizeD2;

    Min.y = Max.y = Heights[0];

    for(int i = 1; i < VerticesCount; i++)
    {
        if(Heights[i] < Min.y) Min.y = Heights[i];
        if(Heights[i] > Max.y) Max.y = Heights[i];
    }

    CVertex *Vertices = new CVertex[VerticesCount];

    i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            Vertices[i].Position = vec3((float)x - SizeD2, Heights[i], SizeD2 - (float)z);
            Vertices[i].Normal = normalize(vec3(GetHeight(x - 1, z) - GetHeight(x + 1, z),
2.0f, GetHeight(x, z + 1) - GetHeight(x, z - 1)));

            i++;
        }
    }

    glGenBuffers(1, &VertexBufferObject);

    glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
    glBufferData(GL_ARRAY_BUFFER, VerticesCount * sizeof(CVertex), Vertices,
GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    int IndicesCount = Size * Size * 2 * 3;

    int *Indices = new int[IndicesCount];

    i = 0;

    for(int z = 0; z < Size; z++)
    {
        for(int x = 0; x < Size; x++)
        {
            Indices[i++] = GetIndex(x, z);
            Indices[i++] = GetIndex(x + 1, z);
            Indices[i++] = GetIndex(x + 1, z + 1);

            Indices[i++] = GetIndex(x + 1, z + 1);
            Indices[i++] = GetIndex(x, z + 1);
            Indices[i++] = GetIndex(x, z);
        }
    }

    BSPTree.Init(Vertices, Indices, IndicesCount, Min, Max);

    delete [] Vertices;
    delete [] Indices;

    return true;
}

bool CTerrain::LoadBinary(char *FileName)
{
    CString DirectoryFileName = ModuleDirectory + FileName;

    FILE *File;

    if(fopen_s(&File, DirectoryFileName, "rb") != 0)
    {
        ErrorLog.Append("Error opening file " + DirectoryFileName + "!\r\n");
        return false;
    }

    int Size;

    if(fread(&Size, sizeof(int), 1, File) != 1 || Size <= 0)
    {
        ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
        fclose(File);
        return false;
    }

    Destroy();

    this->Size = Size;
    SizeP1 = Size + 1;
    SizeD2 = (float)Size / 2.0f;

    VerticesCount = SizeP1 * SizeP1;

    Heights = new float[VerticesCount];

    if(fread(Heights, sizeof(float), VerticesCount, File) != VerticesCount)
    {
        ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
        fclose(File);
        Destroy();
        return false;
    }

    fclose(File);

    Min.x = Min.z = -SizeD2;
    Max.x = Max.z = SizeD2;

    Min.y = Max.y = Heights[0];

    for(int i = 1; i < VerticesCount; i++)
    {
        if(Heights[i] < Min.y) Min.y = Heights[i];
        if(Heights[i] > Max.y) Max.y = Heights[i];
    }

    CVertex *Vertices = new CVertex[VerticesCount];

    int i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            Vertices[i].Position = vec3((float)x - SizeD2, Heights[i], SizeD2 - (float)z);
            Vertices[i].Normal = normalize(vec3(GetHeight(x - 1, z) - GetHeight(x + 1, z), 2.0f, GetHeight(x, z + 1) - GetHeight(x, z - 1)));

            i++;
        }
    }

    glGenBuffers(1, &VertexBufferObject);
```

```
	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
	glBufferData(GL_ARRAY_BUFFER, VerticesCount * sizeof(CVertex), Vertices,
GL_STATIC_DRAW);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	int IndicesCount = Size * Size * 2 * 3;

	int *Indices = new int[IndicesCount];

	i = 0;

	for(int z = 0; z < Size; z++)
	{
		for(int x = 0; x < Size; x++)
		{
			Indices[i++] = GetIndex(x, z);
			Indices[i++] = GetIndex(x + 1, z);
			Indices[i++] = GetIndex(x + 1, z + 1);

			Indices[i++] = GetIndex(x + 1, z + 1);
			Indices[i++] = GetIndex(x, z + 1);
			Indices[i++] = GetIndex(x, z);
		}
	}

	BSPTree.Init(Vertices, Indices, IndicesCount, Min, Max);

	delete [] Vertices;
	delete [] Indices;

	return true;
}

bool CTerrain::SaveBinary(char *FileName)
{
	CString DirectoryFileName = ModuleDirectory + FileName;

	FILE *File;

	if(fopen_s(&File, DirectoryFileName, "wb+") != 0)
	{
		return false;
	}
	fwrite(&Size, sizeof(int), 1, File);

	fwrite(Heights, sizeof(float), VerticesCount, File);

	fclose(File);

	return true;
}

int CTerrain::CheckVisibility(CFrustum &Frustum, bool SortVisibleGeometryNodes)
{
	return BSPTree.CheckVisibility(Frustum, SortVisibleGeometryNodes);
}

void CTerrain::Render(bool VisualizeRenderingOrder)
{
	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);

	glEnableClientState(GL_VERTEX_ARRAY);
	glVertexPointer(3, GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 0));

	glEnableClientState(GL_NORMAL_ARRAY);
	glNormalPointer(GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 1));

	BSPTree.Render(VisualizeRenderingOrder);

	glDisableClientState(GL_NORMAL_ARRAY);
	glDisableClientState(GL_VERTEX_ARRAY);

	glBindBuffer(GL_ARRAY_BUFFER, 0);
}

void CTerrain::RenderAABB(int Depth)
{
	BSPTree.RenderAABB(Depth);
}

void CTerrain::Destroy()
{
	if(Heights != NULL)
	{
		delete [] Heights;
	}
	if(VertexBufferObject != 0)
	{
		glDeleteBuffers(1, &VertexBufferObject);
	}

	BSPTree.Destroy();

	SetDefaults();
}

vec3 CTerrain::GetMin()
{
	return Min;
}

vec3 CTerrain::GetMax()
{
	return Max;
}

int CTerrain::GetIndex(int X, int Z)
{
	return SizeP1 * Z + X;
}

float CTerrain::GetHeight(int X, int Z)
{
	return Heights[GetIndex(X < 0 ? 0 : X > Size ? Size : X, Z < 0 ? 0 : Z > Size ? Size : Z)];
}

float CTerrain::GetHeight(float X, float Z)
{
	Z = -Z;

	X += SizeD2;
	Z += SizeD2;

	float Size = (float)this->Size;

	if(X < 0.0f) X = 0.0f;
	if(X > Size) X = Size;
	if(Z < 0.0f) Z = 0.0f;
	if(Z > Size) Z = Size;
	int ix = (int)X, ixp1 = ix + 1;
	int iz = (int)Z, izp1 = iz + 1;

	float fx = X - (float)ix;
	float fz = Z - (float)iz;

	float a = GetHeight(ix, iz);
	float b = GetHeight(ixp1, iz);
	float c = GetHeight(ix, izp1);
	float d = GetHeight(ixp1, izp1);

	float ab = a + (b - a) * fx;
	float cd = c + (d - c) * fx;

	return ab + (cd - ab) * fz;
}

float CTerrain::GetHeight(float *Heights, int Size, float X, float Z)
{
	float SizeM1F = (float)Size - 1.0f;

	if(X < 0.0f) X = 0.0f;
	if(X > SizeM1F) X = SizeM1F;
	if(Z < 0.0f) Z = 0.0f;
	if(Z > SizeM1F) Z = SizeM1F;

	int ix = (int)X, ixp1 = ix + 1;
	int iz = (int)Z, izp1 = iz + 1;

	int SizeM1 = Size - 1;

	if(ixp1 > SizeM1) ixp1 = SizeM1;
	if(izp1 > SizeM1) izp1 = SizeM1;

	float fx = X - (float)ix;
	float fz = Z - (float)iz;

	int izMSize = iz * Size, izp1MSize = izp1 * Size;

	float a = Heights[izMSize + ix];
	float b = Heights[izMSize + ixp1];
	float c = Heights[izp1MSize + ix];
	float d = Heights[izp1MSize + ixp1];
	float ab = a + (b - a) * fx;
	float cd = c + (d - c) * fx;

	return ab + (cd - ab) * fz;
}

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

COpenGLRenderer::COpenGLRenderer()
{
	ViewMatrix = mat4x4(1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f);

	Wireframe = false;
	RenderAABB = false;
	RenderTree2D = false;
	VisualizeRenderingOrder = false;
	SortVisibleGeometryNodes = true;
	VisibilityCheckingPerformanceTest = false;

	Depth = -1;
}

COpenGLRenderer::~COpenGLRenderer()
{
}

bool COpenGLRenderer::Init()
{
	bool Error = false;

	Error |= !Shader.Load("glsl120shader.vs", "glsl120shader.fs");

	Error |= !Terrain.LoadBinary("terrain1.bin");

	if(Error)
	{
		return false;
	}

	Shader.UniformLocations = new GLuint[1];
	Shader.UniformLocations[0] = glGetUniformLocation(Shader, "CameraPosition");
```

```
    float Height = Terrain.GetHeight(0.0f, 0.0f);

    Camera.Look(vec3(0.0f, Height + 1.75f, 0.0f), vec3(0.0f, Height + 1.75f, -1.0f));

    return true;
}

void COpenGLRenderer::Render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    if(!VisibilityCheckingPerformanceTest)
    {
        if(!RenderTree2D)
        {
            glMatrixMode(GL_PROJECTION);
            glLoadMatrixf(&Camera.ProjectionMatrix);

            glMatrixMode(GL_MODELVIEW);
            glLoadMatrixf(&Camera.ViewMatrix);

            glEnable(GL_DEPTH_TEST);
            glEnable(GL_CULL_FACE);
        }
        else
        {
            glMatrixMode(GL_PROJECTION);
            glLoadMatrixf(&ProjectionMatrix);

            glMatrixMode(GL_MODELVIEW);
            glLoadMatrixf(&ViewMatrix);
        }
    }

    int TrianglesRendered = Terrain.CheckVisibility(Camera.Frustum,
SortVisibleGeometryNodes);

    if(!VisibilityCheckingPerformanceTest)
    {
        if(!RenderTree2D && Wireframe)
        {
            TrianglesRendered *= 2;
        }
    }

    Text.Set("Triangles rendered: %d", TrianglesRendered);

    if(!VisibilityCheckingPerformanceTest)
    {
        if(!RenderTree2D && Wireframe)
        {
            glColor3f(0.0f, 0.0f, 0.0f);

            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

            Terrain.Render();

            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        }

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

        glUseProgram(Shader);
        glUniform3fv(Shader.UniformLocations[0], 1, !RenderTree2D ? &Camera.Position :
&vec3(0.0f, 4096.0f, 0.0f));

        Terrain.Render(VisualizeRenderingOrder);

        glUseProgram(0);

        if(RenderAABB)
        {
            glColor3f(0.0f, 1.0f, 0.0f);

            Terrain.RenderAABB(Depth);
        }

        if(!RenderTree2D)
        {
            glDisable(GL_CULL_FACE);
            glDisable(GL_DEPTH_TEST);
        }
        else
        {
            glColor3f(1.0f, 0.5f, 0.25f);

            Camera.Frustum.Render();
```

```
		}
	}
}

void COpenGLRenderer::Animate(float FrameTime)
{
}

void COpenGLRenderer::Resize(int Width, int Height)
{
	this->Width = Width;
	this->Height = Height;

	glViewport(0, 0, Width, Height);

	Camera.SetPerspective(45.0f, (float)Width / (float)Height, 0.125f, 1024.0f);

	CalculateProjectionMatrix();
}

void COpenGLRenderer::Destroy()
{
	Shader.Destroy();

	Terrain.Destroy();
}

void COpenGLRenderer::CalculateProjectionMatrix()
{
	float Aspect = (float)Width / (float)Height;

	vec3 Min = Terrain.GetMin(), Max = Terrain.GetMax();

	ProjectionMatrix = ortho(Min.x * Aspect, Max.x * Aspect, -Max.z, -Min.z, -4096.0f,
4096.0f);
}

void COpenGLRenderer::CheckCameraTerrainPosition(vec3 &Movement)
{
	vec3 CameraPosition = Camera.Reference + Movement, Min = Terrain.GetMin(), Max =
Terrain.GetMax();

	if(CameraPosition.x < Min.x) Movement += vec3(Min.x - CameraPosition.x, 0.0f, 0.0f);
	if(CameraPosition.x > Max.x) Movement += vec3(Max.x - CameraPosition.x, 0.0f, 0.0f);
	if(CameraPosition.z < Min.z) Movement += vec3(0.0f, 0.0f, Min.z - CameraPosition.z);
	if(CameraPosition.z > Max.z) Movement += vec3(0.0f, 0.0f, Max.z - CameraPosition.z);

	CameraPosition = Camera.Reference + Movement;

	float Height = Terrain.GetHeight(CameraPosition.x, CameraPosition.z);

	Movement += vec3(0.0f, Height + 1.75f - Camera.Reference.y, 0.0f);
}

void COpenGLRenderer::CheckCameraKeys(float FrameTime)
{
	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)
	{
		vec3 Movement = Camera.OnKeys(Keys, FrameTime * 0.5f);

		CheckCameraTerrainPosition(Movement);

		Camera.Move(Movement);
	}
}

void COpenGLRenderer::OnKeyDown(UINT Key)
{
	switch(Key)
	{
		case VK_F1:
			Wireframe = !Wireframe;
			break;

		case VK_F2:
			RenderAABB = !RenderAABB;
			break;

		case VK_F3:
			RenderTree2D = !RenderTree2D;
			break;

		case VK_F4:
			VisualizeRenderingOrder = !VisualizeRenderingOrder;
			break;

		case VK_F5:
			SortVisibleGeometryNodes = !SortVisibleGeometryNodes;
			break;

		case VK_F6:
			VisibilityCheckingPerformanceTest = !VisibilityCheckingPerformanceTest;
			break;

		case VK_F7:
			Terrain.SaveBinary("terrain-saved.bin");
			break;

		case '1':
			if(Terrain.LoadBinary("terrain1.bin")) { CalculateProjectionMatrix(); vec3 
Movement; CheckCameraTerrainPosition(Movement); Camera.Move(Movement); }
			break;

		case '2':
			if(Terrain.LoadTexture2D("terrain2.jpg", 32.0f, -16.0f)) 
{ CalculateProjectionMatrix(); vec3 Movement; CheckCameraTerrainPosition(Movement); 
Camera.Move(Movement); }
			break;

		case '3':
			if(Terrain.LoadTexture2D("terrain3.jpg", 128.0f, -64.0f)) 
{ CalculateProjectionMatrix(); vec3 Movement; CheckCameraTerrainPosition(Movement); 
Camera.Move(Movement); }
			break;

		case '4':
			if(Terrain.LoadTexture2D("terrain4.jpg", 128.0f, -64.0f)) 
{ CalculateProjectionMatrix(); vec3 Movement; CheckCameraTerrainPosition(Movement); 
Camera.Move(Movement); }
			break;
		case VK_ADD:
			Depth++;
			break;

		case VK_SUBTRACT:
			if(Depth > -1) Depth--;
			break;
	}
}

void COpenGLRenderer::OnLButtonDown(int X, int Y)
{
	LastClickedX = X;
	LastClickedY = Y;
}

void COpenGLRenderer::OnLButtonUp(int X, int Y)
{
	if(X == LastClickedX && Y == LastClickedY)
	{
	}
}

void COpenGLRenderer::OnMouseMove(int X, int Y)
{
	if(GetKeyState(VK_RBUTTON) & 0x80)
	{
		Camera.OnMouseMove(LastX - X, LastY - Y);
	}

	LastX = X;
	LastY = Y;
}

void COpenGLRenderer::OnMouseWheel(short zDelta)
{
	Camera.OnMouseWheel(zDelta);
}

void COpenGLRenderer::OnRButtonDown(int X, int Y)
{
	LastClickedX = X;
	LastClickedY = Y;
```

```
}

void COpenGLRenderer::OnRButtonUp(int X, int Y)
{
	if(X == LastClickedX && Y == LastClickedY)
	{
	}
}
```

第六章 地形渲染、四叉树、视锥体裁剪、阴影

第一节 Shader Source

<1>:glsl120shader.vs

```
#version 120
uniform mat4x4 ShadowMatrix;
varying vec3 var_Normal;
void main()
{
    gl_FrontColor = gl_Color;
    gl_TexCoord[0] = ShadowMatrix * gl_Vertex;
    var_Normal = gl_Normal;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

glsl120shader.fs

```
#version 120
uniform sampler2D ShadowMap, RotationTexture;
uniform vec3 LightDirection;
uniform float Scale, Radius;
varying vec3 var_Normal;
vec2 PoissonDisk[16] = vec2[](
   vec2( -0.94201624, -0.39906216 ),
   vec2( 0.94558609, -0.76890725 ),
   vec2( -0.094184101, -0.92938870 ),
   vec2( 0.34495938, 0.29387760 ),
   vec2( -0.91588581, 0.45771432 ),
   vec2( -0.81544232, -0.87912464 ),
   vec2( -0.38277543, 0.27676845 ),
   vec2( 0.97484398, 0.75648379 ),
   vec2( 0.44323325, -0.97511554 ),
   vec2( 0.53742981, -0.47373420 ),
   vec2( -0.26496911, -0.41893023 ),
   vec2( 0.79197514, 0.19090188 ),
   vec2( -0.24188840, 0.99706507 ),
   vec2( -0.81409955, 0.91437590 ),
   vec2( 0.19984126, 0.78641367 ),
   vec2( 0.14383161, -0.14100790 )
);

void main()
{
    vec3 ShadowTexCoord = gl_TexCoord[0].xyz / gl_TexCoord[0].w;
```

```
	ShadowTexCoord.z -= 0.005;

	vec4 ScaleRotationVector = (texture2D(RotationTexture, gl_FragCoord.st * Scale) * 2.0 - 1.0) * Radius;
	mat2x2 ScaleRotationMatrix = mat2x2(ScaleRotationVector.xy, ScaleRotationVector.zw);
	float Shadow = 0.0;
	for(int i = 0; i < 16; i++)
	{
		float Depth = texture2D(ShadowMap, ShadowTexCoord.st + ScaleRotationMatrix * PoissonDisk[i]).r;
		if(ShadowTexCoord.z < Depth)
		{
			Shadow += 1.0;
		}
	}
	Shadow /= 16.0;
	vec3 Normal = normalize(var_Normal);
	float NdotLD = max(0.0, dot(Normal, LightDirection));
	gl_FragColor = vec4(gl_Color.rgb * (0.25 + 0.75 * NdotLD * Shadow), 1.0);
}
```

第二节 Source Code Header

```
class CPlane
{
private:
	vec3 N;
	float ND;
	int O;

public:
	CPlane();
	~CPlane();

public:
	void Set(const vec3 &A, const vec3 &B, const vec3 &C);
	bool AABBBehind(const vec3 *AABBVertices);
	float AABBDistance(const vec3 *AABBVertices);
};

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

class CFrustum
{
```

```
private:
	vec3 Vertices[8];

private:
	CPlane Planes[6];

public:
	CFrustum();
	~CFrustum();

public:
	void Set(const mat4x4 &ViewProjectionMatrixInverse);
	bool AABBVisible(const vec3 *AABBVertices);
	float AABBDistance(const vec3 *AABBVertices);
	void Render();
};

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

class CCamera
{
public:
	vec3 X, Y, Z, Position, Reference;

public:
	mat4x4 ViewMatrix, ViewMatrixInverse, ProjectionMatrix, ProjectionMatrixInverse, ViewProjectionMatrix, ViewProjectionMatrixInverse;

public:
	CFrustum Frustum;

public:
	CCamera();
	~CCamera();

public:
	void Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference = false);
	void Move(const vec3 &Movement);
	vec3 OnKeys(BYTE Keys, float FrameTime);
	void OnMouseMove(int dx, int dy);
	void OnMouseWheel(float zDelta);
	void SetPerspective(float fovy, float aspect, float n, float f);
```

```
private:
	void CalculateViewMatrix();
};

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

class CVertex
{
public:
	vec3 Position;
	vec3 Normal;
};

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

class CAABB
{
private:
	vec3 Vertices[8];

public:
	CAABB();
	~CAABB();

public:
	void Set(const vec3 &Min, const vec3 &Max);
	bool PointInside(const vec3 &Point);
	bool Visible(CFrustum &Frustum);
	float Distance(CFrustum &Frustum);
	void Render();
};

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

class CBSPTreeNode
{
private:
	vec3 Min, Max;

private:
	int Depth;

private:
	CAABB AABB;

private:
	bool Visible;
	float Distance;

private:
	int *Indices;

private:
	int IndicesCount;

private:
	GLuint IndexBufferObject;

private:
	CBSPTreeNode *Children[2];

public:
	CBSPTreeNode();
	~CBSPTreeNode();

private:
	void SetDefaults();

public:
	void InitAABB(const vec3 &Min, const vec3 &Max, int Depth, float MinAABBSize);
	bool CheckTriangle(CVertex *Vertices, int *Indices, int A, int B, int C);
	void AllocateMemory();
	bool AddTriangle(CVertex *Vertices, int *Indices, int A, int B, int C);
	void ResetAABB(CVertex *Vertices);
	int InitIndexBufferObject();
	int CheckVisibility(CFrustum &Frustum, CBSPTreeNode **VisibleGeometryNodes, int &VisibleGeometryNodesCount);
	float GetDistance();
	void Render();
	void RenderAABB(int Depth);
	void Destroy();
};

// ----------------------------------------------------------------------------------------------------------------------------
```

```
-----------------------------------------------

class CBSPTree
{
private:
	CBSPTreeNode *Root;

private:
	CBSPTreeNode **VisibleGeometryNodes;
	int VisibleGeometryNodesCount;

public:
	CBSPTree();
	~CBSPTree();

private:
	void SetDefaults();

public:
	void Init(CVertex *Vertices, int *Indices, int IndicesCount, const vec3 &Min, const vec3
&Max, float MinAABBSize = 16.0f);
	void QuickSortVisibleGeometryNodes(int Left, int Right);
	int CheckVisibility(CFrustum &Frustum, bool SortVisibleGeometryNodes);
	void Render(bool VisualizeRenderingOrder);
	void RenderAABB(int Depth);
	void Destroy();
};

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

class CTerrain
{
private:
	int Size, SizeP1;
	float SizeD2;

private:
	vec3 Min, Max;

private:
	float *Heights;

private:
	int VerticesCount, IndicesCount;

private:
	GLuint VertexBufferObject, IndexBufferObject;

public:
	CBSPTree BSPTree;

public:
	CTerrain();
	~CTerrain();

private:
	void SetDefaults();

public:
	bool LoadTexture2D(char *FileName, float Scale = 256.0f, float Offset = -128.0f);
	bool LoadBinary(char *FileName);
	bool SaveBinary(char *FileName);
	int CheckVisibility(CFrustum &Frustum, bool SortVisibleGeometryNodes = true);
	void Render(bool VisualizeRenderingOrder = false);
	void RenderSlow();
	void RenderSlowToShadowMap();
	void RenderAABB(int Depth = -1);
	void Destroy();

public:
	int GetSize();
	vec3 GetMin();
	vec3 GetMax();
	void GetMinMax(mat4x4 &ViewMatrix, vec3 &Min, vec3 &Max);
	int GetTrianglesCount();

private:
	int GetIndex(int X, int Z);
	float GetHeight(int X, int Z);

public:
	float GetHeight(float X, float Z);

private:
	float GetHeight(float *Heights, int Size, float X, float Z);
};
```

```
// ----------------------------------------------------------------------------
----------------------------------------------

#define SHADOW_MAP_SIZE 4096

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

class COpenGLRenderer
{
private:
	int LastX, LastY, LastClickedX, LastClickedY;

private:
	int Width, Height;

private:
	CCamera Camera;

private:
	CShaderProgram Shader;

private:
	CTerrain Terrain;

private:
	float LightAngle;

private:
	mat4x4 LightViewMatrix, LightProjectionMatrix, ShadowMatrix;

private:
	int ShadowMapSize;
	GLuint ShadowMap, RotationTexture, FBO;

private:
	bool Wireframe, DisplayShadowMap, RenderAABB, VisualizeRenderingOrder,
SortVisibleGeometryNodes, RenderSlow;
	int Depth;

public:
	COpenGLRenderer();
	~COpenGLRenderer();
public:
	bool Init();
	void Render();
	void Animate(float FrameTime);
	void Resize(int Width, int Height);
	void Destroy();

private:
	void RenderShadowMap();
	void CheckCameraTerrainPosition(vec3 &Movement);

public:
	void CheckCameraKeys(float FrameTime);

public:
	void OnKeyDown(UINT Key);
	void OnLButtonDown(int X, int Y);
	void OnLButtonUp(int X, int Y);
	void OnMouseMove(int X, int Y);
	void OnMouseWheel(short zDelta);
	void OnRButtonDown(int X, int Y);
	void OnRButtonUp(int X, int Y);
};
```

第三节 Source Code Cpp

```
CPlane::CPlane()
{
}

CPlane::~CPlane()
{
}

void CPlane::Set(const vec3 &A, const vec3 &B, const vec3 &C)
{
	N = normalize(cross(B - A, C - A));
	ND = dot(N, A);
	O = N.z < 0.0f ? (N.y < 0.0f ? (N.x < 0.0f ? 0 : 1) : (N.x < 0.0f ? 2 : 3)) : (N.y < 0.0f ? (N.x < 0.0f ? 4 : 5) : (N.x < 0.0f ? 6 : 7));
}

bool CPlane::AABBBehind(const vec3 *AABBVertices)
{
	return dot(N, AABBVertices[O]) < ND;
```

```
}

float CPlane::AABBDistance(const vec3 *AABBVertices)
{
	return dot(N, AABBVertices[O]);
}

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

CFrustum::CFrustum()
{
}

CFrustum::~CFrustum()
{
}

void CFrustum::Set(const mat4x4 &ViewProjectionMatrixInverse)
{
	vec4 A = ViewProjectionMatrixInverse * vec4(-1.0f, -1.0f,  1.0f, 1.0f);
	vec4 B = ViewProjectionMatrixInverse * vec4( 1.0f, -1.0f,  1.0f, 1.0f);
	vec4 C = ViewProjectionMatrixInverse * vec4(-1.0f,  1.0f,  1.0f, 1.0f);
	vec4 D = ViewProjectionMatrixInverse * vec4( 1.0f,  1.0f,  1.0f, 1.0f);
	vec4 E = ViewProjectionMatrixInverse * vec4(-1.0f, -1.0f, -1.0f, 1.0f);
	vec4 F = ViewProjectionMatrixInverse * vec4( 1.0f, -1.0f, -1.0f, 1.0f);
	vec4 G = ViewProjectionMatrixInverse * vec4(-1.0f,  1.0f, -1.0f, 1.0f);
	vec4 H = ViewProjectionMatrixInverse * vec4( 1.0f,  1.0f, -1.0f, 1.0f);

	Vertices[0] = vec3(A.x / A.w, A.y / A.w, A.z / A.w);
	Vertices[1] = vec3(B.x / B.w, B.y / B.w, B.z / B.w);
	Vertices[2] = vec3(C.x / C.w, C.y / C.w, C.z / C.w);
	Vertices[3] = vec3(D.x / D.w, D.y / D.w, D.z / D.w);
	Vertices[4] = vec3(E.x / E.w, E.y / E.w, E.z / E.w);
	Vertices[5] = vec3(F.x / F.w, F.y / F.w, F.z / F.w);
	Vertices[6] = vec3(G.x / G.w, G.y / G.w, G.z / G.w);
	Vertices[7] = vec3(H.x / H.w, H.y / H.w, H.z / H.w);

	Planes[0].Set(Vertices[4], Vertices[0], Vertices[2]);
	Planes[1].Set(Vertices[1], Vertices[5], Vertices[7]);
	Planes[2].Set(Vertices[4], Vertices[5], Vertices[1]);
	Planes[3].Set(Vertices[2], Vertices[3], Vertices[7]);
	Planes[4].Set(Vertices[0], Vertices[1], Vertices[3]);
	Planes[5].Set(Vertices[5], Vertices[4], Vertices[6]);
```

```
}

bool CFrustum::AABBVisible(const vec3 *AABBVertices)
{
	for(int i = 0; i < 6; i++)
	{
		if(Planes[i].AABBBehind(AABBVertices))
		{
			return false;
		}
	}

	return true;
}

float CFrustum::AABBDistance(const vec3 *AABBVertices)
{
	return Planes[5].AABBDistance(AABBVertices);
}

void CFrustum::Render()
{
	glBegin(GL_LINES);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[1]);
	glVertex3fv(&Vertices[2]); glVertex3fv(&Vertices[3]);
	glVertex3fv(&Vertices[4]); glVertex3fv(&Vertices[5]);
	glVertex3fv(&Vertices[6]); glVertex3fv(&Vertices[7]);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[2]);
	glVertex3fv(&Vertices[1]); glVertex3fv(&Vertices[3]);
	glVertex3fv(&Vertices[4]); glVertex3fv(&Vertices[6]);
	glVertex3fv(&Vertices[5]); glVertex3fv(&Vertices[7]);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[4]);
	glVertex3fv(&Vertices[1]); glVertex3fv(&Vertices[5]);
	glVertex3fv(&Vertices[2]); glVertex3fv(&Vertices[6]);
	glVertex3fv(&Vertices[3]); glVertex3fv(&Vertices[7]);

	glEnd();
}

// ---------------------------------------------------------------------------
----------------------------------------------
```

```
CCamera::CCamera()
{
	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);

	CalculateViewMatrix();
}

CCamera::~CCamera()
{
}

void CCamera::Look(const vec3 &Position, const vec3 &Reference, bool
RotateAroundReference)
{
	this->Position = Position;
	this->Reference = Reference;

	Z = normalize(Position - Reference);

	GetXY(Z, X, Y);

	if(!RotateAroundReference)
	{
		this->Reference = 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::SetPerspective(float fovy, float aspect, float n, float f)
{
    ProjectionMatrix = perspective(fovy, aspect, n, f);
    ProjectionMatrixInverse = inverse(ProjectionMatrix);
    ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;
    ViewProjectionMatrixInverse = ViewMatrixInverse * ProjectionMatrixInverse;
```

```
	Frustum.Set(ViewProjectionMatrixInverse);
}

void CCamera::CalculateViewMatrix()
{
	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);
	ViewMatrixInverse = inverse(ViewMatrix);
	ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;
	ViewProjectionMatrixInverse = ViewMatrixInverse * ProjectionMatrixInverse;

	Frustum.Set(ViewProjectionMatrixInverse);
}

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

CAABB::CAABB()
{
}

CAABB::~CAABB()
{
}

void CAABB::Set(const vec3 &Min, const vec3 &Max)
{
	Vertices[0] = vec3(Min.x, Min.y, Min.z);
	Vertices[1] = vec3(Max.x, Min.y, Min.z);
	Vertices[2] = vec3(Min.x, Max.y, Min.z);
	Vertices[3] = vec3(Max.x, Max.y, Min.z);
	Vertices[4] = vec3(Min.x, Min.y, Max.z);
	Vertices[5] = vec3(Max.x, Min.y, Max.z);
	Vertices[6] = vec3(Min.x, Max.y, Max.z);
	Vertices[7] = vec3(Max.x, Max.y, Max.z);
}

bool CAABB::PointInside(const vec3 &Point)
{
	if(Point.x < Vertices[0].x) return false;
	if(Point.y < Vertices[0].y) return false;
	if(Point.z < Vertices[0].z) return false;

	if(Point.x > Vertices[7].x) return false;
	if(Point.y > Vertices[7].y) return false;
	if(Point.z > Vertices[7].z) return false;

	return true;
}

bool CAABB::Visible(CFrustum &Frustum)
{
	return Frustum.AABBVisible(Vertices);
}

float CAABB::Distance(CFrustum &Frustum)
{
	return Frustum.AABBDistance(Vertices);
}

void CAABB::Render()
{
	glBegin(GL_LINES);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[1]);
	glVertex3fv(&Vertices[2]); glVertex3fv(&Vertices[3]);
	glVertex3fv(&Vertices[4]); glVertex3fv(&Vertices[5]);
	glVertex3fv(&Vertices[6]); glVertex3fv(&Vertices[7]);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[2]);
	glVertex3fv(&Vertices[1]); glVertex3fv(&Vertices[3]);
	glVertex3fv(&Vertices[4]); glVertex3fv(&Vertices[6]);
	glVertex3fv(&Vertices[5]); glVertex3fv(&Vertices[7]);

	glVertex3fv(&Vertices[0]); glVertex3fv(&Vertices[4]);
	glVertex3fv(&Vertices[1]); glVertex3fv(&Vertices[5]);
	glVertex3fv(&Vertices[2]); glVertex3fv(&Vertices[6]);
	glVertex3fv(&Vertices[3]); glVertex3fv(&Vertices[7]);

	glEnd();
}

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

CBSPTreeNode::CBSPTreeNode()
{
	SetDefaults();
```

```
}

CBSPTreeNode::~CBSPTreeNode()
{
}

void CBSPTreeNode::SetDefaults()
{
    Min = Max = vec3(0.0f);

    Depth = 0;

    Indices = NULL;

    IndicesCount = 0;

    IndexBufferObject = 0;

    Children[0] = NULL;
    Children[1] = NULL;
}

void CBSPTreeNode::InitAABB(const vec3 &Min, const vec3 &Max, int Depth, float
MinAABBSize)
{
    this->Min = Min;
    this->Max = Max;

    this->Depth = Depth;

    vec3 Mid = (Min + Max) / 2.0f;
    vec3 Size = Max - Min;

    AABB.Set(Min, Max);

    if(Size.x > MinAABBSize || Size.z > MinAABBSize)
    {
        Children[0] = new CBSPTreeNode();
        Children[1] = new CBSPTreeNode();

        if(Size.x >= Size.z)
        {
            Children[0]->InitAABB(vec3(Min.x, Min.y, Min.z), vec3(Mid.x, Max.y, Max.z),
Depth + 1, MinAABBSize);
```

```
            Children[1]->InitAABB(vec3(Mid.x, Min.y, Min.z), vec3(Max.x, Max.y, Max.z),
Depth + 1, MinAABBSize);
        }
        else
        {
            Children[0]->InitAABB(vec3(Min.x, Min.y, Min.z), vec3(Max.x, Max.y, Mid.z),
Depth + 1, MinAABBSize);
            Children[1]->InitAABB(vec3(Min.x, Min.y, Mid.z), vec3(Max.x, Max.y, Max.z),
Depth + 1, MinAABBSize);
        }
    }
}

bool CBSPTreeNode::CheckTriangle(CVertex *Vertices, int *Indices, int A, int B, int C)
{
    if(AABB.PointInside(Vertices[Indices[A]].Position))
    {
        if(AABB.PointInside(Vertices[Indices[B]].Position))
        {
            if(AABB.PointInside(Vertices[Indices[C]].Position))
            {
                bool BelongsToAChild = false;

                if(Children[0] != NULL)
                {
                    BelongsToAChild |= Children[0]->CheckTriangle(Vertices, Indices, A,
B, C);
                }

                if(Children[1] != NULL && !BelongsToAChild)
                {
                    BelongsToAChild |= Children[1]->CheckTriangle(Vertices, Indices, A,
B, C);
                }

                if(!BelongsToAChild)
                {
                    IndicesCount += 3;
                }

                return true;
            }
        }
    }
    return false;
}

void CBSPTreeNode::AllocateMemory()
{
    if(IndicesCount > 0)
    {
        Indices = new int[IndicesCount];
        IndicesCount = 0;
    }

    if(Children[0] != NULL)
    {
        Children[0]->AllocateMemory();
    }

    if(Children[1] != NULL)
    {
        Children[1]->AllocateMemory();
    }
}

bool CBSPTreeNode::AddTriangle(CVertex *Vertices, int *Indices, int A, int B, int C)
{
    if(AABB.PointInside(Vertices[Indices[A]].Position))
    {
        if(AABB.PointInside(Vertices[Indices[B]].Position))
        {
            if(AABB.PointInside(Vertices[Indices[C]].Position))
            {
                bool BelongsToAChild = false;

                if(Children[0] != NULL)
                {
                    BelongsToAChild |= Children[0]->AddTriangle(Vertices, Indices, A, B,
C);
                }

                if(Children[1] != NULL && !BelongsToAChild)
                {
                    BelongsToAChild |= Children[1]->AddTriangle(Vertices, Indices, A, B,
C);
                }
                if(!BelongsToAChild)
                {
                    this->Indices[IndicesCount++] = Indices[A];
                    this->Indices[IndicesCount++] = Indices[B];
                    this->Indices[IndicesCount++] = Indices[C];
                }

                return true;
            }
        }
    }

    return false;
}

void CBSPTreeNode::ResetAABB(CVertex *Vertices)
{
    float MinY = Min.y, MaxY = Max.y;

    Min.y = MaxY;
    Max.y = MinY;

    if(IndicesCount > 0)
    {
        for(int i = 0; i < IndicesCount; i++)
        {
            if(Vertices[Indices[i]].Position.y < Min.y) Min.y = Vertices[Indices[i]].Position.y;
            if(Vertices[Indices[i]].Position.y > Max.y) Max.y = Vertices[Indices[i]].Position.y;
        }
    }

    if(Children[0] != NULL)
    {
        Children[0]->ResetAABB(Vertices);

        if(Children[0]->Min.y < Min.y) Min.y = Children[0]->Min.y;
        if(Children[0]->Max.y > Max.y) Max.y = Children[0]->Max.y;
    }

    if(Children[1] != NULL)
    {
        Children[1]->ResetAABB(Vertices);
        if(Children[1]->Min.y < Min.y) Min.y = Children[1]->Min.y;
        if(Children[1]->Max.y > Max.y) Max.y = Children[1]->Max.y;
    }

    AABB.Set(Min, Max);
}

int CBSPTreeNode::InitIndexBufferObject()
{
    int GeometryNodesCount = 0;

    if(IndicesCount > 0)
    {
        glGenBuffers(1, &IndexBufferObject);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndicesCount * sizeof(int), Indices,
GL_STATIC_DRAW);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

        delete [] Indices;
        Indices = NULL;

        GeometryNodesCount++;
    }

    if(Children[0] != NULL)
    {
        GeometryNodesCount += Children[0]->InitIndexBufferObject();
    }

    if(Children[1] != NULL)
    {
        GeometryNodesCount += Children[1]->InitIndexBufferObject();
    }

    return GeometryNodesCount;
}

int CBSPTreeNode::CheckVisibility(CFrustum &Frustum, CBSPTreeNode
**VisibleGeometryNodes, int &VisibleGeometryNodesCount)
{
    int TrianglesRendered = 0;
```

```
	Visible = AABB.Visible(Frustum);

	if(Visible)
	{
		if(IndicesCount > 0)
		{
			Distance = AABB.Distance(Frustum);

			VisibleGeometryNodes[VisibleGeometryNodesCount++] = this;

			TrianglesRendered += IndicesCount / 3;
		}

		if(Children[0] != NULL)
		{
			TrianglesRendered += Children[0]->CheckVisibility(Frustum,
VisibleGeometryNodes, VisibleGeometryNodesCount);
		}

		if(Children[1] != NULL)
		{
			TrianglesRendered += Children[1]->CheckVisibility(Frustum,
VisibleGeometryNodes, VisibleGeometryNodesCount);
		}
	}

	return TrianglesRendered;
}

float CBSPTreeNode::GetDistance()
{
	return Distance;
}

void CBSPTreeNode::Render()
{
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);

	glDrawElements(GL_TRIANGLES, IndicesCount, GL_UNSIGNED_INT, NULL);

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}

void CBSPTreeNode::RenderAABB(int Depth)
```

```
{
	if(Visible)
	{
		if(Depth == -1 || Depth == this->Depth)
		{
			AABB.Render();
		}

		if(Children[0] != NULL)
		{
			Children[0]->RenderAABB(Depth);
		}

		if(Children[1] != NULL)
		{
			Children[1]->RenderAABB(Depth);
		}
	}
}

void CBSPTreeNode::Destroy()
{
	if(Indices != NULL)
	{
		delete [] Indices;
	}

	if(IndexBufferObject != 0)
	{
		glDeleteBuffers(1, &IndexBufferObject);
	}

	if(Children[0] != NULL)
	{
		Children[0]->Destroy();
		delete Children[0];
	}

	if(Children[1] != NULL)
	{
		Children[1]->Destroy();
		delete Children[1];
	}
```

```
    SetDefaults();
}

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

CBSPTree::CBSPTree()
{
    SetDefaults();
}

CBSPTree::~CBSPTree()
{
}

void CBSPTree::SetDefaults()
{
    Root = NULL;

    VisibleGeometryNodes = NULL;
}

void CBSPTree::Init(CVertex *Vertices, int *Indices, int IndicesCount, const vec3 &Min, const vec3 &Max, float MinAABBSize)
{
    Destroy();

    if(Vertices != NULL && Indices != NULL && IndicesCount > 0)
    {
        Root = new CBSPTreeNode();

        Root->InitAABB(Min, Max, 0, MinAABBSize);

        for(int i = 0; i < IndicesCount; i += 3)
        {
            Root->CheckTriangle(Vertices, Indices, i, i + 1, i + 2);
        }

        Root->AllocateMemory();

        for(int i = 0; i < IndicesCount; i += 3)
        {
            Root->AddTriangle(Vertices, Indices, i, i + 1, i + 2);
        }
        Root->ResetAABB(Vertices);

        int GeometryNodesCount = Root->InitIndexBufferObject();

        VisibleGeometryNodes = new CBSPTreeNode*[GeometryNodesCount];
    }
}

void CBSPTree::QuickSortVisibleGeometryNodes(int Left, int Right)
{
    float Pivot = VisibleGeometryNodes[(Left + Right) / 2]->GetDistance();
    int i = Left, j = Right;

    while(i <= j)
    {
        while(VisibleGeometryNodes[i]->GetDistance() < Pivot) i++;
        while(VisibleGeometryNodes[j]->GetDistance() > Pivot) j--;

        if(i <= j)
        {
            if(i != j)
            {
                CBSPTreeNode *Temp = VisibleGeometryNodes[i];
                VisibleGeometryNodes[i] = VisibleGeometryNodes[j];
                VisibleGeometryNodes[j] = Temp;
            }

            i++;
            j--;
        }
    }

    if(Left < j)
    {
        QuickSortVisibleGeometryNodes(Left, j);
    }

    if(i < Right)
    {
        QuickSortVisibleGeometryNodes(i, Right);
    }
}
```

```
int CBSPTree::CheckVisibility(CFrustum &Frustum, bool SortVisibleGeometryNodes)
{
    int TrianglesRendered = 0;

    VisibleGeometryNodesCount = 0;

    if(Root != NULL)
    {
        TrianglesRendered = Root->CheckVisibility(Frustum, VisibleGeometryNodes,
VisibleGeometryNodesCount);

        if(SortVisibleGeometryNodes)
        {
            if(VisibleGeometryNodesCount > 1)
            {
                QuickSortVisibleGeometryNodes(0, VisibleGeometryNodesCount - 1);
            }
        }
    }

    return TrianglesRendered;
}

void CBSPTree::Render(bool VisualizeRenderingOrder)
{
    if(VisibleGeometryNodesCount > 0)
    {
        if(!VisualizeRenderingOrder)
        {
            for(int i = 0; i < VisibleGeometryNodesCount; i++)
            {
                VisibleGeometryNodes[i]->Render();
            }
        }
        else
        {
            for(int i = 0; i < VisibleGeometryNodesCount; i++)
            {
                float Color = (float)(i + 1) / (float)VisibleGeometryNodesCount;

                glColor3f(Color, Color, Color);

                VisibleGeometryNodes[i]->Render();
            }
        }
    }
}

void CBSPTree::RenderAABB(int Depth)
{
    if(Root != NULL)
    {
        Root->RenderAABB(Depth);
    }
}

void CBSPTree::Destroy()
{
    if(Root != NULL)
    {
        Root->Destroy();
        delete Root;
    }

    if(VisibleGeometryNodes != NULL)
    {
        delete [] VisibleGeometryNodes;
    }

    SetDefaults();
}

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

CTerrain::CTerrain()
{
    SetDefaults();
}

CTerrain::~CTerrain()
{
}

void CTerrain::SetDefaults()
{
    Size = 0;
    SizeP1 = 0;
    SizeD2 = 0.0f;

    Min = Max = vec3(0.0f);

    Heights = NULL;

    VerticesCount = 0;
    IndicesCount = 0;

    VertexBufferObject = 0;
    IndexBufferObject = 0;
}

bool CTerrain::LoadTexture2D(char *FileName, float Scale, float Offset)
{
    CTexture Texture;

    if(!Texture.LoadTexture2D(FileName))
    {
        return false;
    }

    if(Texture.GetWidth() != Texture.GetHeight())
    {
        ErrorLog.Append("Unsupported texture dimensions (%s)!\r\n", FileName);
        Texture.Destroy();
        return false;
    }

    Destroy();

    Size = Texture.GetWidth();
    SizeP1 = Size + 1;
    SizeD2 = (float)Size / 2.0f;

    VerticesCount = SizeP1 * SizeP1;

    float *TextureHeights = new float[Size * Size];

    glBindTexture(GL_TEXTURE_2D, Texture);
    glGetTexImage(GL_TEXTURE_2D, 0, GL_GREEN, GL_FLOAT, TextureHeights);
    glBindTexture(GL_TEXTURE_2D, 0);

    Texture.Destroy();
    for(int i = 0; i < Size * Size; i++)
    {
        TextureHeights[i] = TextureHeights[i] * Scale + Offset;
    }

    Heights = new float[VerticesCount];

    int i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            Heights[i++] = GetHeight(TextureHeights, Size, (float)x - 0.5f, (float)z - 0.5f);
        }
    }

    delete [] TextureHeights;

    float *SmoothedHeights = new float[VerticesCount];

    i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            SmoothedHeights[i] = 0.0f;

            SmoothedHeights[i] += GetHeight(x - 1, z + 1) + GetHeight(x, z + 1) * 2 +
GetHeight(x + 1, z + 1);
            SmoothedHeights[i] += GetHeight(x - 1, z) * 2 + GetHeight(x, z) * 3 +
GetHeight(x + 1, z) * 2;
            SmoothedHeights[i] += GetHeight(x - 1, z - 1) + GetHeight(x, z - 1) * 2 +
GetHeight(x + 1, z - 1);

            SmoothedHeights[i] /= 15.0f;

            i++;
        }
    }

    delete [] Heights;
```

```
	Heights = SmoothedHeights;

	Min.x = Min.z = -SizeD2;
	Max.x = Max.z = SizeD2;

	Min.y = Max.y = Heights[0];

	for(int i = 1; i < VerticesCount; i++)
	{
		if(Heights[i] < Min.y) Min.y = Heights[i];
		if(Heights[i] > Max.y) Max.y = Heights[i];
	}

	CVertex *Vertices = new CVertex[VerticesCount];

	i = 0;

	for(int z = 0; z <= Size; z++)
	{
		for(int x = 0; x <= Size; x++)
		{
			Vertices[i].Position = vec3((float)x - SizeD2, Heights[i], SizeD2 - (float)z);
			Vertices[i].Normal = normalize(vec3(GetHeight(x - 1, z) - GetHeight(x + 1, z), 
2.0f, GetHeight(x, z + 1) - GetHeight(x, z - 1)));

			i++;
		}
	}

	glGenBuffers(1, &VertexBufferObject);

	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
	glBufferData(GL_ARRAY_BUFFER, VerticesCount * sizeof(CVertex), Vertices, 
GL_STATIC_DRAW);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	IndicesCount = Size * Size * 2 * 3;

	int *Indices = new int[IndicesCount];

	i = 0;

	for(int z = 0; z < Size; z++)
	{
		for(int x = 0; x < Size; x++)
		{
			Indices[i++] = GetIndex(x, z);
			Indices[i++] = GetIndex(x + 1, z);
			Indices[i++] = GetIndex(x + 1, z + 1);

			Indices[i++] = GetIndex(x + 1, z + 1);
			Indices[i++] = GetIndex(x, z + 1);
			Indices[i++] = GetIndex(x, z);
		}
	}

	glGenBuffers(1, &IndexBufferObject);

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndicesCount * sizeof(int), Indices,
GL_STATIC_DRAW);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

	BSPTree.Init(Vertices, Indices, IndicesCount, Min, Max);

	delete [] Vertices;
	delete [] Indices;

	return true;
}

bool CTerrain::LoadBinary(char *FileName)
{
	CString DirectoryFileName = ModuleDirectory + FileName;

	FILE *File;

	if(fopen_s(&File, DirectoryFileName, "rb") != 0)
	{
		ErrorLog.Append("Error opening file " + DirectoryFileName + "!\r\n");
		return false;
	}

	int Size;

	if(fread(&Size, sizeof(int), 1, File) != 1 || Size <= 0)
	{
```

```
	ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
	fclose(File);
	return false;
}

Destroy();

this->Size = Size;
SizeP1 = Size + 1;
SizeD2 = (float)Size / 2.0f;

VerticesCount = SizeP1 * SizeP1;

Heights = new float[VerticesCount];

if(fread(Heights, sizeof(float), VerticesCount, File) != VerticesCount)
{
	ErrorLog.Append("Error reading file " + DirectoryFileName + "!\r\n");
	fclose(File);
	Destroy();
	return false;
}

fclose(File);

Min.x = Min.z = -SizeD2;
Max.x = Max.z = SizeD2;

Min.y = Max.y = Heights[0];

for(int i = 1; i < VerticesCount; i++)
{
	if(Heights[i] < Min.y) Min.y = Heights[i];
	if(Heights[i] > Max.y) Max.y = Heights[i];
}

CVertex *Vertices = new CVertex[VerticesCount];

int i = 0;

for(int z = 0; z <= Size; z++)
{
	for(int x = 0; x <= Size; x++)
	{
			Vertices[i].Position = vec3((float)x - SizeD2, Heights[i], SizeD2 - (float)z);
			Vertices[i].Normal = normalize(vec3(GetHeight(x - 1, z) - GetHeight(x + 1, z),
2.0f, GetHeight(x, z + 1) - GetHeight(x, z - 1)));

			i++;
		}
	}

	glGenBuffers(1, &VertexBufferObject);

	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);
	glBufferData(GL_ARRAY_BUFFER, VerticesCount * sizeof(CVertex), Vertices,
GL_STATIC_DRAW);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	IndicesCount = Size * Size * 2 * 3;

	int *Indices = new int[IndicesCount];

	i = 0;

	for(int z = 0; z < Size; z++)
	{
		for(int x = 0; x < Size; x++)
		{
			Indices[i++] = GetIndex(x, z);
			Indices[i++] = GetIndex(x + 1, z);
			Indices[i++] = GetIndex(x + 1, z + 1);

			Indices[i++] = GetIndex(x + 1, z + 1);
			Indices[i++] = GetIndex(x, z + 1);
			Indices[i++] = GetIndex(x, z);
		}
	}

	glGenBuffers(1, &IndexBufferObject);

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndicesCount * sizeof(int), Indices,
GL_STATIC_DRAW);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

	BSPTree.Init(Vertices, Indices, IndicesCount, Min, Max);
	delete [] Vertices;
	delete [] Indices;

	return true;
}

bool CTerrain::SaveBinary(char *FileName)
{
	CString DirectoryFileName = ModuleDirectory + FileName;

	FILE *File;

	if(fopen_s(&File, DirectoryFileName, "wb+") != 0)
	{
		return false;
	}

	fwrite(&Size, sizeof(int), 1, File);

	fwrite(Heights, sizeof(float), VerticesCount, File);

	fclose(File);

	return true;
}

int CTerrain::CheckVisibility(CFrustum &Frustum, bool SortVisibleGeometryNodes)
{
	return BSPTree.CheckVisibility(Frustum, SortVisibleGeometryNodes);
}

void CTerrain::Render(bool VisualizeRenderingOrder)
{
	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);

	glEnableClientState(GL_VERTEX_ARRAY);
	glVertexPointer(3, GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 0));

	glEnableClientState(GL_NORMAL_ARRAY);
	glNormalPointer(GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 1));

	BSPTree.Render(VisualizeRenderingOrder);

	glDisableClientState(GL_NORMAL_ARRAY);
	glDisableClientState(GL_VERTEX_ARRAY);

	glBindBuffer(GL_ARRAY_BUFFER, 0);
}

void CTerrain::RenderSlow()
{
	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);

	glEnableClientState(GL_VERTEX_ARRAY);
	glVertexPointer(3, GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 0));

	glEnableClientState(GL_NORMAL_ARRAY);
	glNormalPointer(GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 1));

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);

	glDrawElements(GL_TRIANGLES, IndicesCount, GL_UNSIGNED_INT, NULL);

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

	glDisableClientState(GL_NORMAL_ARRAY);
	glDisableClientState(GL_VERTEX_ARRAY);

	glBindBuffer(GL_ARRAY_BUFFER, 0);
}

void CTerrain::RenderSlowToShadowMap()
{
	glBindBuffer(GL_ARRAY_BUFFER, VertexBufferObject);

	glEnableClientState(GL_VERTEX_ARRAY);
	glVertexPointer(3, GL_FLOAT, sizeof(CVertex), (void*)(sizeof(vec3) * 0));

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IndexBufferObject);

	glDrawElements(GL_TRIANGLES, IndicesCount, GL_UNSIGNED_INT, NULL);

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

	glDisableClientState(GL_VERTEX_ARRAY);

	glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

```
void CTerrain::RenderAABB(int Depth)
{
	BSPTree.RenderAABB(Depth);
}

void CTerrain::Destroy()
{
	if(Heights != NULL)
	{
		delete [] Heights;
	}

	if(VertexBufferObject != 0)
	{
		glDeleteBuffers(1, &VertexBufferObject);
	}

	if(IndexBufferObject != 0)
	{
		glDeleteBuffers(1, &IndexBufferObject);
	}

	BSPTree.Destroy();

	SetDefaults();
}

int CTerrain::GetSize()
{
	return Size;
}

vec3 CTerrain::GetMin()
{
	return Min;
}

vec3 CTerrain::GetMax()
{
	return Max;
}

void CTerrain::GetMinMax(mat4x4 &ViewMatrix, vec3 &Min, vec3 &Max)
```

```
{
    int i = 0;

    for(int z = 0; z <= Size; z++)
    {
        for(int x = 0; x <= Size; x++)
        {
            vec4 Position = ViewMatrix * vec4((float)x - SizeD2, Heights[i], SizeD2 -
(float)z, 1.0f);

            if(i == 0)
            {
                Min.x = Max.x = Position.x;
                Min.y = Max.y = Position.y;
                Min.z = Max.z = Position.z;
            }
            else
            {
                if(Position.x < Min.x) Min.x = Position.x;
                if(Position.y < Min.y) Min.y = Position.y;
                if(Position.z < Min.z) Min.z = Position.z;

                if(Position.x > Max.x) Max.x = Position.x;
                if(Position.y > Max.y) Max.y = Position.y;
                if(Position.z > Max.z) Max.z = Position.z;
            }

            i++;
        }
    }
}

int CTerrain::GetTrianglesCount()
{
    return IndicesCount / 3;
}

int CTerrain::GetIndex(int X, int Z)
{
    return SizeP1 * Z + X;
}

float CTerrain::GetHeight(int X, int Z)
{
    return Heights[GetIndex(X < 0 ? 0 : X > Size ? Size : X, Z < 0 ? 0 : Z > Size ? Size : Z)];
}

float CTerrain::GetHeight(float X, float Z)
{
    Z = -Z;

    X += SizeD2;
    Z += SizeD2;

    float Size = (float)this->Size;

    if(X < 0.0f) X = 0.0f;
    if(X > Size) X = Size;
    if(Z < 0.0f) Z = 0.0f;
    if(Z > Size) Z = Size;

    int ix = (int)X, ixp1 = ix + 1;
    int iz = (int)Z, izp1 = iz + 1;

    float fx = X - (float)ix;
    float fz = Z - (float)iz;

    float a = GetHeight(ix, iz);
    float b = GetHeight(ixp1, iz);
    float c = GetHeight(ix, izp1);
    float d = GetHeight(ixp1, izp1);

    float ab = a + (b - a) * fx;
    float cd = c + (d - c) * fx;

    return ab + (cd - ab) * fz;
}

float CTerrain::GetHeight(float *Heights, int Size, float X, float Z)
{
    float SizeM1F = (float)Size - 1.0f;

    if(X < 0.0f) X = 0.0f;
    if(X > SizeM1F) X = SizeM1F;
    if(Z < 0.0f) Z = 0.0f;
    if(Z > SizeM1F) Z = SizeM1F;

    int ix = (int)X, ixp1 = ix + 1;
    int iz = (int)Z, izp1 = iz + 1;

    int SizeM1 = Size - 1;

    if(ixp1 > SizeM1) ixp1 = SizeM1;
    if(izp1 > SizeM1) izp1 = SizeM1;

    float fx = X - (float)ix;
    float fz = Z - (float)iz;

    int izMSize = iz * Size, izp1MSize = izp1 * Size;

    float a = Heights[izMSize + ix];
    float b = Heights[izMSize + ixp1];
    float c = Heights[izp1MSize + ix];
    float d = Heights[izp1MSize + ixp1];

    float ab = a + (b - a) * fx;
    float cd = c + (d - c) * fx;

    return ab + (cd - ab) * fz;
}

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

COpenGLRenderer::COpenGLRenderer()
{
    LightAngle = 22.5f;

    Wireframe = false;
    DisplayShadowMap = false;
    RenderAABB = false;
    VisualizeRenderingOrder = false;
    SortVisibleGeometryNodes = true;
    RenderSlow = false;

    Depth = -1;
}

COpenGLRenderer::~COpenGLRenderer()
{
}
```

```
bool COpenGLRenderer::Init()
{
	bool Error = false;

	if(!GLEW_EXT_framebuffer_object)
	{
		ErrorLog.Append("GL_EXT_framebuffer_object not supported!\r\n");
		Error = true;
	}

	Error |= !Shader.Load("glsl120shader.vs", "glsl120shader.fs");

	Error |= !Terrain.LoadBinary("terrain1.bin");

	if(Error)
	{
		return false;
	}

	Shader.UniformLocations = new GLuint[2];
	Shader.UniformLocations[0] = glGetUniformLocation(Shader, "ShadowMatrix");
	Shader.UniformLocations[1] = glGetUniformLocation(Shader, "LightDirection");

	glUseProgram(Shader);
	glUniform1i(glGetUniformLocation(Shader, "ShadowMap"), 0);
	glUniform1i(glGetUniformLocation(Shader, "RotationTexture"), 1);
	glUniform1f(glGetUniformLocation(Shader, "Scale"), 1.0f / 64.0f);
	glUniform1f(glGetUniformLocation(Shader, "Radius"), 1.0f / 1024.0f);
	glUseProgram(0);

	ShadowMapSize = SHADOW_MAP_SIZE > gl_max_texture_size ? gl_max_texture_size :
SHADOW_MAP_SIZE;

	glGenTextures(1, &ShadowMap);
	glBindTexture(GL_TEXTURE_2D, ShadowMap);
	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, ShadowMapSize,
ShadowMapSize, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
	glBindTexture(GL_TEXTURE_2D, 0);

	srand(GetTickCount());
```

```
    vec4 *RotationTextureData = new vec4[4096];

    float RandomAngle = 3.14f * 2.0f * (float)rand() / (float)RAND_MAX;

    for(int i = 0; i < 4096; i++)
    {
        RotationTextureData[i].x = cos(RandomAngle);
        RotationTextureData[i].y = sin(RandomAngle);
        RotationTextureData[i].z = -RotationTextureData[i].y;
        RotationTextureData[i].w = RotationTextureData[i].x;

        RotationTextureData[i] *= 0.5f;
        RotationTextureData[i] += 0.5f;

        RandomAngle += 3.14f * 2.0f * (float)rand() / (float)RAND_MAX;
    }

    glGenTextures(1, &RotationTexture);
    glBindTexture(GL_TEXTURE_2D, RotationTexture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 64, 64, 0, GL_RGBA, GL_FLOAT,
RotationTextureData);
    glBindTexture(GL_TEXTURE_2D, 0);

    delete [] RotationTextureData;

    glGenFramebuffersEXT(1, &FBO);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
    glDrawBuffers(0, NULL); glReadBuffer(GL_NONE);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
GL_TEXTURE_2D, ShadowMap, 0);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

    RenderShadowMap();

    float Height = Terrain.GetHeight(0.0f, 0.0f);

    Camera.Look(vec3(0.0f, Height + 1.75f, 0.0f), vec3(0.0f, Height + 1.75f, -1.0f));

    return true;
}
```

```
void COpenGLRenderer::Render()
{
    glViewport(0, 0, Width, Height);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(&Camera.ProjectionMatrix);

    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(&Camera.ViewMatrix);

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);

    if(!RenderSlow)
    {
        Terrain.CheckVisibility(Camera.Frustum, SortVisibleGeometryNodes);
    }

    if(Wireframe)
    {
        glColor3f(0.0f, 0.0f, 0.0f);

        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

        if(RenderSlow)
        {
            Terrain.RenderSlow();
        }
        else
        {
            Terrain.Render();
        }

        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    }

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

    glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, ShadowMap);
    glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, RotationTexture);

    glUseProgram(Shader);
```

```
if(RenderSlow)
{
	Terrain.RenderSlow();
}
else
{
	Terrain.Render(VisualizeRenderingOrder);
}

glUseProgram(0);

glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, 0);
glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, 0);

if(!RenderSlow && RenderAABB)
{
	glColor3f(0.0f, 1.0f, 0.0f);

	Terrain.RenderAABB(Depth);
}

glDisable(GL_CULL_FACE);
glDisable(GL_DEPTH_TEST);

if(DisplayShadowMap)
{
	glViewport(16, 16, 256, 256);

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

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

	glEnable(GL_TEXTURE_2D);

	glBindTexture(GL_TEXTURE_2D, ShadowMap);

	glBegin(GL_QUADS);
		glTexCoord2f(0.0f, 0.0f); glVertex2f(-1.0f, -1.0f);
		glTexCoord2f(1.0f, 0.0f); glVertex2f(1.0f, -1.0f);
```

```
            glTexCoord2f(1.0f, 1.0f); glVertex2f(1.0f, 1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex2f(-1.0f, 1.0f);
        glEnd();

        glBindTexture(GL_TEXTURE_2D, 0);

        glDisable(GL_TEXTURE_2D);
    }
}

void COpenGLRenderer::Animate(float FrameTime)
{
}

void COpenGLRenderer::Resize(int Width, int Height)
{
    this->Width = Width;
    this->Height = Height;

    Camera.SetPerspective(45.0f, (float)Width / (float)Height, 0.125f, 1024.0f);
}

void COpenGLRenderer::Destroy()
{
    Shader.Destroy();

    Terrain.Destroy();

    glDeleteTextures(1, &ShadowMap);
    glDeleteTextures(1, &RotationTexture);

    if(GLEW_EXT_framebuffer_object)
    {
        glDeleteFramebuffersEXT(1, &FBO);
    }
}

void COpenGLRenderer::RenderShadowMap()
{
    vec3 LightPosition = rotate(vec3((float)Terrain.GetSize(), 0.0f, 0.0f), -LightAngle,
vec3(0.0f, 1.0f, -1.0f));

    vec3 LightDirection = normalize(LightPosition);
```

```
	LightViewMatrix = look(LightPosition, vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 1.0f, 0.0f));

	vec3 Min, Max;

	Terrain.GetMinMax(LightViewMatrix, Min, Max);

	LightProjectionMatrix = ortho(Min.x, Max.x, Min.y, Max.y, -Max.z, -Min.z);

	ShadowMatrix = BiasMatrix * LightProjectionMatrix * LightViewMatrix;

	glUseProgram(Shader);
	glUniformMatrix4fv(Shader.UniformLocations[0], 1, GL_FALSE, &ShadowMatrix);
	glUniform3fv(Shader.UniformLocations[1], 1, &LightDirection);
	glUseProgram(0);

	glViewport(0, 0, ShadowMapSize, ShadowMapSize);

	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);

	glClear(GL_DEPTH_BUFFER_BIT);

	glMatrixMode(GL_PROJECTION);
	glLoadMatrixf(&LightProjectionMatrix);

	glMatrixMode(GL_MODELVIEW);
	glLoadMatrixf(&LightViewMatrix);

	glEnable(GL_DEPTH_TEST);

	Terrain.RenderSlowToShadowMap();

	glDisable(GL_DEPTH_TEST);

	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}

void COpenGLRenderer::CheckCameraTerrainPosition(vec3 &Movement)
{
	vec3 CameraPosition = Camera.Reference + Movement, Min = Terrain.GetMin(), Max =
Terrain.GetMax();

	if(CameraPosition.x < Min.x) Movement += vec3(Min.x - CameraPosition.x, 0.0f, 0.0f);
	if(CameraPosition.x > Max.x) Movement += vec3(Max.x - CameraPosition.x, 0.0f, 0.0f);
	if(CameraPosition.z < Min.z) Movement += vec3(0.0f, 0.0f, Min.z - CameraPosition.z);
```

```
    if(CameraPosition.z > Max.z) Movement += vec3(0.0f, 0.0f, Max.z - CameraPosition.z);

    CameraPosition = Camera.Reference + Movement;

    float Height = Terrain.GetHeight(CameraPosition.x, CameraPosition.z);

    Movement += vec3(0.0f, Height + 1.75f - Camera.Reference.y, 0.0f);
}

void COpenGLRenderer::CheckCameraKeys(float FrameTime)
{
    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)
    {
        vec3 Movement = Camera.OnKeys(Keys, FrameTime * 0.5f);

        CheckCameraTerrainPosition(Movement);

        Camera.Move(Movement);
    }
}

void COpenGLRenderer::OnKeyDown(UINT Key)
{
    switch(Key)
    {
        case VK_F1:
            Wireframe = !Wireframe;
            break;

        case VK_F2:
            DisplayShadowMap = !DisplayShadowMap;
            break;
        case VK_F3:
            RenderAABB = !RenderAABB;
            break;

        case VK_F4:
            VisualizeRenderingOrder = !VisualizeRenderingOrder;
            break;

        case VK_F5:
            SortVisibleGeometryNodes = !SortVisibleGeometryNodes;
            break;

        case VK_F6:
            RenderSlow = !RenderSlow;
            break;

        case VK_F7:
            Terrain.SaveBinary("terrain-saved.bin");
            break;

        case '1':
            if(Terrain.LoadBinary("terrain1.bin")) { vec3 Movement;
CheckCameraTerrainPosition(Movement); Camera.Move(Movement); RenderShadowMap(); }
            break;

        case '2':
            if(Terrain.LoadTexture2D("terrain2.jpg", 32.0f, -16.0f)) { vec3 Movement;
CheckCameraTerrainPosition(Movement); Camera.Move(Movement); RenderShadowMap(); }
            break;

        case '3':
            if(Terrain.LoadTexture2D("terrain3.jpg", 128.0f, -64.0f)) {    vec3 Movement;
CheckCameraTerrainPosition(Movement); Camera.Move(Movement); RenderShadowMap(); }
            break;

        case '4':
            if(Terrain.LoadTexture2D("terrain4.jpg", 128.0f, -64.0f)) { vec3 Movement;
CheckCameraTerrainPosition(Movement); Camera.Move(Movement); RenderShadowMap(); }
            break;

        case VK_MULTIPLY:
            Depth++;
            break;
        case VK_DIVIDE:
            if(Depth > -1) Depth--;
            break;

        case VK_ADD:
            LightAngle += 3.75f;
            RenderShadowMap();
            break;

        case VK_SUBTRACT:
            LightAngle -= 3.75f;
            RenderShadowMap();
            break;
    }
}

void COpenGLRenderer::OnLButtonDown(int X, int Y)
{
    LastClickedX = X;
    LastClickedY = Y;
}

void COpenGLRenderer::OnLButtonUp(int X, int Y)
{
    if(X == LastClickedX && Y == LastClickedY)
    {
    }
}

void COpenGLRenderer::OnMouseMove(int X, int Y)
{
    if(GetKeyState(VK_RBUTTON) & 0x80)
    {
        Camera.OnMouseMove(LastX - X, LastY - Y);
    }

    LastX = X;
    LastY = Y;
}

void COpenGLRenderer::OnMouseWheel(short zDelta)
{
    Camera.OnMouseWheel(zDelta);
```

```
}

void COpenGLRenderer::OnRButtonDown(int X, int Y)
{
	LastClickedX = X;
	LastClickedY = Y;
}

void COpenGLRenderer::OnRButtonUp(int X, int Y)
{
	if(X == LastClickedX && Y == LastClickedY)
	{
	}
}
```