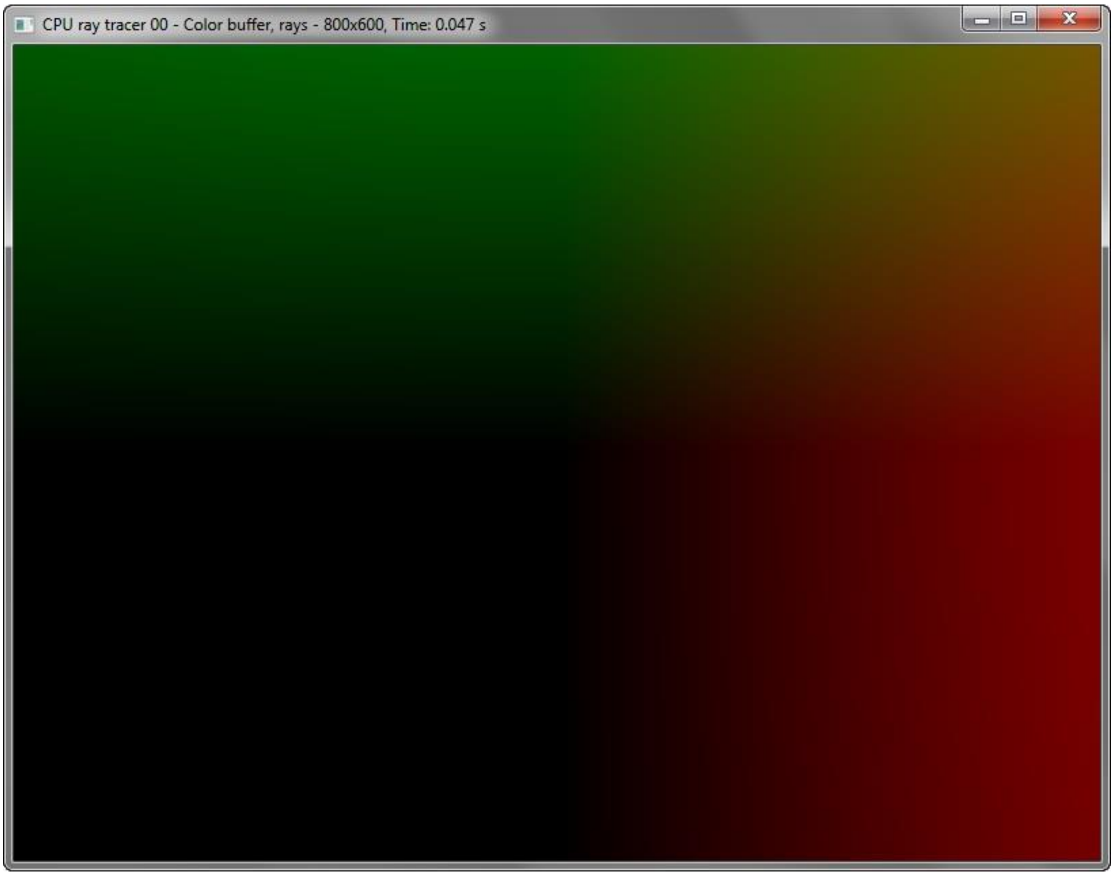
目录

第一章: CPU ray tracer 00 - Color buffer, rays


CPU ray tracer 00 - Color buffer, rays - 800x600, Time: 0.047 s

```
#include "glmath.h"

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

class CCamera
{
public:
	vec3 X, Y, Z, Reference, Position;
	mat4x4 Vin, Pin, VPin, RayMatrix;

public:
	CCamera();
	~CCamera();

	void CalculateRayMatrix();
	void Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference = false);
	bool OnKeyDown(UINT nChar);
	void OnMouseMove(int dx, int dy);
	void OnMouseWheel(short zDelta);
```

```
};

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

class CRayTracer
{
private:
	BYTE *ColorBuffer;
	BITMAPINFO ColorBufferInfo;
	int Width, LineWidth, Height;

public:
	CRayTracer();
	~CRayTracer();

	bool Init();
	void RayTrace(int x, int y);
	void Resize(int Width, int Height);
	void Destroy();

	void ClearColorBuffer();
	void SwapBuffers(HDC hDC);
};

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

class CWnd
{
protected:
	char *WindowName;
	HWND hWnd;
	int Width, Height, x, y, LastX, LastY;

public:
	CWnd();
	~CWnd();

	bool Create(HINSTANCE hInstance, char *WindowName, int Width, int Height);
	void RePaint();
	void Show(bool Maximized = false);
	void MsgLoop();
	void Destroy();
	void OnKeyDown(UINT Key);
	void OnMouseMove(int X, int Y);
	void OnMouseWheel(short zDelta);
	void OnPaint();
	void OnRButtonDown(int X, int Y);
	void OnSize(int Width, int Height);
};
//////////////////////////CPP////////////////////////////////////////////////////////
CCamera::CCamera()
{
	X = vec3(1.0, 0.0, 0.0);
	Y = vec3(0.0, 1.0, 0.0);
	Z = vec3(0.0, 0.0, 1.0);

	Reference = vec3(0.0, 0.0, 0.0);
	Position = vec3(0.0, 0.0, 5.0);
}

CCamera::~CCamera()
{
}

void CCamera::CalculateRayMatrix()
{
	Vin[0] = X.x; Vin[4] = Y.x; Vin[8] = Z.x;
	Vin[1] = X.y; Vin[5] = Y.y; Vin[9] = Z.y;
	Vin[2] = X.z; Vin[6] = Y.z; Vin[10] = Z.z;

	RayMatrix = Vin * Pin * BiasMatrixInverse * VPin;
}

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

	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;
	}

	CalculateRayMatrix();
}

bool CCamera::OnKeyDown(UINT nChar)
{
	float Distance = 0.125f;

	if(GetKeyState(VK_CONTROL) & 0x80) Distance *= 0.5f;
	if(GetKeyState(VK_SHIFT) & 0x80) Distance *= 2.0f;

	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(nChar == 'W') Movement += Forward;
	if(nChar == 'S') Movement -= Forward;
	if(nChar == 'A') Movement -= Right;
	if(nChar == 'D') Movement += Right;
	if(nChar == 'R') Movement += Up;
	if(nChar == 'F') Movement -= Up;

	Reference += Movement;
	Position += Movement;

	return Movement.x != 0.0f || Movement.y != 0.0f || Movement.z != 0.0f;
}

void CCamera::OnMouseMove(int dx, int dy)
{
	float sensitivity = 0.25f;

	float hangle = (float)dx * sensitivity;
	float vangle = (float)dy * sensitivity;
	Position -= Reference;

	Y = rotate(Y, vangle, X);
	Z = rotate(Z, vangle, X);

	if(Y.y < 0.0f)
	{
		Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);
		Y = cross(Z, X);
	}

	X = rotate(X, hangle, vec3(0.0f, 1.0f, 0.0f));
	Y = rotate(Y, hangle, vec3(0.0f, 1.0f, 0.0f));
	Z = rotate(Z, hangle, vec3(0.0f, 1.0f, 0.0f));

	Position = Reference + Z * length(Position);

	CalculateRayMatrix();
}

void CCamera::OnMouseWheel(short 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;
}

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

CCamera Camera;

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

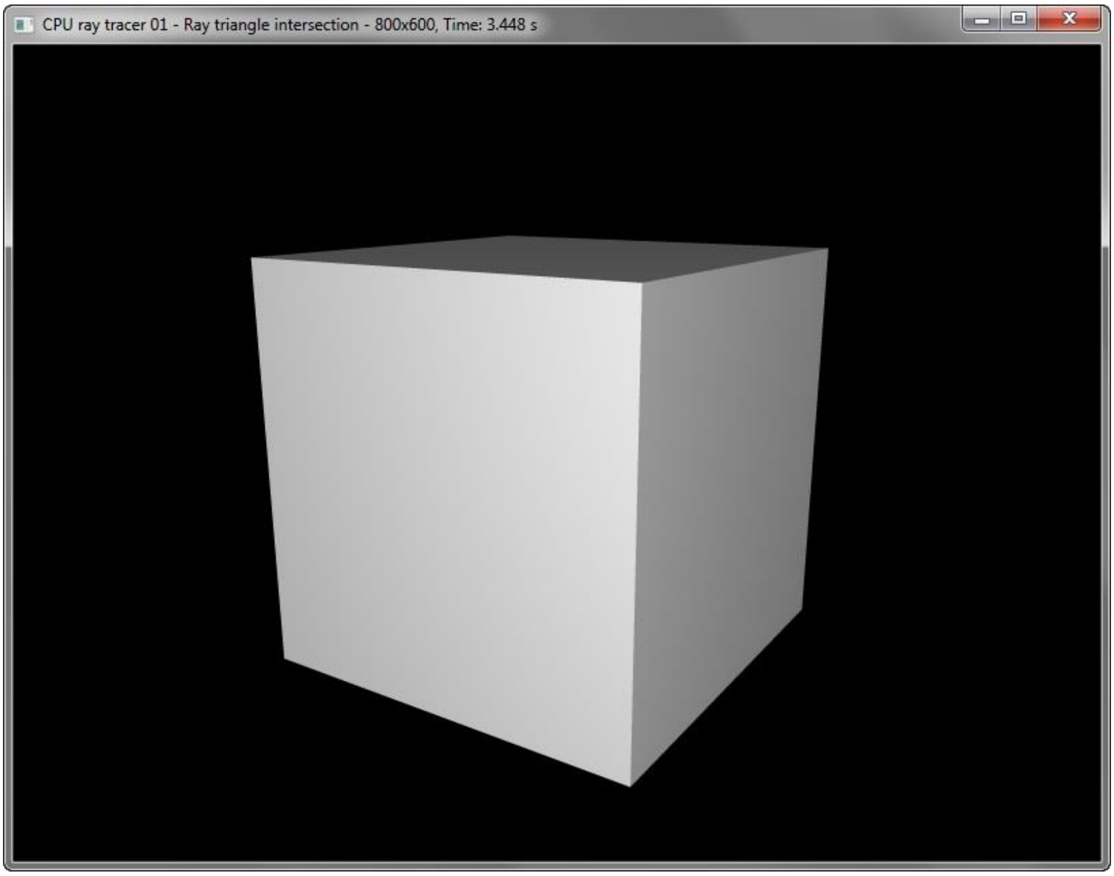
```
CRayTracer::CRayTracer()
{
	ColorBuffer = NULL;
}

CRayTracer::~CRayTracer()
{
}

bool CRayTracer::Init()
{
	return true;
}

void CRayTracer::RayTrace(int x, int y)
{
	if(ColorBuffer != NULL)
	{
		vec3 Ray = normalize(*(vec3*)&(Camera.RayMatrix * vec4((float)x + 0.5f, (float)y + 0.5f, 0.0f, 1.0f)));

		BYTE *colorbuffer = (LineWidth * y + x) * 3 + ColorBuffer;

		colorbuffer[2] = Ray.r <= 0.0f ? 0 : Ray.r >= 1.0 ? 255 : (BYTE)(Ray.r * 255);
		colorbuffer[1] = Ray.g <= 0.0f ? 0 : Ray.g >= 1.0 ? 255 : (BYTE)(Ray.g * 255);
		colorbuffer[0] = Ray.b <= 0.0f ? 0 : Ray.b >= 1.0 ? 255 : (BYTE)(Ray.b * 255);
	}
}

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

	if(ColorBuffer != NULL)
	{
		delete [] ColorBuffer;
		ColorBuffer = NULL;
	}

	if(Width > 0 && Height > 0)
	{
		LineWidth = Width;
		int WidthMod4 = Width % 4;

		if(WidthMod4 > 0)
		{
			LineWidth += 4 - WidthMod4;
		}

		ColorBuffer = new BYTE[LineWidth * Height * 3];

		memset(&ColorBufferInfo, 0, sizeof(BITMAPINFOHEADER));
		ColorBufferInfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
		ColorBufferInfo.bmiHeader.biPlanes = 1;
		ColorBufferInfo.bmiHeader.biBitCount = 24;
		ColorBufferInfo.bmiHeader.biCompression = BI_RGB;
		ColorBufferInfo.bmiHeader.biWidth = LineWidth;
		ColorBufferInfo.bmiHeader.biHeight = Height;

		Camera.VPin[0] = 1.0f / (float)Width;
		Camera.VPin[5] = 1.0f / (float)Height;

		float tany = tan(45.0f / 360.0f * (float)M_PI), aspect = (float)Width / (float)Height;

		Camera.Pin[0] = tany * aspect;
		Camera.Pin[5] = tany;
		Camera.Pin[10] = 0.0f;
		Camera.Pin[14] = -1.0f;

		Camera.CalculateRayMatrix();
	}
}

void CRayTracer::Destroy()
{
	if(ColorBuffer != NULL)
	{
		delete [] ColorBuffer;
		ColorBuffer = NULL;
	}
}

void CRayTracer::ClearColorBuffer()
{
	if(ColorBuffer != NULL)
	{
		memset(ColorBuffer, 0, LineWidth * Height * 3);
	}
}

void CRayTracer::SwapBuffers(HDC hDC)
{
	if(ColorBuffer != NULL)
	{
		StretchDIBits(hDC, 0, 0, Width, Height, 0, 0, Width, Height, ColorBuffer,
&ColorBufferInfo, DIB_RGB_COLORS, SRCCOPY);
	}
}

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

CRayTracer RayTracer;

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

CWnd::CWnd()
{
}

CWnd::~CWnd()
{
}

bool CWnd::Create(HINSTANCE hInstance, char *WindowName, int Width, int Height)
{
	WNDCLASSEX WndClassEx;

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

	WndClassEx.cbSize = sizeof(WNDCLASSEX);
	WndClassEx.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
	WndClassEx.lpfnWndProc = WndProc;
	WndClassEx.hInstance = hInstance;
	WndClassEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
	WndClassEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
	WndClassEx.hCursor = LoadCursor(NULL, IDC_ARROW);
	WndClassEx.lpszClassName = "Win32CPURayTracerWindow";
	if(RegisterClassEx(&WndClassEx) == 0)
	{
		ErrorLog.Set("RegisterClassEx failed!");
		return false;
	}

	this->WindowName = WindowName;

	this->Width = Width;
	this->Height = Height;

	DWORD Style = WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN;

	if((hWnd = CreateWindowEx(WS_EX_APPWINDOW, WndClassEx.lpszClassName,
WindowName, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL)) == NULL)
	{
		ErrorLog.Set("CreateWindowEx failed!");
		return false;
	}

	return RayTracer.Init();
}

void CWnd::RePaint()
{
	x = y = 0;
	InvalidateRect(hWnd, NULL, FALSE);
}

void CWnd::Show(bool Maximized)
{
	RECT dRect, wRect, cRect;

	GetWindowRect(GetDesktopWindow(), &dRect);
	GetWindowRect(hWnd, &wRect);
	GetClientRect(hWnd, &cRect);

	wRect.right += Width - cRect.right;
	wRect.bottom += Height - cRect.bottom;

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

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

	ShowWindow(hWnd, Maximized ? SW_SHOWMAXIMIZED : SW_SHOWNORMAL);
}

void CWnd::MsgLoop()
{
	MSG Msg;

	while(GetMessage(&Msg, NULL, 0, 0) > 0)
	{
		TranslateMessage(&Msg);
		DispatchMessage(&Msg);
	}
}

void CWnd::Destroy()
{
	RayTracer.Destroy();

	DestroyWindow(hWnd);
}

void CWnd::OnKeyDown(UINT Key)
{
	if(Camera.OnKeyDown(Key))
	{
		RePaint();
	}
}

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

		LastX = X;
		LastY = Y;

		RePaint();
```

```
    }
}

void CWnd::OnMouseWheel(short zDelta)
{
    Camera.OnMouseWheel(zDelta);

    RePaint();
}

void CWnd::OnPaint()
{
    PAINTSTRUCT ps;

    HDC hDC = BeginPaint(hWnd, &ps);

    static DWORD Start;
    static bool RayTracing;

    if(x == 0 && y == 0)
    {
        RayTracer.ClearColorBuffer();

        Start = GetTickCount();

        RayTracing = true;
    }

    DWORD start = GetTickCount();

    while(GetTickCount() - start < 125 && y < Height)
    {
        int x16 = x + 16, y16 = y + 16;

        for(int yy = y; yy < y16; yy++)
        {
            if(yy < Height)
            {
                for(int xx = x; xx < x16; xx++)
                {
                    if(xx < Width)
                    {
                        RayTracer.RayTrace(xx, yy);
                    }
                    else
                    {
                        break;
                    }
                }
            }
            else
            {
                break;
            }
        }

        x = x16;

        if(x >= Width)
        {
            x = 0;
            y = y16;
        }
    }

    RayTracer.SwapBuffers(hDC);

    if(RayTracing)
    {
        if(y >= Height)
        {
            RayTracing = false;
        }

        DWORD End = GetTickCount();

        CString text = WindowName;

        text.Append(" - %dx%d", Width, Height);
        text.Append(", Time: %.03f s", (float)(End - Start) * 0.001f);

        SetWindowText(hWnd, text);

        InvalidateRect(hWnd, NULL, FALSE);
    }

    EndPaint(hWnd, &ps);
}
```

```
void CWnd::OnRButtonDown(int X, int Y)
{
	LastX = X;
	LastY = Y;
}

void CWnd::OnSize(int Width, int Height)
{
	this->Width = Width;
	this->Height = Height;

	RayTracer.Resize(Width, Height);

	RePaint();
}

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

CWnd Wnd;

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

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
	switch(uiMsg)
	{
		case WM_CLOSE:
			PostQuitMessage(0);
			break;

		case WM_MOUSEMOVE:
			Wnd.OnMouseMove(LOWORD(lParam), HIWORD(lParam));
			break;

		case 0x020A: // WM_MOUSWHEEL
			Wnd.OnMouseWheel(HIWORD(wParam));
			break;

		case WM_KEYDOWN:
			Wnd.OnKeyDown((UINT)wParam);
```

```
            break;

        case WM_PAINT:
            Wnd.OnPaint();
            break;

        case WM_RBUTTONDOWN:
            Wnd.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));
            break;

        case WM_SIZE:
            Wnd.OnSize(LOWORD(lParam), HIWORD(lParam));
            break;

        default:
            return DefWindowProc(hWnd, uiMsg, wParam, lParam);
    }

    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine,
int iShow)
{
    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);

    if(Wnd.Create(hInstance, "CPU ray tracer 00 - Color buffer, rays", 800, 600))
    {
        Wnd.Show();
        Wnd.MsgLoop();
    }
    else
    {
        MessageBox(NULL, ErrorLog, "Error", MB_OK | MB_ICONERROR);
    }

    Wnd.Destroy();

    return 0;
}
```

第二章: CPU ray tracer 01 - Ray triangle intersection


CPU ray tracer 01 - Ray triangle intersection - 800x600, Time: 3.448 s

```
class CCamera
{
public:
	vec3 X, Y, Z, Reference, Position;
	mat4x4 Vin, Pin, VPin, RayMatrix;

public:
	CCamera();
	~CCamera();

	void CalculateRayMatrix();
	void Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference 
= false);
	bool OnKeyDown(UINT nChar);
	void OnMouseMove(int dx, int dy);
	void OnMouseWheel(short zDelta);
};

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

```
class CTriangle
{
public:
	float D, D1, D2, D3;
	vec3 a, b, c, Color, N, N1, N2, N3;

public:
	CTriangle();
	CTriangle(const vec3 &a, const vec3 &b, const vec3 &c, const vec3 &Color);

	bool Inside(const vec3 &Point);
	bool Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance,
vec3 &Point);
	bool Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance, float
&Distance);
	bool Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance);
};

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

class RTData
{
public:
	float Distance, TestDistance;
	vec3 Color, Point, TestPoint;
	CTriangle *Triangle;

public:
	RTData();
};

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

class CRayTracer
{
private:
	BYTE *ColorBuffer;
	BITMAPINFO ColorBufferInfo;
	int Width, LineWidth, Height;

protected:
	CTriangle *Triangles, *LastTriangle;
	int TrianglesCount;

public:
	bool SuperSampling;

public:
	CRayTracer();
	~CRayTracer();

	bool Init();
	void RayTrace(int x, int y);
	void Resize(int Width, int Height);
	void Destroy();

	void ClearColorBuffer();
	void SwapBuffers(HDC hDC);

protected:
	vec3 RayTrace(const vec3 &Origin, const vec3 &Ray);

protected:
	virtual bool InitScene() = 0;
};

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

class CMyRayTracer : public CRayTracer
{
protected:
	bool InitScene();
};

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

class CWnd
{
protected:
	char *WindowName;
	HWND hWnd;
	int Width, Height, x, y, LastX, LastY;

public:
    CWnd();
    ~CWnd();

    bool Create(HINSTANCE hInstance, char *WindowName, int Width, int Height);
    void RePaint();
    void Show(bool Maximized = false);
    void MsgLoop();
    void Destroy();

    void OnKeyDown(UINT Key);
    void OnMouseMove(int X, int Y);
    void OnMouseWheel(short zDelta);
    void OnPaint();
    void OnRButtonDown(int X, int Y);
    void OnSize(int Width, int Height);
};
```

```
///////////////////////////////////CPP///////////////////////////////////

CCamera::CCamera()
{
	X = vec3(1.0, 0.0, 0.0);
	Y = vec3(0.0, 1.0, 0.0);
	Z = vec3(0.0, 0.0, 1.0);

	Reference = vec3(0.0, 0.0, 0.0);
	Position = vec3(0.0, 0.0, 5.0);
}

CCamera::~CCamera()
{
}

void CCamera::CalculateRayMatrix()
{
	Vin[0] = X.x; Vin[4] = Y.x; Vin[8] = Z.x;
	Vin[1] = X.y; Vin[5] = Y.y; Vin[9] = Z.y;
	Vin[2] = X.z; Vin[6] = Y.z; Vin[10] = Z.z;

	RayMatrix = Vin * Pin * BiasMatrixInverse * VPin;
}

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

	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;
	}

	CalculateRayMatrix();
}
```

```
bool CCamera::OnKeyDown(UINT nChar)
{
	float Distance = 0.125f;

	if(GetKeyState(VK_CONTROL) & 0x80) Distance *= 0.5f;
	if(GetKeyState(VK_SHIFT) & 0x80) Distance *= 2.0f;

	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(nChar == 'W') Movement += Forward;
	if(nChar == 'S') Movement -= Forward;
	if(nChar == 'A') Movement -= Right;
	if(nChar == 'D') Movement += Right;
	if(nChar == 'R') Movement += Up;
	if(nChar == 'F') Movement -= Up;

	Reference += Movement;
	Position += Movement;

	return Movement.x != 0.0f || Movement.y != 0.0f || Movement.z != 0.0f;
}

void CCamera::OnMouseMove(int dx, int dy)
{
	float sensitivity = 0.25f;

	float hangle = (float)dx * sensitivity;
	float vangle = (float)dy * sensitivity;

	Position -= Reference;

	Y = rotate(Y, vangle, X);
	Z = rotate(Z, vangle, X);

	if(Y.y < 0.0f)
	{
		Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);
		Y = cross(Z, X);
	}

	X = rotate(X, hangle, vec3(0.0f, 1.0f, 0.0f));
	Y = rotate(Y, hangle, vec3(0.0f, 1.0f, 0.0f));
	Z = rotate(Z, hangle, vec3(0.0f, 1.0f, 0.0f));

	Position = Reference + Z * length(Position);

	CalculateRayMatrix();
}

void CCamera::OnMouseWheel(short 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;
}

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

CCamera Camera;

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

CTriangle::CTriangle()
{
}

CTriangle::CTriangle(const vec3 &a, const vec3 &b, const vec3 &c, const vec3 &Color) : a(a),
b(b), c(c), Color(Color)
{
    N = normalize(cross(b - a, c - a));
    D = -dot(N, a);

    N1 = normalize(cross(N, b - a));
    D1 = -dot(N1, a);

    N2 = normalize(cross(N, c - b));
    D2 = -dot(N2, b);

    N3 = normalize(cross(N, a - c));
    D3 = -dot(N3, c);
}

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::Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance, float
&Distance, vec3 &Point)
{
    float NdotR = -dot(N, Ray);

    if(NdotR > 0.0f)
    {
        Distance = (dot(N, Origin) + D) / NdotR;

        if(Distance >= 0.0f && Distance < MaxDistance)
        {
            Point = Ray * Distance + Origin;

            return Inside(Point);
        }
    }

    return false;
}
```

```
bool CTriangle::Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance, float
&Distance)
{
	float NdotR = -dot(N, Ray);

	if(NdotR > 0.0f)
	{
		Distance = (dot(N, Origin) + D) / NdotR;

		if(Distance >= 0.0f && Distance < MaxDistance)
		{
			return Inside(Ray * Distance + Origin);
		}
	}

	return false;
}

bool CTriangle::Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance)
{
	float NdotR = -dot(N, Ray);

	if(NdotR > 0.0f)
	{
		float Distance = (dot(N, Origin) + D) / NdotR;

		if(Distance >= 0.0f && Distance < MaxDistance)
		{
			return Inside(Ray * Distance + Origin);
		}
	}

	return false;
}

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

RTData::RTData()
{
	Distance = 1048576.0f;
	Triangle = NULL;
}
```

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

CRayTracer::CRayTracer()
{
    ColorBuffer = NULL;

    Triangles = NULL;
    LastTriangle = NULL;
    TrianglesCount = 0;

    SuperSampling = false;
}

CRayTracer::~CRayTracer()
{
}

bool CRayTracer::Init()
{
    if(InitScene() == false)
    {
        return false;
    }

    LastTriangle = Triangles + TrianglesCount;

    return true;
}

void CRayTracer::RayTrace(int x, int y)
{
    if(ColorBuffer != NULL && Triangles != NULL && TrianglesCount > 0)
    {
        vec3 Color;

        if(!SuperSampling)
        {
            Color = RayTrace(Camera.Position, normalize(*(vec3*)&(Camera.RayMatrix *
vec4((float)x + 0.5f, (float)y + 0.5f, 0.0f, 1.0f))));
        }
        else
        {
            for(float yy = 0.125f; yy < 1.0f; yy += 0.25f)
```

```
                {
                    for(float xx = 0.125f; xx < 1.0f; xx += 0.25f)
                    {
                        Color += RayTrace(Camera.Position,
normalize(*(vec3*)&(Camera.RayMatrix * vec4((float)x + xx, (float)y + yy, 0.0f, 1.0f))));
                    }
                }

                Color /= 16.0f;
            }

            BYTE *colorbuffer = (LineWidth * y + x) * 3 + ColorBuffer;

            colorbuffer[2] = Color.r <= 0.0f ? 0 : Color.r >= 1.0 ? 255 : (BYTE)(Color.r * 255);
            colorbuffer[1] = Color.g <= 0.0f ? 0 : Color.g >= 1.0 ? 255 : (BYTE)(Color.g * 255);
            colorbuffer[0] = Color.b <= 0.0f ? 0 : Color.b >= 1.0 ? 255 : (BYTE)(Color.b * 255);
        }
}

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

    if(ColorBuffer != NULL)
    {
        delete [] ColorBuffer;
        ColorBuffer = NULL;
    }

    if(Width > 0 && Height > 0)
    {
        LineWidth = Width;

        int WidthMod4 = Width % 4;

        if(WidthMod4 > 0)
        {
            LineWidth += 4 - WidthMod4;
        }

        ColorBuffer = new BYTE[LineWidth * Height * 3];

        memset(&ColorBufferInfo, 0, sizeof(BITMAPINFOHEADER));
        ColorBufferInfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
        ColorBufferInfo.bmiHeader.biPlanes = 1;
        ColorBufferInfo.bmiHeader.biBitCount = 24;
        ColorBufferInfo.bmiHeader.biCompression = BI_RGB;
        ColorBufferInfo.bmiHeader.biWidth = LineWidth;
        ColorBufferInfo.bmiHeader.biHeight = Height;

        Camera.VPin[0] = 1.0f / (float)Width;
        Camera.VPin[5] = 1.0f / (float)Height;

        float tany = tan(45.0f / 360.0f * (float)M_PI), aspect = (float)Width / (float)Height;

        Camera.Pin[0] = tany * aspect;
        Camera.Pin[5] = tany;
        Camera.Pin[10] = 0.0f;
        Camera.Pin[14] = -1.0f;

        Camera.CalculateRayMatrix();
    }
}

void CRayTracer::Destroy()
{
    if(Triangles != NULL)
    {
        delete [] Triangles;
        Triangles = NULL;
        LastTriangle = NULL;
        TrianglesCount = 0;
    }

    if(ColorBuffer != NULL)
    {
        delete [] ColorBuffer;
        ColorBuffer = NULL;
    }
}

void CRayTracer::ClearColorBuffer()
{
    if(ColorBuffer != NULL)
    {
        memset(ColorBuffer, 0, LineWidth * Height * 3);
    }
```

```
}

void CRayTracer::SwapBuffers(HDC hDC)
{
    if(ColorBuffer != NULL)
    {
        StretchDIBits(hDC, 0, 0, Width, Height, 0, 0, Width, Height, ColorBuffer,
&ColorBufferInfo, DIB_RGB_COLORS, SRCCOPY);
    }
}

vec3 CRayTracer::RayTrace(const vec3 &Origin, const vec3 &Ray)
{
    RTData rtdata;

    for(CTriangle *Triangle = Triangles; Triangle < LastTriangle; Triangle++)
    {
        if(Triangle->Intersect(Origin, Ray, rtdata.Distance, rtdata.TestDistance,
rtdata.TestPoint))
        {
            rtdata.Point = rtdata.TestPoint;
            rtdata.Distance = rtdata.TestDistance;
            rtdata.Triangle = Triangle;
        }
    }

    if(rtdata.Triangle)
    {
        rtdata.Color = rtdata.Triangle->Color;

        float NdotL = -dot(rtdata.Triangle->N, Ray);

        if(NdotL < 0.0f)
        {
            NdotL = 0.0f;
        }

        rtdata.Color *= 0.75f * NdotL + 0.25f;
    }

    return rtdata.Color;
}

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

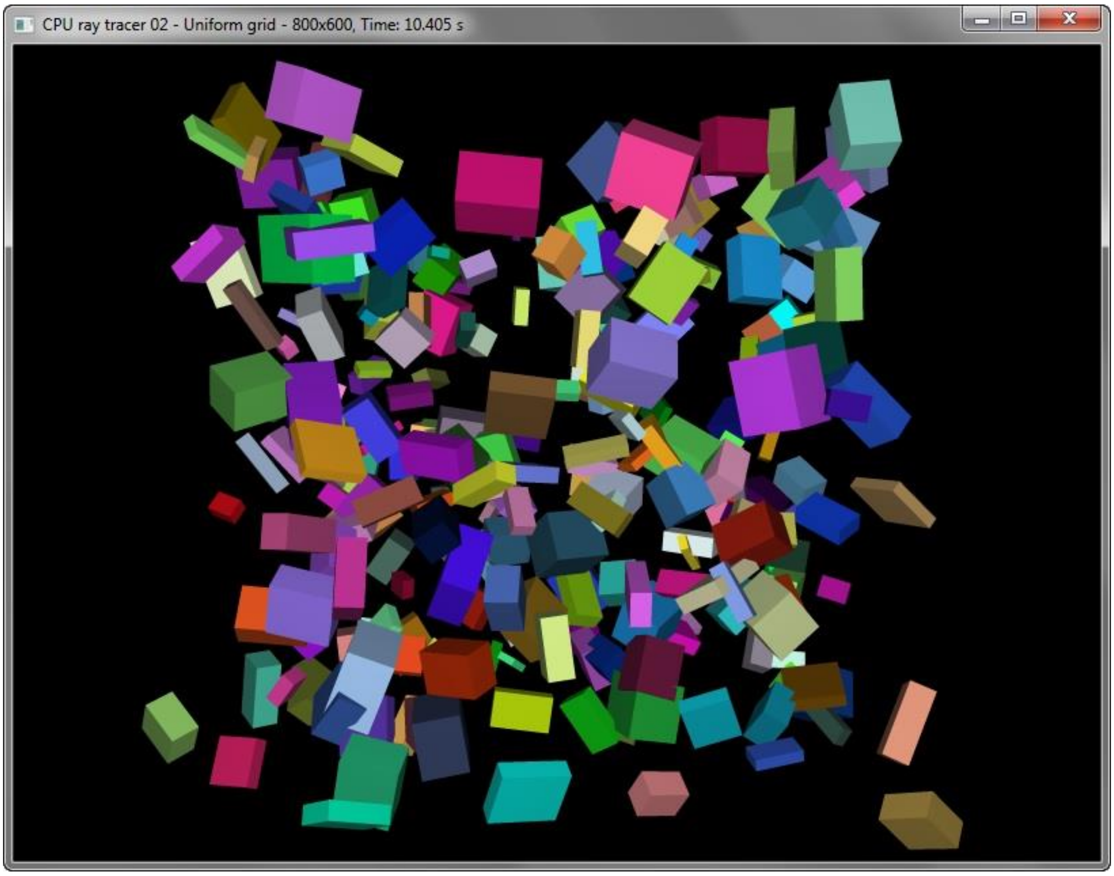
```
-----------------------------------------------

bool CMyRayTracer::InitScene()
{
	bool Error = false;

	if(Error)
	{
		return false;
	}

	TrianglesCount = 12;

	Triangles = new CTriangle[TrianglesCount];

	int t = 0;

	Triangles[t++] = CTriangle(vec3( 0.5f, -0.5f,  0.5f), vec3( 0.5f, -0.5f, -0.5f), vec3( 0.5f,  0.5f, -0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3( 0.5f,  0.5f, -0.5f), vec3( 0.5f,  0.5f,  0.5f), vec3( 0.5f, -0.5f,  0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3(-0.5f, -0.5f, -0.5f), vec3(-0.5f, -0.5f,  0.5f), vec3(-0.5f,  0.5f,  0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3(-0.5f,  0.5f,  0.5f), vec3(-0.5f,  0.5f, -0.5f), vec3(-0.5f, -0.5f, -0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3(-0.5f,  0.5f,  0.5f), vec3( 0.5f,  0.5f,  0.5f), vec3( 0.5f,  0.5f, -0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3( 0.5f,  0.5f, -0.5f), vec3(-0.5f,  0.5f, -0.5f), vec3(-0.5f,  0.5f,  0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3(-0.5f, -0.5f, -0.5f), vec3( 0.5f, -0.5f, -0.5f), vec3( 0.5f, -0.5f,  0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3( 0.5f, -0.5f,  0.5f), vec3(-0.5f, -0.5f,  0.5f), vec3(-0.5f, -0.5f, -0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3(-0.5f, -0.5f,  0.5f), vec3( 0.5f, -0.5f,  0.5f), vec3( 0.5f,  0.5f,  0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3( 0.5f,  0.5f,  0.5f), vec3(-0.5f,  0.5f,  0.5f), vec3(-0.5f, -0.5f,  0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3( 0.5f, -0.5f, -0.5f), vec3(-0.5f, -0.5f, -0.5f), vec3(-0.5f,  0.5f, -0.5f), vec3(1.0f, 1.0f, 1.0f));
	Triangles[t++] = CTriangle(vec3(-0.5f,  0.5f, -0.5f), vec3( 0.5f,  0.5f, -0.5f), vec3( 0.5f, -0.5f, -0.5f), vec3(1.0f, 1.0f, 1.0f));

	Camera.Look(vec3(0.0f, 0.0f, 2.5f), vec3(0.0f, 0.0f, 0.0f), true);
	return true;
}

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

CMyRayTracer RayTracer;

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

CWnd::CWnd()
{
}

CWnd::~CWnd()
{
}

bool CWnd::Create(HINSTANCE hInstance, char *WindowName, int Width, int Height)
{
	WNDCLASSEX WndClassEx;

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

	WndClassEx.cbSize = sizeof(WNDCLASSEX);
	WndClassEx.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
	WndClassEx.lpfnWndProc = WndProc;
	WndClassEx.hInstance = hInstance;
	WndClassEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
	WndClassEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
	WndClassEx.hCursor = LoadCursor(NULL, IDC_ARROW);
	WndClassEx.lpszClassName = "Win32CPURayTracerWindow";

	if(RegisterClassEx(&WndClassEx) == 0)
	{
		ErrorLog.Set("RegisterClassEx failed!");
		return false;
	}

	this->WindowName = WindowName;

	this->Width = Width;
	this->Height = Height;
	DWORD Style = WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN;

	if((hWnd = CreateWindowEx(WS_EX_APPWINDOW, WndClassEx.lpszClassName, WindowName, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL)) == NULL)
	{
		ErrorLog.Set("CreateWindowEx failed!");
		return false;
	}

	return RayTracer.Init();
}

void CWnd::RePaint()
{
	x = y = 0;
	InvalidateRect(hWnd, NULL, FALSE);
}

void CWnd::Show(bool Maximized)
{
	RECT dRect, wRect, cRect;

	GetWindowRect(GetDesktopWindow(), &dRect);
	GetWindowRect(hWnd, &wRect);
	GetClientRect(hWnd, &cRect);

	wRect.right += Width - cRect.right;
	wRect.bottom += Height - cRect.bottom;

	wRect.right -= wRect.left;
	wRect.bottom -= wRect.top;

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

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

	ShowWindow(hWnd, Maximized ? SW_SHOWMAXIMIZED : SW_SHOWNORMAL);
}

void CWnd::MsgLoop()
{
	MSG Msg;
```

```
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
}

void CWnd::Destroy()
{
    RayTracer.Destroy();

    DestroyWindow(hWnd);
}

void CWnd::OnKeyDown(UINT Key)
{
    switch(Key)
    {
        case VK_F1:
            RayTracer.SuperSampling = !RayTracer.SuperSampling;
            RePaint();
            break;
    }

    if(Camera.OnKeyDown(Key))
    {
        RePaint();
    }
}

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

        LastX = X;
        LastY = Y;

        RePaint();
    }
}
```

```
void CWnd::OnMouseWheel(short zDelta)
{
    Camera.OnMouseWheel(zDelta);

    RePaint();
}

void CWnd::OnPaint()
{
    PAINTSTRUCT ps;

    HDC hDC = BeginPaint(hWnd, &ps);

    static DWORD Start;
    static bool RayTracing;

    if(x == 0 && y == 0)
    {
        RayTracer.ClearColorBuffer();

        Start = GetTickCount();

        RayTracing = true;
    }

    DWORD start = GetTickCount();

    while(GetTickCount() - start < 125 && y < Height)
    {
        int x16 = x + 16, y16 = y + 16;

        for(int yy = y; yy < y16; yy++)
        {
            if(yy < Height)
            {
                for(int xx = x; xx < x16; xx++)
                {
                    if(xx < Width)
                    {
                        RayTracer.RayTrace(xx, yy);
                    }
                    else
                    {
                        break;
                    }
                }
            }
            else
            {
                break;
            }
        }

        x = x16;

        if(x >= Width)
        {
            x = 0;
            y = y16;
        }
    }

    RayTracer.SwapBuffers(hDC);

    if(RayTracing)
    {
        if(y >= Height)
        {
            RayTracing = false;
        }

        DWORD End = GetTickCount();

        CString text = WindowName;

        text.Append(" - %dx%d", Width, Height);
        text.Append(", Time: %.03f s", (float)(End - Start) * 0.001f);

        SetWindowText(hWnd, text);

        InvalidateRect(hWnd, NULL, FALSE);
    }

    EndPaint(hWnd, &ps);
}

void CWnd::OnRButtonDown(int X, int Y)
{
        LastX = X;
        LastY = Y;
}

void CWnd::OnSize(int Width, int Height)
{
        this->Width = Width;
        this->Height = Height;

        RayTracer.Resize(Width, Height);

        RePaint();
}

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

CWnd Wnd;

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

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM
lParam)
{
        switch(uiMsg)
        {
                case WM_CLOSE:
                        PostQuitMessage(0);
                        break;

                case WM_MOUSEMOVE:
                        Wnd.OnMouseMove(LOWORD(lParam), HIWORD(lParam));
                        break;

                case 0x020A: // WM_MOUSWHEEL
                        Wnd.OnMouseWheel(HIWORD(wParam));
                        break;

                case WM_KEYDOWN:
                        Wnd.OnKeyDown((UINT)wParam);
                        break;

                case WM_PAINT:
```

```
            Wnd.OnPaint();
            break;

        case WM_RBUTTONDOWN:
            Wnd.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));
            break;

        case WM_SIZE:
            Wnd.OnSize(LOWORD(lParam), HIWORD(lParam));
            break;

        default:
            return DefWindowProc(hWnd, uiMsg, wParam, lParam);
    }

    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine,
int iShow)
{
    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);

    if(Wnd.Create(hInstance, "CPU ray tracer 01 - Ray triangle intersection", 800, 600))
    {
        Wnd.Show();
        Wnd.MsgLoop();
    }
    else
    {
        MessageBox(NULL, ErrorLog, "Error", MB_OK | MB_ICONERROR);
    }

    Wnd.Destroy();

    return 0;
}
```

第三章: CPU ray tracer 02 - Uniform grid



```
class CCamera
{
public:
	vec3 X, Y, Z, Reference, Position;
	mat4x4 Vin, Pin, VPin, RayMatrix;

public:
	CCamera();
	~CCamera();

	void CalculateRayMatrix();
	void Look(const vec3 &Position, const vec3 &Reference, bool RotateAroundReference
= false);
	bool OnKeyDown(UINT nChar);
	void OnMouseMove(int dx, int dy);
	void OnMouseWheel(short zDelta);
};

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

```
class CTriangle
{
public:
	float D, D1, D2, D3, lab, lbc, lca;
	vec3 a, b, c, ab, bc, ca, Color, N, N1, N2, N3;

public:
	CTriangle();
	CTriangle(const vec3 &a, const vec3 &b, const vec3 &c, const vec3 &Color);

	bool Inside(float x, float y, float z);
	bool Inside(const vec3 &Point);
	bool Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance, vec3 &Point);
	bool Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance);
	bool Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance);
};

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

class RTData
{
public:
	float Distance, TestDistance;
	vec3 Color, Point, TestPoint;
	CTriangle *Triangle;

public:
	RTData();
};

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

class CVoxel
{
public:
	CTriangle **Triangles;
	int TrianglesCount;

protected:
	int MaxTrianglesCount;
	float Size;
	vec3 Min, Max, MinE, MaxE;

public:
	CVoxel();
	~CVoxel();

	void Add(CTriangle *Triangle);
	void Delete();
	bool Inside(const vec3 &Point);
	bool Intersect(CTriangle *Triangle);
	bool IntersectEdgesX(CTriangle *Triangle, float x, float y1, float y2, float z1, float z2);
	bool IntersectEdgesY(CTriangle *Triangle, float y, float x1, float x2, float z1, float z2);
	bool IntersectEdgesZ(CTriangle *Triangle, float z, float x1, float x2, float y1, float y2);
	bool IntersectFacesX(CTriangle *Triangle, float D1, float D2);
	bool IntersectFacesY(CTriangle *Triangle, float D1, float D2);
	bool IntersectFacesZ(CTriangle *Triangle, float D1, float D2);
	void Set(const vec3 &Min, float Size);
};

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

class CUniformGrid
{
protected:
	vec3 Min, Max;

protected:
	int X, Y, Z, Xm1, Ym1, Zm1, XY, XYZ;
	float VoxelSize;
	CVoxel *Voxels;

public:
	CUniformGrid();
	~CUniformGrid();

public:
	void Delete();
	void Generate(CTriangle *Triangles, int TrianglesCount, float VoxelSize = 1.0f);
	vec3 Traverse(const vec3 &Voxel, const vec3 &Origin, const vec3 &Ray);

	float VoxelToWorldX(float x);
	float VoxelToWorldY(float y);
	float VoxelToWorldZ(float z);
	vec3 VoxelToWorld(const vec3 &Voxel);
	int WorldToVoxelX(float x);
	int WorldToVoxelY(float y);
	int WorldToVoxelZ(float z);
	vec3 WorldToVoxel(const vec3 &World);
};

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

class CRayTracer
{
private:
	BYTE *ColorBuffer;
	BITMAPINFO ColorBufferInfo;
	int Width, LineWidth, Height;

protected:
	CTriangle *Triangles;
	int TrianglesCount;

private:
	CUniformGrid UniformGrid;

public:
	bool SuperSampling;

public:
	CRayTracer();
	~CRayTracer();

	bool Init();
	void RayTrace(int x, int y);
	void Resize(int Width, int Height);
	void Destroy();

	void ClearColorBuffer();
	void SwapBuffers(HDC hDC);

protected:
	virtual bool InitScene() = 0;
};
```

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

class CMyRayTracer : public CRayTracer
{
protected:
        bool InitScene();
};

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

class CWnd
{
protected:
        char *WindowName;
        HWND hWnd;
        int Width, Height, x, y, LastX, LastY;

public:
        CWnd();
        ~CWnd();

        bool Create(HINSTANCE hInstance, char *WindowName, int Width, int Height);
        void RePaint();
        void Show(bool Maximized = false);
        void MsgLoop();
        void Destroy();

        void OnKeyDown(UINT Key);
        void OnMouseMove(int X, int Y);
        void OnMouseWheel(short zDelta);
        void OnPaint();
        void OnRButtonDown(int X, int Y);
        void OnSize(int Width, int Height);
};
//////////////////////////////////CPP///////////////////////////////////////
CCamera::CCamera()
{
        X = vec3(1.0, 0.0, 0.0);
        Y = vec3(0.0, 1.0, 0.0);
        Z = vec3(0.0, 0.0, 1.0);

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

```
	Position = vec3(0.0, 0.0, 5.0);
}

CCamera::~CCamera()
{
}

void CCamera::CalculateRayMatrix()
{
	Vin[0] = X.x; Vin[4] = Y.x; Vin[8] = Z.x;
	Vin[1] = X.y; Vin[5] = Y.y; Vin[9] = Z.y;
	Vin[2] = X.z; Vin[6] = Y.z; Vin[10] = Z.z;

	RayMatrix = Vin * Pin * BiasMatrixInverse * VPin;
}

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

	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;
	}

	CalculateRayMatrix();
}

bool CCamera::OnKeyDown(UINT nChar)
{
	float Distance = 0.125f;

	if(GetKeyState(VK_CONTROL) & 0x80) Distance *= 0.5f;
	if(GetKeyState(VK_SHIFT) & 0x80) Distance *= 2.0f;

	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(nChar == 'W') Movement += Forward;
	if(nChar == 'S') Movement -= Forward;
	if(nChar == 'A') Movement -= Right;
	if(nChar == 'D') Movement += Right;
	if(nChar == 'R') Movement += Up;
	if(nChar == 'F') Movement -= Up;

	Reference += Movement;
	Position += Movement;

	return Movement.x != 0.0f || Movement.y != 0.0f || Movement.z != 0.0f;
}

void CCamera::OnMouseMove(int dx, int dy)
{
	float sensitivity = 0.25f;

	float hangle = (float)dx * sensitivity;
	float vangle = (float)dy * sensitivity;

	Position -= Reference;

	Y = rotate(Y, vangle, X);
	Z = rotate(Z, vangle, X);

	if(Y.y < 0.0f)
	{
		Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);
		Y = cross(Z, X);
	}

	X = rotate(X, hangle, vec3(0.0f, 1.0f, 0.0f));
	Y = rotate(Y, hangle, vec3(0.0f, 1.0f, 0.0f));
	Z = rotate(Z, hangle, vec3(0.0f, 1.0f, 0.0f));

	Position = Reference + Z * length(Position);
	CalculateRayMatrix();
}

void CCamera::OnMouseWheel(short 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;
}

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

CCamera Camera;

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

CTriangle::CTriangle()
{
}

CTriangle::CTriangle(const vec3 &a, const vec3 &b, const vec3 &c, const vec3 &Color) : a(a),
b(b), c(c), Color(Color)
{
	ab = b - a; bc = c - b; ca = a - c;

	N = normalize(cross(ab, -ca));
	D = -dot(N, a);

	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);

    lab = length(ab); ab /= lab;
    lbc = length(bc); bc /= lbc;
    lca = length(ca); ca /= lca;
}

bool CTriangle::Inside(float x, float y, float z)
{
    if(N1.x * x + N1.y * y + N1.z * z + D1 < 0.0f) return false;
    if(N2.x * x + N2.y * y + N2.z * z + D2 < 0.0f) return false;
    if(N3.x * x + N3.y * y + N3.z * z + D3 < 0.0f) return false;

    return true;
}

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::Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance, float
&Distance, vec3 &Point)
{
    float NdotR = -dot(N, Ray);

    if(NdotR > 0.0f)
    {
        Distance = (dot(N, Origin) + D) / NdotR;

        if(Distance >= 0.0f && Distance < MaxDistance)
        {
            Point = Ray * Distance + Origin;

            return Inside(Point);
        }
    }

    return false;
}

bool CTriangle::Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance, float
&Distance)
{
    float NdotR = -dot(N, Ray);

    if(NdotR > 0.0f)
    {
        Distance = (dot(N, Origin) + D) / NdotR;

        if(Distance >= 0.0f && Distance < MaxDistance)
        {
            return Inside(Ray * Distance + Origin);
        }
    }

    return false;
}

bool CTriangle::Intersect(const vec3 &Origin, const vec3 &Ray, float MaxDistance)
{
    float NdotR = -dot(N, Ray);

    if(NdotR > 0.0f)
    {
        float Distance = (dot(N, Origin) + D) / NdotR;

        if(Distance >= 0.0f && Distance < MaxDistance)
        {
            return Inside(Ray * Distance + Origin);
        }
    }

    return false;
}

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

RTData::RTData()
```

```
{
	Distance = 1048576.0f;
	Triangle = NULL;
}

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

CVoxel::CVoxel()
{
	Triangles = NULL;
	TrianglesCount = 0;
	MaxTrianglesCount = 1;
	Size = 0.0f;
}

CVoxel::~CVoxel()
{
}

void CVoxel::Add(CTriangle *Triangle)
{
	if(TrianglesCount % MaxTrianglesCount == 0)
	{
		CTriangle **OldTriangles = Triangles;

		MaxTrianglesCount *= 2;

		Triangles = new CTriangle*[MaxTrianglesCount];

		for(int i = 0; i < TrianglesCount; i++)
		{
			Triangles[i] = OldTriangles[i];
		}

		if(OldTriangles != NULL)
		{
			delete [] OldTriangles;
		}
	}

	Triangles[TrianglesCount] = Triangle;

	TrianglesCount++;
```

```
}

void CVoxel::Delete()
{
    if(Triangles != NULL)
    {
        delete [] Triangles;
        Triangles = NULL;
        TrianglesCount = 0;
        MaxTrianglesCount = 1;
        Size = 0.0f;
        Min = Max = MinE = MaxE = vec3(0.0f);
    }
}

bool CVoxel::Inside(const vec3 &Point)
{
    if(MinE.x < Point.x && Point.x < MaxE.x)
    {
        if(MinE.y < Point.y && Point.y < MaxE.y)
        {
            if(MinE.z < Point.z && Point.z < MaxE.z)
            {
                return true;
            }
        }
    }

    return false;
}

bool CVoxel::IntersectEdgesX(CTriangle *Triangle, float x, float y1, float y2, float z1, float z2)
{
    float NdotR = -Triangle->N.x;

    if(NdotR != 0.0f)
    {
        vec3 Origin = vec3(x, y1, z1);

        float Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

        if(Distance >= 0.0f && Distance <= Size)
        {
            return Triangle->Inside(Origin.x + Distance, Origin.y, Origin.z);
        }

        Origin.z = z2;

        Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

        if(Distance >= 0.0f && Distance <= Size)
        {
            return Triangle->Inside(Origin.x + Distance, Origin.y, Origin.z);
        }

        Origin.y = y2;

        Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

        if(Distance >= 0.0f && Distance <= Size)
        {
            return Triangle->Inside(Origin.x + Distance, Origin.y, Origin.z);
        }

        Origin.z = z1;

        Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

        if(Distance >= 0.0f && Distance <= Size)
        {
            return Triangle->Inside(Origin.x + Distance, Origin.y, Origin.z);
        }
    }

    return false;
}

bool CVoxel::IntersectEdgesY(CTriangle *Triangle, float y, float x1, float x2, float z1, float z2)
{
    float NdotR = -Triangle->N.y;

    if(NdotR != 0.0f)
    {
        vec3 Origin = vec3(x1, y, z1);

        float Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

        if(Distance >= 0.0f && Distance <= Size)
```

```
		{
			return Triangle->Inside(Origin.x, Origin.y + Distance, Origin.z);
		}

		Origin.x = x2;

		Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

		if(Distance >= 0.0f && Distance <= Size)
		{
			return Triangle->Inside(Origin.x, Origin.y + Distance, Origin.z);
		}

		Origin.z = z2;

		Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

		if(Distance >= 0.0f && Distance <= Size)
		{
			return Triangle->Inside(Origin.x, Origin.y + Distance, Origin.z);
		}

		Origin.x = x1;

		Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

		if(Distance >= 0.0f && Distance <= Size)
		{
			return Triangle->Inside(Origin.x, Origin.y + Distance, Origin.z);
		}
	}

	return false;
}

bool CVoxel::IntersectEdgesZ(CTriangle *Triangle, float z, float x1, float x2, float y1, float y2)
{
	float NdotR = -Triangle->N.z;

	if(NdotR != 0.0f)
	{
		vec3 Origin = vec3(x1, y1, z);

		float Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;
```

```
        if(Distance >= 0.0f && Distance <= Size)
        {
            return Triangle->Inside(Origin.x, Origin.y, Origin.z + Distance);
        }

        Origin.x = x2;

        Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

        if(Distance >= 0.0f && Distance <= Size)
        {
            return Triangle->Inside(Origin.x, Origin.y, Origin.z + Distance);
        }

        Origin.y = y2;

        Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

        if(Distance >= 0.0f && Distance <= Size)
        {
            return Triangle->Inside(Origin.x, Origin.y, Origin.z + Distance);
        }

        Origin.x = x1;

        Distance = (dot(Triangle->N, Origin) + Triangle->D) / NdotR;

        if(Distance >= 0.0f && Distance <= Size)
        {
            return Triangle->Inside(Origin.x, Origin.y, Origin.z + Distance);
        }
    }

    return false;
}

bool CVoxel::IntersectFacesX(CTriangle *Triangle, float D1, float D2)
{
    vec3 *Origin = (vec3*)&Triangle->a;
    vec3 *Ray = (vec3*)&Triangle->ab;
    float *Length = &Triangle->lab;

    float NdotR, d, y, z;
```

```
for(int i = 0; i < 3; i++)
{
        NdotR = -Ray->x;

        if(NdotR != 0.0f)
        {
                d = (Origin->x - D1) / NdotR;

                if(0.0f <= d && d <= *Length)
                {
                        y = Ray->y * d + Origin->y;

                        if(Min.y <= y && y <= Max.y)
                        {
                                z = Ray->z * d + Origin->z;

                                if(Min.z <= z && z <= Max.z)
                                {
                                        return true;
                                }
                        }
                }

                d = (Origin->x - D2) / NdotR;

                if(0.0f <= d && d <= *Length)
                {
                        y = Ray->y * d + Origin->y;

                        if(Min.y <= y && y <= Max.y)
                        {
                                z = Ray->z * d + Origin->z;

                                if(Min.z <= z && z <= Max.z)
                                {
                                        return true;
                                }
                        }
                }
        }

        Ray++;
        Origin++;
        Length++;
    }

    return false;
}

bool CVoxel::IntersectFacesY(CTriangle *Triangle, float D1, float D2)
{
    vec3 *Origin = (vec3*)&Triangle->a;
    vec3 *Ray = (vec3*)&Triangle->ab;
    float *Length = &Triangle->lab;

    float NdotR, d, x, z;

    for(int i = 0; i < 3; i++)
    {
        NdotR = -Ray->y;

        if(NdotR != 0.0f)
        {
            d = (Origin->y - D1) / NdotR;

            if(0.0f <= d && d <= *Length)
            {
                x = Ray->x * d + Origin->x;

                if(Min.x <= x && x <= Max.x)
                {
                    z = Ray->z * d + Origin->z;

                    if(Min.z <= z && z <= Max.z)
                    {
                        return true;
                    }
                }
            }

            d = (Origin->y - D2) / NdotR;

            if(0.0f <= d && d <= *Length)
            {
                x = Ray->x * d + Origin->x;

                if(Min.x <= x && x <= Max.x)
                {
                    z = Ray->z * d + Origin->z;

                    if(Min.z <= z && z <= Max.z)
                    {
                        return true;
                    }
                }
            }
        }

        Ray++;
        Origin++;
        Length++;
    }

    return false;
}

bool CVoxel::IntersectFacesZ(CTriangle *Triangle, float D1, float D2)
{
    vec3 *Origin = (vec3*)&Triangle->a;
    vec3 *Ray = (vec3*)&Triangle->ab;
    float *Length = &Triangle->lab;

    float NdotR, d, x, y;

    for(int i = 0; i < 3; i++)
    {
        NdotR = -Ray->z;

        if(NdotR != 0.0f)
        {
            d = (Origin->z - D1) / NdotR;

            if(0.0f <= d && d <= *Length)
            {
                x = Ray->x * d + Origin->x;

                if(Min.x <= x && x <= Max.x)
                {
                    y = Ray->y * d + Origin->y;

                    if(Min.y <= y && y <= Max.y)
```

```
					{
						return true;
					}
				}
			}

			d = (Origin->z - D2) / NdotR;

			if(0.0f <= d && d <= *Length)
			{
				x = Ray->x * d + Origin->x;

				if(Min.x <= x && x <= Max.x)
				{
					y = Ray->y * d + Origin->y;

					if(Min.y <= y && y <= Max.y)
					{
						return true;
					}
				}
			}
		}

		Ray++;
		Origin++;
		Length++;
	}

	return false;
}

bool CVoxel::Intersect(CTriangle *Triangle)
{
	if(Inside(Triangle->a)) return true;
	if(Inside(Triangle->b)) return true;
	if(Inside(Triangle->c)) return true;

	if(IntersectFacesX(Triangle, Min.x, Max.x)) return true;
	if(IntersectFacesY(Triangle, Min.y, Max.y)) return true;
	if(IntersectFacesZ(Triangle, Min.z, Max.z)) return true;

	if(IntersectEdgesX(Triangle, Min.x, Min.y, Max.y, Min.z, Max.z)) return true;
	if(IntersectEdgesY(Triangle, Min.y, Min.x, Max.x, Min.z, Max.z)) return true;
```

```
    if(IntersectEdgesZ(Triangle, Min.z, Min.x, Max.x, Min.y, Max.y)) return true;

    return false;
}

void CVoxel::Set(const vec3 &Min, float Size)
{
    this->Size = Size;
    this->Min = Min;
    this->Max = this->Min + Size;
    this->MinE = this->Min - 0.001f;
    this->MaxE = this->Max + 0.001f;
}

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

CUniformGrid::CUniformGrid()
{
    X = Y = Z = Xm1 = Ym1 = Zm1 = XY = XYZ = 0;
    VoxelSize = 0.0f;
    Voxels = NULL;
}

CUniformGrid::~CUniformGrid()
{
}

void CUniformGrid::Delete()
{
    if(Voxels != NULL)
    {
        for(int i = 0; i < XYZ; i++)
        {
            Voxels[i].Delete();
        }

        Min = Max = vec3(0.0f);
        X = Y = Z = Xm1 = Ym1 = Zm1 = XY = XYZ = 0;
        VoxelSize = 0.0f;
        delete [] Voxels;
        Voxels = NULL;
    }
}
```

```
void CUniformGrid::Generate(CTriangle *Triangles, int TrianglesCount, float VoxelSize)
{
    Delete();

    if(Triangles != NULL && TrianglesCount > 0)
    {
        this->VoxelSize = VoxelSize;

        CTriangle *LastTriangle = Triangles + TrianglesCount;

        Min = Max = Triangles->a;

        for(CTriangle *Triangle = Triangles; Triangle < LastTriangle; Triangle++)
        {
            if(Triangle->a.x < Min.x) Min.x = Triangle->a.x;
            if(Triangle->a.y < Min.y) Min.y = Triangle->a.y;
            if(Triangle->a.z < Min.z) Min.z = Triangle->a.z;

            if(Triangle->b.x < Min.x) Min.x = Triangle->b.x;
            if(Triangle->b.y < Min.y) Min.y = Triangle->b.y;
            if(Triangle->b.z < Min.z) Min.z = Triangle->b.z;

            if(Triangle->c.x < Min.x) Min.x = Triangle->c.x;
            if(Triangle->c.y < Min.y) Min.y = Triangle->c.y;
            if(Triangle->c.z < Min.z) Min.z = Triangle->c.z;

            if(Triangle->a.x > Max.x) Max.x = Triangle->a.x;
            if(Triangle->a.y > Max.y) Max.y = Triangle->a.y;
            if(Triangle->a.z > Max.z) Max.z = Triangle->a.z;

            if(Triangle->b.x > Max.x) Max.x = Triangle->b.x;
            if(Triangle->b.y > Max.y) Max.y = Triangle->b.y;
            if(Triangle->b.z > Max.z) Max.z = Triangle->b.z;

            if(Triangle->c.x > Max.x) Max.x = Triangle->c.x;
            if(Triangle->c.y > Max.y) Max.y = Triangle->c.y;
            if(Triangle->c.z > Max.z) Max.z = Triangle->c.z;
        }

        Min /= VoxelSize; Max /= VoxelSize;

        Min.x = floor(Min.x); Max.x = ceil(Max.x);
        Min.y = floor(Min.y); Max.y = ceil(Max.y);
        Min.z = floor(Min.z); Max.z = ceil(Max.z);

        if(Min.x == Max.x) Max.x += 1.0f;
        if(Min.y == Max.y) Max.y += 1.0f;
        if(Min.z == Max.z) Max.z += 1.0f;

        X = (int)(Max.x - Min.x); Xm1 = X - 1;
        Y = (int)(Max.y - Min.y); Ym1 = Y - 1;
        Z = (int)(Max.z - Min.z); Zm1 = Z - 1;

        XY = X * Y;
        XYZ = XY * Z;

        Min *= VoxelSize; Max *= VoxelSize;

        Voxels = new CVoxel[XYZ];

        for(int z = 0; z < Z; z++)
        {
            for(int y = 0; y < Y; y++)
            {
                for(int x = 0; x < X; x++)
                {
                    Voxels[XY * z + X * y + x].Set(VoxelToWorld(vec3((float)x, (float)y,
(float)z)), VoxelSize);
                }
            }
        }

        for(CTriangle *Triangle = Triangles; Triangle < LastTriangle; Triangle++)
        {
            vec3 tmin = Triangle->a, tmax = tmin;

            if(Triangle->b.x < tmin.x) tmin.x = Triangle->b.x;
            if(Triangle->b.y < tmin.y) tmin.y = Triangle->b.y;
            if(Triangle->b.z < tmin.z) tmin.z = Triangle->b.z;

            if(Triangle->b.x > tmax.x) tmax.x = Triangle->b.x;
            if(Triangle->b.y > tmax.y) tmax.y = Triangle->b.y;
            if(Triangle->b.z > tmax.z) tmax.z = Triangle->b.z;

            if(Triangle->c.x < tmin.x) tmin.x = Triangle->c.x;
            if(Triangle->c.y < tmin.y) tmin.y = Triangle->c.y;
            if(Triangle->c.z < tmin.z) tmin.z = Triangle->c.z;
            if(Triangle->c.x > tmax.x) tmax.x = Triangle->c.x;
            if(Triangle->c.y > tmax.y) tmax.y = Triangle->c.y;
            if(Triangle->c.z > tmax.z) tmax.z = Triangle->c.z;

            int vminx = WorldToVoxelX(tmin.x), vmaxx = WorldToVoxelX(tmax.x);
            int vminy = WorldToVoxelY(tmin.y), vmaxy = WorldToVoxelY(tmax.y);
            int vminz = WorldToVoxelZ(tmin.z), vmaxz = WorldToVoxelZ(tmax.z);

            if(vminx >= X) vminx = Xm1; if(vmaxx >= X) vmaxx = Xm1;
            if(vminy >= Y) vminy = Ym1; if(vmaxy >= Y) vmaxy = Ym1;
            if(vminz >= Z) vminz = Zm1; if(vmaxz >= Z) vmaxz = Zm1;

            for(int z = vminz; z <= vmaxz; z++)
            {
                for(int y = vminy; y <= vmaxy; y++)
                {
                    for(int x = vminx; x <= vmaxx; x++)
                    {
                        CVoxel *Voxel = Voxels + (XY * z + X * y + x);

                        if(Voxel->Intersect(Triangle))
                        {
                            Voxel->Add(Triangle);
                        }
                    }
                }
            }
        }
    }
}

vec3 CUniformGrid::Traverse(const vec3 &Voxel, const vec3 &Origin, const vec3 &Ray)
{
    vec3 voxel = Voxel, step, out, t, delta = VoxelSize / Ray;

    if(Ray.x < 0.0f)
    {
        step.x = -1.0f;
        out.x = voxel.x <= 0.0f ? voxel.x - 1.0f : -1.0f;
        t.x = (VoxelToWorldX(voxel.x) - Origin.x) / Ray.x;
    }
    else
    {
```

```
    step.x = 1.0f;
    out.x = voxel.x >= X ? voxel.x + 1 : X;
    t.x = (VoxelToWorldX(voxel.x + 1.0f) - Origin.x) / Ray.x;
}

if(Ray.y < 0.0f)
{
    step.y = -1.0f;
    out.y = voxel.y <= 0.0f ? voxel.y - 1.0f : -1.0f;
    t.y = (VoxelToWorldY(voxel.y) - Origin.y) / Ray.y;
}
else
{
    step.y = 1.0f;
    out.y = voxel.y >= Y ? voxel.y + 1 : Y;
    t.y = (VoxelToWorldY(voxel.y + 1.0f) - Origin.y) / Ray.y;
}

if(Ray.z < 0.0f)
{
    step.z = -1.0f;
    out.z = voxel.z <= 0.0f ? voxel.z - 1.0f : -1.0f;
    t.z = (VoxelToWorldZ(voxel.z) - Origin.z) / Ray.z;
}
else
{
    step.z = 1.0f;
    out.z = voxel.z >= Z ? voxel.z + 1 : Z;
    t.z = (VoxelToWorldZ(voxel.z + 1.0f) - Origin.z) / Ray.z;
}

delta *= step;

while(1)
{
    int x = (int)voxel.x, y = (int)voxel.y, z = (int)voxel.z;

    if(x >= 0 && x < X && y >= 0 && y < Y && z >= 0 && z < Z)
    {
        CVoxel *Voxel = Voxels + (XY * z + X * y + x);
        CTriangle **Triangles = Voxel->Triangles, *Triangle;
        int TrianglesCount = Voxel->TrianglesCount;

        RTData rtdata;
```

```
            for(int i = 0; i < TrianglesCount; i++)
            {
                Triangle = Triangles[i];

                if(Triangle->Intersect(Origin, Ray, rtdata.Distance, rtdata.TestDistance,
rtdata.TestPoint))
                {
                    if(Voxel->Inside(rtdata.TestPoint))
                    {
                        rtdata.Point = rtdata.TestPoint;
                        rtdata.Distance = rtdata.TestDistance;
                        rtdata.Triangle = Triangle;
                    }
                }
            }

            if(rtdata.Triangle)
            {
                rtdata.Color = rtdata.Triangle->Color;

                float NdotL = -dot(rtdata.Triangle->N, Ray);

                if(NdotL < 0.0f)
                {
                    NdotL = 0.0f;
                }

                rtdata.Color *= 0.75f * NdotL + 0.25f;

                return rtdata.Color;
            }
        }

        float min = t.x;

        if(t.y < min) min = t.y;
        if(t.z < min) min = t.z;

        if(t.x == min)
        {
           voxel.x += step.x;
           if(voxel.x == out.x) break;
           t.x += delta.x;
        }

        if(t.y == min)
        {
            voxel.y += step.y;
            if(voxel.y == out.y) break;
            t.y += delta.y;
        }

        if(t.z == min)
        {
            voxel.z += step.z;
            if(voxel.z == out.z) break;
            t.z += delta.z;
        }
    }

    return vec3(0.0f);
}

float CUniformGrid::VoxelToWorldX(float x)
{
    return x * VoxelSize + Min.x;
}

float CUniformGrid::VoxelToWorldY(float y)
{
    return y * VoxelSize + Min.y;
}

float CUniformGrid::VoxelToWorldZ(float z)
{
    return z * VoxelSize + Min.z;
}

vec3 CUniformGrid::VoxelToWorld(const vec3 &Voxel)
{
    return Voxel * VoxelSize + Min;
}

int CUniformGrid::WorldToVoxelX(float x)
{
    return (int)floor((x - Min.x) / VoxelSize);
}
```

```
int CUniformGrid::WorldToVoxelY(float y)
{
	return (int)floor((y - Min.y) / VoxelSize);
}

int CUniformGrid::WorldToVoxelZ(float z)
{
	return (int)floor((z - Min.z) / VoxelSize);
}

vec3 CUniformGrid::WorldToVoxel(const vec3 &World)
{
	vec3 voxel = (World - Min) / VoxelSize;

	voxel.x = floor(voxel.x);
	voxel.y = floor(voxel.y);
	voxel.z = floor(voxel.z);

	return voxel;
}

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

CRayTracer::CRayTracer()
{
	ColorBuffer = NULL;

	Triangles = NULL;
	TrianglesCount = 0;

	SuperSampling = false;
}

CRayTracer::~CRayTracer()
{
}

bool CRayTracer::Init()
{
	if(InitScene() == false)
	{
		return false;
```

```
    }

    UniformGrid.Generate(Triangles, TrianglesCount);

    return true;
}

void CRayTracer::RayTrace(int x, int y)
{
    if(ColorBuffer != NULL && Triangles != NULL && TrianglesCount > 0)
    {
        vec3 Color, Voxel = UniformGrid.WorldToVoxel(Camera.Position);

        if(!SuperSampling)
        {
            Color = UniformGrid.Traverse(Voxel, Camera.Position,
normalize(*(vec3*)&(Camera.RayMatrix * vec4((float)x + 0.5f, (float)y + 0.5f, 0.0f, 1.0f))));
        }
        else
        {
            for(float yy = 0.125f; yy < 1.0f; yy += 0.25f)
            {
                for(float xx = 0.125f; xx < 1.0f; xx += 0.25f)
                {
                    Color += UniformGrid.Traverse(Voxel, Camera.Position,
normalize(*(vec3*)&(Camera.RayMatrix * vec4((float)x + xx, (float)y + yy, 0.5f, 1.0f))));
                }
            }

            Color /= 16.0f;
        }

        BYTE *colorbuffer = (LineWidth * y + x) * 3 + ColorBuffer;

        colorbuffer[2] = Color.r <= 0.0f ? 0 : Color.r >= 1.0 ? 255 : (BYTE)(Color.r * 255);
        colorbuffer[1] = Color.g <= 0.0f ? 0 : Color.g >= 1.0 ? 255 : (BYTE)(Color.g * 255);
        colorbuffer[0] = Color.b <= 0.0f ? 0 : Color.b >= 1.0 ? 255 : (BYTE)(Color.b * 255);
    }
}

void CRayTracer::Resize(int Width, int Height)
{
    this->Width = Width;
    this->Height = Height;
    if(ColorBuffer != NULL)
    {
        delete [] ColorBuffer;
        ColorBuffer = NULL;
    }

    if(Width > 0 && Height > 0)
    {
        LineWidth = Width;

        int WidthMod4 = Width % 4;

        if(WidthMod4 > 0)
        {
            LineWidth += 4 - WidthMod4;
        }

        ColorBuffer = new BYTE[LineWidth * Height * 3];

        memset(&ColorBufferInfo, 0, sizeof(BITMAPINFOHEADER));
        ColorBufferInfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
        ColorBufferInfo.bmiHeader.biPlanes = 1;
        ColorBufferInfo.bmiHeader.biBitCount = 24;
        ColorBufferInfo.bmiHeader.biCompression = BI_RGB;
        ColorBufferInfo.bmiHeader.biWidth = LineWidth;
        ColorBufferInfo.bmiHeader.biHeight = Height;

        Camera.VPin[0] = 1.0f / (float)Width;
        Camera.VPin[5] = 1.0f / (float)Height;

        float tany = tan(45.0f / 360.0f * (float)M_PI), aspect = (float)Width / (float)Height;

        Camera.Pin[0] = tany * aspect;
        Camera.Pin[5] = tany;
        Camera.Pin[10] = 0.0f;
        Camera.Pin[14] = -1.0f;

        Camera.CalculateRayMatrix();
    }
}

void CRayTracer::Destroy()
{
	UniformGrid.Delete();

	if(Triangles != NULL)
	{
		delete [] Triangles;
		Triangles = NULL;
		TrianglesCount = 0;
	}

	if(ColorBuffer != NULL)
	{
		delete [] ColorBuffer;
		ColorBuffer = NULL;
	}
}

void CRayTracer::ClearColorBuffer()
{
	if(ColorBuffer != NULL)
	{
		memset(ColorBuffer, 0, LineWidth * Height * 3);
	}
}

void CRayTracer::SwapBuffers(HDC hDC)
{
	if(ColorBuffer != NULL)
	{
		StretchDIBits(hDC, 0, 0, Width, Height, 0, 0, Width, Height, ColorBuffer,
&ColorBufferInfo, DIB_RGB_COLORS, SRCCOPY);
	}
}

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

bool CMyRayTracer::InitScene()
{
	bool Error = false;

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

```
    int BoxesCount = 256;

    TrianglesCount = 12 * BoxesCount;

    Triangles = new CTriangle[TrianglesCount];

    int t = 0;

    srand(GetTickCount());

    for(int i = 0; i < BoxesCount; i++)
    {
        vec3 m = -8.0f + 16.0f * vec3((float)rand() / (float)RAND_MAX, (float)rand() / (float)RAND_MAX, (float)rand() / (float)RAND_MAX);
        vec3 s = 0.25f + 1.5f * vec3((float)rand() / (float)RAND_MAX, (float)rand() / (float)RAND_MAX, (float)rand() / (float)RAND_MAX);
        vec3 color = vec3((float)rand() / (float)RAND_MAX, (float)rand() / (float)RAND_MAX, (float)rand() / (float)RAND_MAX);

        mat3x3 RS;

        RS = RS * mat3x3(rotate(360.0f * (float)rand() / (float)RAND_MAX, vec3(1.0f, 0.0f, 0.0f)));
        RS = RS * mat3x3(rotate(360.0f * (float)rand() / (float)RAND_MAX, vec3(0.0f, 1.0f, 0.0f)));
        RS = RS * mat3x3(rotate(360.0f * (float)rand() / (float)RAND_MAX, vec3(0.0f, 0.0f, 1.0f)));

        RS = RS * mat3x3(scale(s.x, s.y, s.z));

        Triangles[t++] = CTriangle(m + RS * vec3( 0.5f, -0.5f,  0.5f), m + RS * vec3( 0.5f, -0.5f, -0.5f), m + RS * vec3( 0.5f,  0.5f, -0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3( 0.5f,  0.5f, -0.5f), m + RS * vec3( 0.5f, 0.5f,  0.5f), m + RS * vec3( 0.5f, -0.5f,  0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3(-0.5f, -0.5f, -0.5f), m + RS * vec3(-0.5f, -0.5f,  0.5f), m + RS * vec3(-0.5f,  0.5f,  0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3(-0.5f,  0.5f,  0.5f), m + RS * vec3(-0.5f, 0.5f, -0.5f), m + RS * vec3(-0.5f, -0.5f, -0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3(-0.5f,  0.5f,  0.5f), m + RS * vec3( 0.5f, 0.5f,  0.5f), m + RS * vec3( 0.5f,  0.5f, -0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3( 0.5f,  0.5f, -0.5f), m + RS * vec3(-0.5f, 0.5f, -0.5f), m + RS * vec3(-0.5f,  0.5f,  0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3(-0.5f, -0.5f, -0.5f), m + RS * vec3( 0.5f, -
0.5f, -0.5f), m + RS * vec3( 0.5f, -0.5f,    0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3( 0.5f, -0.5f,    0.5f), m + RS * vec3(-0.5f, -
0.5f,    0.5f), m + RS * vec3(-0.5f, -0.5f, -0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3(-0.5f, -0.5f,    0.5f), m + RS * vec3( 0.5f, -
0.5f,    0.5f), m + RS * vec3( 0.5f,    0.5f,    0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3( 0.5f,    0.5f,    0.5f), m + RS * vec3(-0.5f,
0.5f,    0.5f), m + RS * vec3(-0.5f, -0.5f,    0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3( 0.5f, -0.5f, -0.5f), m + RS * vec3(-0.5f, -
0.5f, -0.5f), m + RS * vec3(-0.5f,    0.5f, -0.5f), color);
        Triangles[t++] = CTriangle(m + RS * vec3(-0.5f,    0.5f, -0.5f), m + RS * vec3( 0.5f,
0.5f, -0.5f), m + RS * vec3( 0.5f, -0.5f, -0.5f), color);
    }

    Camera.Look(vec3(0.0f, 0.0f, 10.0f), vec3(0.0f, 0.0f, 0.0f), true);

    return true;
}

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

CMyRayTracer RayTracer;

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

CWnd::CWnd()
{
}

CWnd::~CWnd()
{
}

bool CWnd::Create(HINSTANCE hInstance, char *WindowName, int Width, int Height)
{
    WNDCLASSEX WndClassEx;

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

    WndClassEx.cbSize = sizeof(WNDCLASSEX);
    WndClassEx.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
    WndClassEx.lpfnWndProc = WndProc;
    WndClassEx.hInstance = hInstance;
    WndClassEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    WndClassEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    WndClassEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    WndClassEx.lpszClassName = "Win32CPURayTracerWindow";

    if(RegisterClassEx(&WndClassEx) == 0)
    {
        ErrorLog.Set("RegisterClassEx failed!");
        return false;
    }

    this->WindowName = WindowName;

    this->Width = Width;
    this->Height = Height;

    DWORD Style = WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN;

    if((hWnd = CreateWindowEx(WS_EX_APPWINDOW, WndClassEx.lpszClassName,
WindowName, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL)) == NULL)
    {
        ErrorLog.Set("CreateWindowEx failed!");
        return false;
    }

    return RayTracer.Init();
}

void CWnd::RePaint()
{
    x = y = 0;
    InvalidateRect(hWnd, NULL, FALSE);
}

void CWnd::Show(bool Maximized)
{
    RECT dRect, wRect, cRect;

    GetWindowRect(GetDesktopWindow(), &dRect);
    GetWindowRect(hWnd, &wRect);
    GetClientRect(hWnd, &cRect);

    wRect.right += Width - cRect.right;
    wRect.bottom += Height - cRect.bottom;
```

```
	wRect.right -= wRect.left;
	wRect.bottom -= wRect.top;

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

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

	ShowWindow(hWnd, Maximized ? SW_SHOWMAXIMIZED : SW_SHOWNORMAL);
}

void CWnd::MsgLoop()
{
	MSG Msg;

	while(GetMessage(&Msg, NULL, 0, 0) > 0)
	{
		TranslateMessage(&Msg);
		DispatchMessage(&Msg);
	}
}

void CWnd::Destroy()
{
	RayTracer.Destroy();

	DestroyWindow(hWnd);
}

void CWnd::OnKeyDown(UINT Key)
{
	switch(Key)
	{
		case VK_F1:
			RayTracer.SuperSampling = !RayTracer.SuperSampling;
			RePaint();
			break;
	}

	if(Camera.OnKeyDown(Key))
	{
		RePaint();
	}
```

```
}

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

		LastX = X;
		LastY = Y;

		RePaint();
	}
}

void CWnd::OnMouseWheel(short zDelta)
{
	Camera.OnMouseWheel(zDelta);

	RePaint();
}

void CWnd::OnPaint()
{
	PAINTSTRUCT ps;

	HDC hDC = BeginPaint(hWnd, &ps);

	static DWORD Start;
	static bool RayTracing;

	if(x == 0 && y == 0)
	{
		RayTracer.ClearColorBuffer();

		Start = GetTickCount();

		RayTracing = true;
	}

	DWORD start = GetTickCount();

	while(GetTickCount() - start < 125 && y < Height)
	{
    int x16 = x + 16, y16 = y + 16;

    for(int yy = y; yy < y16; yy++)
    {
        if(yy < Height)
        {
            for(int xx = x; xx < x16; xx++)
            {
                if(xx < Width)
                {
                    RayTracer.RayTrace(xx, yy);
                }
                else
                {
                    break;
                }
            }
        }
        else
        {
            break;
        }
    }

    x = x16;

    if(x >= Width)
    {
        x = 0;
        y = y16;
    }
}

RayTracer.SwapBuffers(hDC);

if(RayTracing)
{
    if(y >= Height)
    {
        RayTracing = false;
    }

    DWORD End = GetTickCount();
```

```
		CString text = WindowName;

		text.Append(" - %dx%d", Width, Height);
		text.Append(", Time: %.03f s", (float)(End - Start) * 0.001f);

		SetWindowText(hWnd, text);

		InvalidateRect(hWnd, NULL, FALSE);
	}

	EndPaint(hWnd, &ps);
}

void CWnd::OnRButtonDown(int X, int Y)
{
	LastX = X;
	LastY = Y;
}

void CWnd::OnSize(int Width, int Height)
{
	this->Width = Width;
	this->Height = Height;

	RayTracer.Resize(Width, Height);

	RePaint();
}

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

CWnd Wnd;

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

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
	switch(uiMsg)
	{
		case WM_CLOSE:
			PostQuitMessage(0);
```

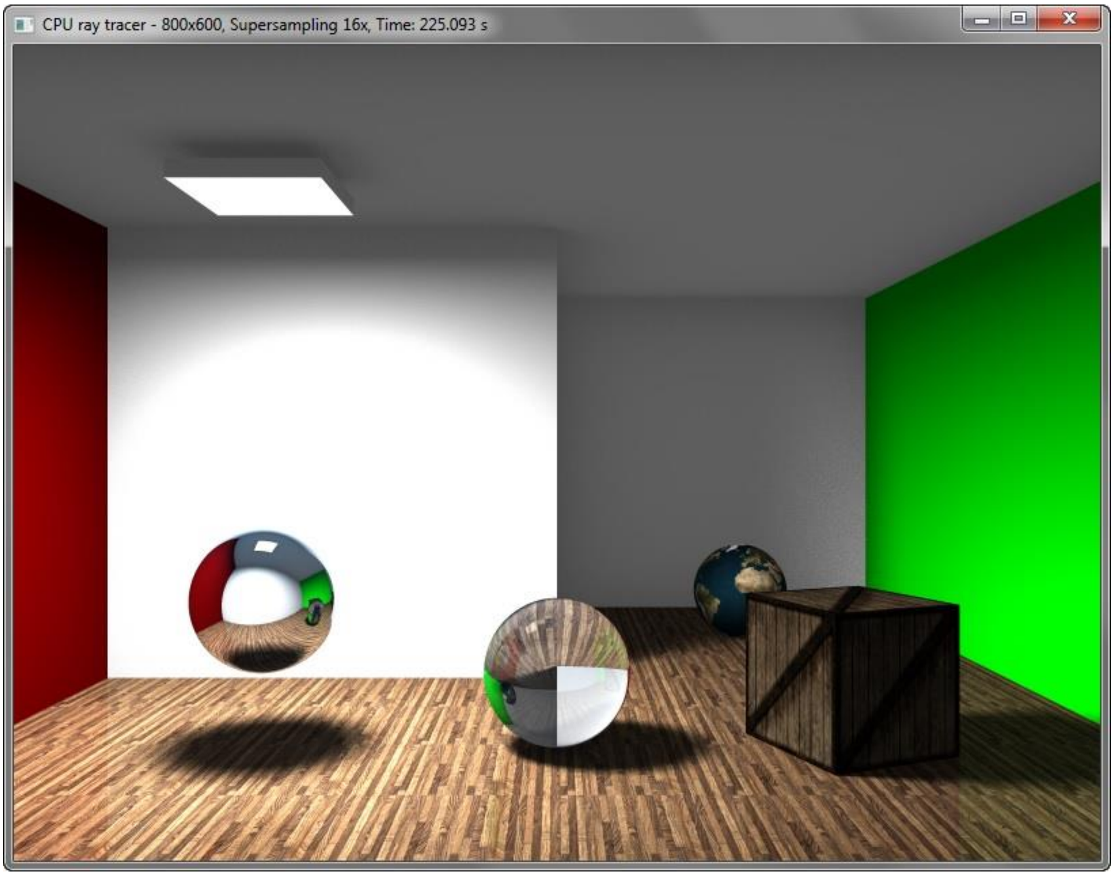
```
            break;

        case WM_MOUSEMOVE:
            Wnd.OnMouseMove(LOWORD(lParam), HIWORD(lParam));
            break;

        case 0x020A: // WM_MOUSWHEEL
            Wnd.OnMouseWheel(HIWORD(wParam));
            break;

        case WM_KEYDOWN:
            Wnd.OnKeyDown((UINT)wParam);
            break;

        case WM_PAINT:
            Wnd.OnPaint();
            break;

        case WM_RBUTTONDOWN:
            Wnd.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));
            break;

        case WM_SIZE:
            Wnd.OnSize(LOWORD(lParam), HIWORD(lParam));
            break;

        default:
            return DefWindowProc(hWnd, uiMsg, wParam, lParam);
    }

    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine, int iShow)
{
    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);

    if(Wnd.Create(hInstance, "CPU ray tracer 02 - Uniform grid", 800, 600))
    {
        Wnd.Show();
        Wnd.MsgLoop();
    }
    else
    {
        MessageBox(NULL, ErrorLog, "Error", MB_OK | MB_ICONERROR);
    }

    Wnd.Destroy();

    return 0;
}
```

第四章: CPU ray tracer

Ray tracing algorithm generates an image by tracing the path of light through pixels in an image plane. It is capable of producing a very high degree of visual realism, but at a greater computational cost.

Features:

Model-View-Controller pattern
Object-oriented programming
Vector and matrix operations
Texture loading using FreeImage
Nearest and bilinear texture filtering
Fast ray-sphere and ray-quad intersection tests (30 million per second)
Third person camera
Ray calculation using inverse OpenGL based matrices
Spherical texture mapping
Spherical and enclosed area lights
Light reflection and refraction
Hard and soft penumbra shadows
Ambient occlusion
Supersampling
High dynamic range
Elapsed time counter


CPU ray tracer - 800x600, Supersampling 16x, Time: 225.093 s

MSP, February 18, 2015, 07:04 PM

How can we render other objects than triangles and spheres, how can we load an obj file and how can we achieve a texture is not repeated?

Admin, February 18, 2015, 07:59 PM

Only triangles and spheres can be rendered in this tutorial. Everything can be rendered using triangles, so we didn't implement rendering of other types of primitives. Obj file loading is simple, you can find a code on this site. If you don't want a texture to be repeated, you have to find and implement a better way of texturing triangles. If you do so, then you're welcome to share your code with us.

```
////////////////////////////////////Header//////////////////////////////////////////
class CTexture
{
private:
	BYTE *Data;
	int Width, Height;

public:
	CTexture();
	~CTexture();

	bool CreateTexture2D(char *Texture2DFileName);
	vec3 GetColorNearest(float s, float t);
	vec3 GetColorBilinear(float s, float t);
	void Destroy();
};

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

class CQuad
{
public:
	float Reflection, Refraction, Eta, ODEta, D, D1, D2, D3, D4;
	vec3 a, b, c, d, Color, ab, ad, m, T, B, N, O, N1, N2, N3, N4;
	CTexture *Texture;
	// static UINT Intersections;

public:
	CQuad();
	CQuad(const vec3 &a, const vec3 &b, const vec3 &c, const vec3 &d, const vec3 &Color, CTexture *Texture = NULL, float Reflection = 0.0f, float Refraction = 0.0f, float Eta = 1.0f);

	bool Inside(vec3 &Point);
	bool Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance, vec3 &Point);
	bool Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance);
	bool Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance);
};

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

class CSphere
{
public:
	float Radius, Radius2, ODRadius, Reflection, Refraction, Eta, ODEta;
	vec3 Position, Color;
	CTexture *Texture;
	// static UINT Intersections;

public:
	CSphere();
	CSphere(const vec3 &Position, float Radius, const vec3 &Color, CTexture *Texture = NULL, float Reflection = 0.0f, float Refraction = 0.0f, float Eta = 1.0f);

	bool Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance, vec3
&Point);
	bool Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance);
	bool Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance);
};

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

class CLight
{
public:
	float Ambient, Diffuse, ConstantAttenuation, LinearAttenuation, QuadraticAttenuation;
	CSphere *Sphere;
	CQuad *Quad;

public:
	CLight();
	~CLight();
};

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

class CCamera
{
public:
	vec3 X, Y, Z, Reference, Position;
	mat4x4 Vin, Pin, Bin, VPin, RayMatrix;

public:
	CCamera();
	~CCamera();

	void CalculateRayMatrix();
	void LookAt(const vec3 &Reference, const vec3 &Position, bool 
RotateAroundReference = false);
	bool OnKeyDown(UINT nChar);
	void OnMouseMove(int dx, int dy);
	void OnMouseWheel(short zDelta);
};

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

```
class RTData
{
public:
	float Distance, TestDistance;
	vec3 Color, Point, TestPoint;
	CQuad *Quad;
	CSphere *Sphere;
	CLight *Light;

public:
	RTData();
};

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

class CRayTracer
{
private:
	BYTE *ColorBuffer;
	BITMAPINFO ColorBufferInfo;
	vec3 *HDRColorBuffer;
	int Width, LineWidth, Height, Samples, GISamples, WidthMSamples, HeightMSamples,
WidthMHeightMSamples2;
	float ODSamples2, ODGISamples, AmbientOcclusionIntensity,
ODGISamplesMAmbientOcclusionIntensity;

protected:
	CQuad *Quads, *LastQuad;
	CSphere *Spheres, *LastSphere;
	CLight *Lights, *LastLight;
	int QuadsCount, SpheresCount, LightsCount;

public:
	bool Textures, SoftShadows, AmbientOcclusion;

public:
	CRayTracer();
	~CRayTracer();

	bool Init();
	void RayTrace(int Line);
	void Resize(int Width, int Height);
	void Destroy();
	void ClearColorBuffer();
	int GetSamples();
	void MapHDRColors();
	bool SetSamples(int Samples);
	void SwapBuffers(HDC hDC);

protected:
	virtual bool InitScene() = 0;
	virtual void DestroyTextures() = 0;

private:
	bool Shadow(void *Object, vec3 &Point, vec3 &LightDirection, float LightDistance);
	vec3 LightIntensity(void *Object, vec3 &Point, vec3 &Normal, vec3 &LightPosition,
CLight *Light, float AO);
	float AmbientOcclusionFactor(void *Object, vec3 &Point, vec3 &Normal);
	void IlluminatePoint(void *Object, vec3 &Point, vec3 &Normal, vec3 &Color);
	vec3 RayTrace(vec3 &Origin, const vec3 &Ray, UINT Depth = 0, void *Object = NULL);
};

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

class CMyRayTracer : public CRayTracer
{
private:
	CTexture Floor, Cube, Earth;

protected:
	bool InitScene();
	void DestroyTextures();
};

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

class CWnd
{
protected:
	char *WindowName;
	HWND hWnd;
	HDC hDC;
	int Width, Height, Line;
	POINT LastCurPos;
```

```
public:
	CWnd();
	~CWnd();

	bool Create(HINSTANCE hInstance, char *WindowName, int Width, int Height);
	void RePaint();
	void Show(bool Maximized = false);
	void MsgLoop();
	void Destroy();

	void OnKeyDown(UINT Key);
	void OnMouseMove(int cx, int cy);
	void OnMouseWheel(short zDelta);
	void OnPaint();
	void OnRButtonDown(int cx, int cy);
	void OnSize(int Width, int Height);
};
////////////////////////////////////////CPP///////////////////////////////////
CTexture::CTexture()
{
	Data = NULL;
	Width = Height = 0;
}

CTexture::~CTexture()
{
}

bool CTexture::CreateTexture2D(char *Texture2DFileName)
{
	CString FileName = ModuleDirectory + Texture2DFileName;
	CString ErrorText = "Error loading file " + FileName + "! -> ";

	FREE_IMAGE_FORMAT fif = FreeImage_GetFileType(FileName);

	if(fif == FIF_UNKNOWN)
	{
		fif = FreeImage_GetFIFFromFilename(FileName);
	}

	if(fif == FIF_UNKNOWN)
	{
		ErrorLog.Append(ErrorText + "fif is FIF_UNKNOWN" + "\r\n");
```

```
        return false;
}

FIBITMAP *dib = NULL;

if(FreeImage_FIFSupportsReading(fif))
{
        dib = FreeImage_Load(fif, FileName);
}

if(dib == NULL)
{
        ErrorLog.Append(ErrorText + "dib is NULL" + "\r\n");
        return false;
}

int Width = FreeImage_GetWidth(dib);
int Height = FreeImage_GetHeight(dib);
int Pitch = FreeImage_GetPitch(dib);
int BPP = FreeImage_GetBPP(dib);

if(Width == 0 || Height == 0)
{
        ErrorLog.Append(ErrorText + "Width or Height is 0" + "\r\n");
        return false;
}

BYTE *Bits = FreeImage_GetBits(dib);

if(Bits == NULL)
{
        ErrorLog.Append(ErrorText + "Bits is NULL" + "\r\n");
        return false;
}

if(BPP != 24 && BPP != 32)
{
        FreeImage_Unload(dib);
        ErrorLog.Append(ErrorText + "BPP is not 24 nor 32" + "\r\n");
        return false;
}

Destroy();
```

```
    Data = new BYTE[Width * Height * 3];

    this->Width = Width;
    this->Height = Height;

    int bpp = BPP / 8;

    BYTE *data = Data, *line = Bits;

    for(int y = 0; y < Height; y++)
    {
        BYTE *pixel = line;

        for(int x = 0; x < Width; x++)
        {
            data[0] = pixel[2];
            data[1] = pixel[1];
            data[2] = pixel[0];

            pixel += bpp;
            data += 3;
        }

        line += Pitch;
    }

    FreeImage_Unload(dib);

    return true;
}

float OD255 = 1.0f / 255;

vec3 CTexture::GetColorNearest(float s, float t)
{
    vec3 Color = vec3(1.0f);

    if(Data != NULL)
    {
        s -= (int)s;
        t -= (int)t;

        if(s < 0.0f) s += 1.0f;
        if(t < 0.0f) t += 1.0f;
```

```
		int x = (int)(s * Width), y = (int)(t * Height);

		BYTE *data = (Width * y + x) * 3 + Data;

		Color.r = OD255 * data[0];
		Color.g = OD255 * data[1];
		Color.b = OD255 * data[2];
	}

	return Color;
}

vec3 CTexture::GetColorBilinear(float s, float t)
{
	vec3 Color = vec3(1.0f);

	if(Data != NULL)
	{
		s -= (int)s;
		t -= (int)t;

		if(s < 0.0f) s += 1.0f;
		if(t < 0.0f) t += 1.0f;

		float dx = s * Width - 0.5f, dy = t * Height - 0.5f;

		if(dx < 0.0f) dx += Width;
		if(dy < 0.0f) dy += Height;

		int x0 = (int)dx, y0 = (int)dy, x1 = (x0 + 1) % Width, y1 = (y0 + 1) % Height;

		int Width3 = Width * 3;

		BYTE *y0w = y0 * Width3 + Data;
		BYTE *y1w = y1 * Width3 + Data;

		int x03 = x0 * 3, x13 = x1 * 3;

		BYTE *a = y0w + x03;
		BYTE *b = y0w + x13;
		BYTE *c = y1w + x13;
		BYTE *d = y1w + x03;
```

```
        float u1 = dx - x0, v1 = dy - y0, u0 = 1.0f - u1, v0 = 1.0f - v1;

        u0 *= OD255;
        u1 *= OD255;

        float u0v0 = u0 * v0, u1v0 = u1 * v0, u1v1 = u1 * v1, u0v1 = u0 * v1;

        Color.r = u0v0 * a[0] + u1v0 * b[0] + u1v1 * c[0] + u0v1 * d[0];
        Color.g = u0v0 * a[1] + u1v0 * b[1] + u1v1 * c[1] + u0v1 * d[1];
        Color.b = u0v0 * a[2] + u1v0 * b[2] + u1v1 * c[2] + u0v1 * d[2];
    }

    return Color;
}

void CTexture::Destroy()
{
    if(Data != NULL)
    {
        delete [] Data;
        Data = NULL;
        Width = Height = 0;
    }
}

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

// UINT CQuad::Intersections;

CQuad::CQuad()
{
}

CQuad::CQuad(const vec3 &a, const vec3 &b, const vec3 &c, const vec3 &d, const vec3
&Color, CTexture *Texture, float Reflection, float Refraction, float Eta) : a(a), b(b), c(c), d(d),
N(N), D(D), Color(Color), Texture(Texture), Reflection(Reflection), Refraction(Refraction),
Eta(Eta)
{
    ab = b - a;
    ad = d - a;
    m = (a + b + c + d) / 4.0f;

    T = normalize(b - a);
```

```
	N = normalize(cross(b - a, c - a));
	B = cross(N, T);
	O = vec3(dot(T, a), dot(B, a), dot(N, a));

	D = -dot(N, a);
	ODEta = 1.0f / Eta;

	N1 = normalize(cross(N, b - a));
	D1 = -dot(N1, a);

	N2 = normalize(cross(N, c - b));
	D2 = -dot(N2, b);

	N3 = normalize(cross(N, d - c));
	D3 = -dot(N3, c);

	N4 = normalize(cross(N, a - d));
	D4 = -dot(N4, d);
}

bool CQuad::Inside(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;
	if(dot(N4, Point) + D4 < 0.0f) return false;

	return true;
}

bool CQuad::Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance,
vec3 &Point)
{
	// Intersections++;

	float NdotR = -dot(N, Ray);

	if(NdotR > 0.0f) // || (Refraction > 0.0f && NdotR < 0.0f))
	{
		Distance = (dot(N, Origin) + D) / NdotR;

		if(Distance >= 0.0f && Distance < MaxDistance)
		{
			Point = Ray * Distance + Origin;
```

```
            return Inside(Point);
        }
    }

    return false;
}

bool CQuad::Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance)
{
    // Intersections++;

    float NdotR = -dot(N, Ray);

    if(NdotR > 0.0f) // || (Refraction > 0.0f && NdotR < 0.0f))
    {
        Distance = (dot(N, Origin) + D) / NdotR;

        if(Distance >= 0.0f && Distance < MaxDistance)
        {
            return Inside(Ray * Distance + Origin);
        }
    }

    return false;
}

bool CQuad::Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance)
{
    // Intersections++;

    float NdotR = -dot(N, Ray);

    if(NdotR > 0.0f) // || (Refraction > 0.0f && NdotR < 0.0f))
    {
        float Distance = (dot(N, Origin) + D) / NdotR;

        if(Distance >= 0.0f && Distance < MaxDistance)
        {
            return Inside(Ray * Distance + Origin);
        }
    }

    return false;
```

```
}

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

// UINT CSphere::Intersections;

CSphere::CSphere()
{
}

CSphere::CSphere(const vec3 &Position, float Radius, const vec3 &Color, CTexture *Texture,
float Reflection, float Refraction, float Eta) : Position(Position), Radius(Radius), Color(Color),
Texture(Texture), Reflection(Reflection), Refraction(Refraction), Eta(Eta)
{
	Radius2 = Radius * Radius;
	ODRadius = 1.0f / Radius;
	ODEta = 1.0f / Eta;
}

bool CSphere::Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance,
vec3 &Point)
{
	// Intersections++;

	vec3 L = Position - Origin;

	float LdotR = dot(L, Ray);

	if(LdotR > 0.0f)
	{
		float D2 = length2(L) - LdotR * LdotR;

		if(D2 < Radius2)
		{
			Distance = LdotR - sqrt(Radius2 - D2);

			if(Distance >= 0.0f && Distance < MaxDistance)
			{
				Point = Ray * Distance + Origin;

				return true;
			}
		}
```

```
    }

    return false;
}

bool CSphere::Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance, float &Distance)
{
    // Intersections++;

    vec3 L = Position - Origin;

    float LdotR = dot(L, Ray);

    if(LdotR > 0.0f)
    {
        float D2 = length2(L) - LdotR * LdotR;

        if(D2 < Radius2)
        {
            Distance = LdotR - sqrt(Radius2 - D2);

            if(Distance >= 0.0f && Distance < MaxDistance)
            {
                return true;
            }
        }
    }

    return false;
}

bool CSphere::Intersect(vec3 &Origin, const vec3 &Ray, float MaxDistance)
{
    // Intersections++;

    vec3 L = Position - Origin;

    float LdotR = dot(L, Ray);

    if(LdotR > 0.0f)
    {
        float D2 = length2(L) - LdotR * LdotR;

        if(D2 < Radius2)
        {
            float Distance = LdotR - sqrt(Radius2 - D2);

            if(Distance >= 0.0f && Distance < MaxDistance)
            {
                return true;
            }
        }
    }

    return false;
}

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

CLight::CLight()
{
    Ambient = 1.0f;
    Diffuse = 1.0f;

    ConstantAttenuation = 1.0f;
    LinearAttenuation = 0.0f;
    QuadraticAttenuation = 0.0f;

    Sphere = NULL;
    Quad = NULL;
}

CLight::~CLight()
{
    if(Sphere)
    {
        delete Sphere;
    }
    else
    {
        delete Quad;
    }
}

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

```
CCamera::CCamera()
{
	X = vec3(1.0, 0.0, 0.0);
	Y = vec3(0.0, 1.0, 0.0);
	Z = vec3(0.0, 0.0, 1.0);

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

	Bin = BiasMatrixInverse();
}

CCamera::~CCamera()
{
}

void CCamera::CalculateRayMatrix()
{
	Vin[0] = X.x; Vin[4] = Y.x; Vin[8] = Z.x;
	Vin[1] = X.y; Vin[5] = Y.y; Vin[9] = Z.y;
	Vin[2] = X.z; Vin[6] = Y.z; Vin[10] = Z.z;

	RayMatrix = Vin * Pin * Bin * VPin;
}

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

	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;
	}

	CalculateRayMatrix();
}
```

```
bool CCamera::OnKeyDown(UINT nChar)
{
	float Distance = 0.125f;

	if(GetKeyState(VK_CONTROL) & 0x80)
	{
		Distance *= 0.5f;
	}

	if(GetKeyState(VK_SHIFT) & 0x80)
	{
		Distance *= 2.0f;
	}

	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(nChar == 'W')
	{
		Movement += Forward;
	}

	if(nChar == 'S')
	{
		Movement -= Forward;
	}

	if(nChar == 'A')
	{
		Movement -= Right;
	}

	if(nChar == 'D')
	{
		Movement += Right;
	}
```

```
    if(nChar == 'R')
    {
        Movement += Up;
    }

    if(nChar == 'F')
    {
        Movement -= Up;
    }

    Reference += Movement;
    Position += Movement;

    return Movement.x != 0.0f || Movement.y != 0.0f || Movement.z != 0.0f;
}

void CCamera::OnMouseMove(int dx, int dy)
{
    float sensitivity = 0.25f;

    float hangle = (float)dx * sensitivity;
    float vangle = (float)dy * sensitivity;

    Position -= Reference;

    Y = rotate(Y, vangle, X);
    Z = rotate(Z, vangle, X);

    if(Y.y < 0.0f)
    {
        Z = vec3(0.0f, Z.y > 0.0f ? 1.0f : -1.0f, 0.0f);
        Y = cross(Z, X);
    }

    X = rotate(X, hangle, vec3(0.0f, 1.0f, 0.0f));
    Y = rotate(Y, hangle, vec3(0.0f, 1.0f, 0.0f));
    Z = rotate(Z, hangle, vec3(0.0f, 1.0f, 0.0f));

    Position = Reference + Z * length(Position);

    CalculateRayMatrix();
}

void CCamera::OnMouseWheel(short 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;
}

CCamera Camera;

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

RTData::RTData()
{
	Distance = 1048576.0f;

	Quad = NULL;
	Light = NULL;
	Sphere = NULL;
}

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

CRayTracer::CRayTracer()
{
	ColorBuffer = NULL;
	HDRColorBuffer = NULL;

	Samples = 1;
	GISamples = 16;

	ODGISamples = 1.0f / (float)GISamples;
	AmbientOcclusionIntensity = 0.5f;
	ODGISamplesMAmbientOcclusionIntensity = ODGISamples *
```

```
AmbientOcclusionIntensity;

	Quads = NULL;
	Spheres = NULL;
	Lights = NULL;

	LastQuad = NULL;
	LastSphere = NULL;
	LastLight = NULL;

	QuadsCount = 0;
	SpheresCount = 0;
	LightsCount = 0;

	Textures = true;
	SoftShadows = false;
	AmbientOcclusion = false;

	srand(GetTickCount());
}

CRayTracer::~CRayTracer()
{
}

bool CRayTracer::Init()
{
	if(InitScene() == false)
	{
		return false;
	}

	LastQuad = Quads + QuadsCount;
	LastSphere = Spheres + SpheresCount;
	LastLight = Lights + LightsCount;

	return true;
}

void CRayTracer::RayTrace(int Line)
{
	if(ColorBuffer == NULL || HDRColorBuffer == NULL) return;

	vec3 *hdrcolorbuffer;
```

```
    BYTE *colorbuffer = LineWidth * Line * 3 + ColorBuffer;

    if(Samples == 1)
    {
        hdrcolorbuffer = Width * Line + HDRColorBuffer;

        for(int x = 0; x < Width; x++)
        {
            vec3 Color = RayTrace(Camera.Position, normalize(Camera.RayMatrix *
vec3((float)x, (float)Line, 0.0f)));

            hdrcolorbuffer->r = Color.r;
            hdrcolorbuffer->g = Color.g;
            hdrcolorbuffer->b = Color.b;

            hdrcolorbuffer++;

            colorbuffer[2] = Color.r <= 0.0f ? 0 : Color.r >= 1.0 ? 255 : (BYTE)(Color.r *
255);
            colorbuffer[1] = Color.g <= 0.0f ? 0 : Color.g >= 1.0 ? 255 : (BYTE)(Color.g *
255);
            colorbuffer[0] = Color.b <= 0.0f ? 0 : Color.b >= 1.0 ? 255 : (BYTE)(Color.b *
255);

            colorbuffer += 3;
        }
    }
    else
    {
        int Y = Line * Samples;

        for(int X = 0; X < WidthMSamples; X += Samples)
        {
            vec3 SamplesSum;

            for(int yy = 0; yy < Samples; yy++)
            {
                int Yyy = Y + yy;

                hdrcolorbuffer = WidthMSamples * Yyy + X + HDRColorBuffer;

                for(int xx = 0; xx < Samples; xx++)
                {
                    vec3 Color = RayTrace(Camera.Position,
normalize(Camera.RayMatrix * vec3((float)(X + xx), (float)Yyy, 0.0f)));

                    hdrcolorbuffer->r = Color.r;
                    hdrcolorbuffer->g = Color.g;
                    hdrcolorbuffer->b = Color.b;

                    hdrcolorbuffer++;

                    SamplesSum.r += Color.r <= 0.0f ? 0.0f : Color.r >= 1.0 ? 1.0f :
Color.r;
                    SamplesSum.g += Color.g <= 0.0f ? 0.0f : Color.g >= 1.0 ? 1.0f :
Color.g;
                    SamplesSum.b += Color.b <= 0.0f ? 0.0f : Color.b >= 1.0 ? 1.0f :
Color.b;
                }
            }

            SamplesSum.r *= ODSamples2;
            SamplesSum.g *= ODSamples2;
            SamplesSum.b *= ODSamples2;

            colorbuffer[2] = (BYTE)(SamplesSum.r * 255);
            colorbuffer[1] = (BYTE)(SamplesSum.g * 255);
            colorbuffer[0] = (BYTE)(SamplesSum.b * 255);

            colorbuffer += 3;
        }
    }
}

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

    if(ColorBuffer != NULL)
    {
        delete [] ColorBuffer;
        ColorBuffer = NULL;
    }

    if(HDRColorBuffer != NULL)
    {
        delete [] HDRColorBuffer;
```

```
			HDRColorBuffer = NULL;
		}

		if(Width > 0 && Height > 0)
		{
			LineWidth = Width;

			int WidthMod4 = Width % 4;

			if(WidthMod4 > 0)
			{
				LineWidth += 4 - WidthMod4;
			}

			ColorBuffer = new BYTE[LineWidth * Height * 3];

			memset(&ColorBufferInfo, 0, sizeof(BITMAPINFOHEADER));
			ColorBufferInfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
			ColorBufferInfo.bmiHeader.biPlanes = 1;
			ColorBufferInfo.bmiHeader.biBitCount = 24;
			ColorBufferInfo.bmiHeader.biCompression = BI_RGB;
			ColorBufferInfo.bmiHeader.biWidth = LineWidth;
			ColorBufferInfo.bmiHeader.biHeight = Height;

			WidthMSamples = Width * Samples;
			HeightMSamples = Height * Samples;
			WidthMHeightMSamples2 = WidthMSamples * HeightMSamples;
			ODSamples2 = 1.0f / (float)(Samples * Samples);

			HDRColorBuffer = new vec3[WidthMHeightMSamples2];

			Camera.VPin[0] = 1.0f / (float)(WidthMSamples - 1);
			Camera.VPin[5] = 1.0f / (float)(HeightMSamples - 1);

			float tany = tan(45.0f / 360.0f * (float)M_PI), aspect = (float)Width / (float)Height;

			Camera.Pin[0] = tany * aspect;
			Camera.Pin[5] = tany;
			Camera.Pin[10] = 0.0f;
			Camera.Pin[14] = -1.0f;

			Camera.CalculateRayMatrix();
		}
	}
```

```
void CRayTracer::Destroy()
{
	DestroyTextures();

	if(Quads != NULL)
	{
		delete [] Quads;
		Quads = NULL;
		QuadsCount = 0;
		LastQuad = NULL;
	}

	if(Spheres != NULL)
	{
		delete [] Spheres;
		Spheres = NULL;
		SpheresCount = 0;
		LastSphere = NULL;
	}

	if(Lights != NULL)
	{
		delete [] Lights;
		Lights = NULL;
		LightsCount = 0;
		LastLight = NULL;
	}

	if(ColorBuffer != NULL)
	{
		delete [] ColorBuffer;
		ColorBuffer = NULL;
	}

	if(HDRColorBuffer != NULL)
	{
		delete [] HDRColorBuffer;
		HDRColorBuffer = NULL;
	}
}

void CRayTracer::ClearColorBuffer()
{
    if(ColorBuffer != NULL)
    {
        memset(ColorBuffer, 0, LineWidth * Height * 3);
    }
}

int CRayTracer::GetSamples()
{
    return Samples * Samples;
}

void CRayTracer::MapHDRColors()
{
    if(ColorBuffer == NULL || HDRColorBuffer == NULL) return;

    float SumLum = 0.0f, LumWhite = 0.0f;
    int LumNotNull = 0;

    vec3 *Color = HDRColorBuffer;

    for(int i = 0; i < WidthMHeightMSamples2; i++)
    {
        float Luminance = (Color->r * 0.2125f + Color->g * 0.7154f + Color->b *
0.0721f);

        if(Luminance > 0.0f)
        {
            SumLum += Luminance;

            LumNotNull++;

            LumWhite = LumWhite > Luminance ? LumWhite : Luminance;
        }

        Color++;
    }

    float AvgLum = SumLum / (float)LumNotNull;

    LumWhite /= AvgLum;

    float LumWhite2 = LumWhite * LumWhite;

    Color = HDRColorBuffer;
    vec3 ColorMMappingFactor;

    for(int i = 0; i < WidthMHeightMSamples2; i++)
    {
        float Luminance = (Color->r * 0.2125f + Color->g * 0.7154f + Color->b *
0.0721f);

        float LumRel = Luminance / AvgLum;
        float MappingFactor = LumRel * (1.0f + LumRel / LumWhite2) / (1.0f + LumRel);

        ColorMMappingFactor.r = Color->r * MappingFactor;
        ColorMMappingFactor.g = Color->g * MappingFactor;
        ColorMMappingFactor.b = Color->b * MappingFactor;

        Color->r = ColorMMappingFactor.r <= 0.0f ? 0.0f : ColorMMappingFactor.r >=
1.0f ? 1.0f : ColorMMappingFactor.r;
        Color->g = ColorMMappingFactor.g <= 0.0f ? 0.0f : ColorMMappingFactor.g >=
1.0f ? 1.0f : ColorMMappingFactor.g;
        Color->b = ColorMMappingFactor.b <= 0.0f ? 0.0f : ColorMMappingFactor.b >=
1.0f ? 1.0f : ColorMMappingFactor.b;

        Color++;
    }

    int LineWidthSWidthM3 = (LineWidth - Width) * 3;

    BYTE *colorbuffer = ColorBuffer;

    if(Samples == 1)
    {
        Color = HDRColorBuffer;

        for(int y = 0; y < Height; y++)
        {
            for(int x = 0; x < Width; x++)
            {
                colorbuffer[2] = (BYTE)(Color->r * 255);
                colorbuffer[1] = (BYTE)(Color->g * 255);
                colorbuffer[0] = (BYTE)(Color->b * 255);

                Color++;
                colorbuffer += 3;
            }
            colorbuffer += LineWidthSWidthM3;
        }
    }
    else
    {
        for(int y = 0, Y = 0; y < Height; y++, Y += Samples)
        {
            for(int X = 0; X < WidthMSamples; X += Samples)
            {
                vec3 ColorSum;

                for(int yy = 0; yy < Samples; yy++)
                {
                    Color = WidthMSamples * (Y + yy) + X + HDRColorBuffer;

                    for(int xx = 0; xx < Samples; xx++)
                    {
                        ColorSum.r += Color->r;
                        ColorSum.g += Color->g;
                        ColorSum.b += Color->b;

                        Color++;
                    }
                }

                ColorSum.r *= ODSamples2;
                ColorSum.g *= ODSamples2;
                ColorSum.b *= ODSamples2;

                colorbuffer[2] = (BYTE)(ColorSum.r * 255);
                colorbuffer[1] = (BYTE)(ColorSum.g * 255);
                colorbuffer[0] = (BYTE)(ColorSum.b * 255);

                colorbuffer += 3;
            }

            colorbuffer += LineWidthSWidthM3;
        }
    }
}

bool CRayTracer::SetSamples(int Samples)
{
    if(this->Samples == Samples) return false;

    this->Samples = Samples;

    Resize(Width, Height);

    return true;
}

void CRayTracer::SwapBuffers(HDC hDC)
{
    if(ColorBuffer != NULL)
    {
        StretchDIBits(hDC, 0, 0, Width, Height, 0, 0, Width, Height, ColorBuffer,
&ColorBufferInfo, DIB_RGB_COLORS, SRCCOPY);
    }
}

bool CRayTracer::Shadow(void *Object, vec3 &Point, vec3 &LightDirection, float
LightDistance)
{
    for(CSphere *Sphere = Spheres; Sphere < LastSphere; Sphere++)
    {
        if(Sphere == Object) continue;

        if(Sphere->Intersect(Point, LightDirection, LightDistance))
        {
            return true;
        }
    }

    for(CQuad *Quad = Quads; Quad < LastQuad; Quad++)
    {
        if(Quad == Object) continue;

        if(Quad->Intersect(Point, LightDirection, LightDistance))
        {
            return true;
        }
    }

    return false;
}
```

```
vec3 CRayTracer::LightIntensity(void *Object, vec3 &Point, vec3 &Normal, vec3
&LightPosition, CLight *Light, float AO)
{
    vec3 LightDirection = LightPosition - Point;

    float LightDistance2 = length2(LightDirection);
    float LightDistance = sqrt(LightDistance2);

    LightDirection *= 1.0f / LightDistance;

    float Attenuation = Light->QuadraticAttenuation * LightDistance2 +
Light->LinearAttenuation * LightDistance + Light->ConstantAttenuation;

    float NdotLD = dot(Normal, LightDirection);

    if(NdotLD > 0.0f)
    {
        if(Light->Sphere)
        {
            if(Shadow(Object, Point, LightDirection, LightDistance) == false)
            {
                return Light->Sphere->Color * ((Light->Ambient * AO + Light->Diffuse
* NdotLD) / Attenuation);
            }
        }
        else
        {
            float LNdotLD = -dot(Light->Quad->N, LightDirection);

            if(LNdotLD > 0.0f)
            {
                if(Shadow(Object, Point, LightDirection, LightDistance) == false)
                {
                    return Light->Quad->Color * ((Light->Ambient * AO +
Light->Diffuse * NdotLD * LNdotLD) / Attenuation);
                }
            }
        }
    }

    return (Light->Sphere ? Light->Sphere->Color : Light->Quad->Color) *
(Light->Ambient * AO / Attenuation);
}
```

```
float TDRM = 2.0f / (float)RAND_MAX;

float CRayTracer::AmbientOcclusionFactor(void *Object, vec3 &Point, vec3 &Normal)
{
    float AO = 0.0f;

    for(int i = 0; i < GISamples; i++)
    {
        vec3 RandomRay = normalize(vec3(TDRM * (float)rand() - 1.0f, TDRM *
(float)rand() - 1.0f, TDRM * (float)rand() - 1.0f));

        float NdotRR = dot(Normal, RandomRay);

        if(NdotRR < 0.0f)
        {
            RandomRay = -RandomRay;
            NdotRR = -NdotRR;
        }

        float Distance = 1048576.0f, TestDistance;

        for(CSphere *Sphere = Spheres; Sphere < LastSphere; Sphere++)
        {
            if(Sphere == Object) continue;

            if(Sphere->Intersect(Point, RandomRay, Distance, TestDistance))
            {
                Distance = TestDistance;
            }
        }

        for(CQuad *Quad = Quads; Quad < LastQuad; Quad++)
        {
            if(Quad == Object) continue;

            if(Quad->Intersect(Point, RandomRay, Distance, TestDistance))
            {
                Distance = TestDistance;
            }
        }

        AO += NdotRR / (1.0f + Distance * Distance);
    }
```

```
	return 1.0f - AO * ODGISamplesMAmbientOcclusionIntensity;
}

float ODRM = 1.0f / (float)RAND_MAX;

void CRayTracer::IlluminatePoint(void *Object, vec3 &Point, vec3 &Normal, vec3 &Color)
{
	float AO = 1.0f;

	if(AmbientOcclusion)
	{
		AO = AmbientOcclusionFactor(Object, Point, Normal);
	}

	if(LightsCount == 0)
	{
		float NdotCD = dot(Normal, normalize(Camera.Position - Point));

		if(NdotCD > 0.0f)
		{
			Color *= 0.5f * (AO + NdotCD);
		}
		else
		{
			Color *= 0.5f * AO;
		}
	}
	else if(SoftShadows == false)
	{
		vec3 LightsIntensitiesSum;

		for(CLight *Light = Lights; Light < LastLight; Light++)
		{
			LightsIntensitiesSum += LightIntensity(Object, Point, Normal, Light->Sphere ?
Light->Sphere->Position : Light->Quad->m, Light, AO);
		}

		Color *= LightsIntensitiesSum;
	}
	else
	{
		vec3 LightsIntensitiesSum;

		for(CLight *Light = Lights; Light < LastLight; Light++)
```

```
        {
            if(Light->Sphere)
            {
                for(int i = 0; i < GISamples; i++)
                {
                    vec3 RandomRay = /*normalize(*/vec3(TDRM * (float)rand() - 1.0f, TDRM * (float)rand() - 1.0f, TDRM * (float)rand() - 1.0f)/*)*/;

                    vec3 RandomLightPosition = RandomRay * Light->Sphere->Radius + Light->Sphere->Position;

                    LightsIntensitiesSum += LightIntensity(Object, Point, Normal, RandomLightPosition, Light, AO);
                }
            }
            else
            {
                for(int i = 0; i < GISamples; i++)
                {
                    float s = ODRM * (float)rand();
                    float t = ODRM * (float)rand();

                    vec3 RandomLightPosition = Light->Quad->ab * s + Light->Quad->ad * t + Light->Quad->a;

                    LightsIntensitiesSum += LightIntensity(Object, Point, Normal, RandomLightPosition, Light, AO);
                }
            }
        }

        Color *= LightsIntensitiesSum * ODGISamples;
    }
}

float M_1_PI_2 = (float)M_1_PI * 0.5f;

vec3 CRayTracer::RayTrace(vec3 &Origin, const vec3 &Ray, UINT Depth, void *Object)
{
    RTData Data;

    for(CSphere *Sphere = Spheres; Sphere < LastSphere; Sphere++)
    {
        if(Sphere == Object) continue;
        if(Sphere->Intersect(Origin, Ray, Data.Distance, Data.TestDistance, Data.TestPoint))
        {
            Data.Point = Data.TestPoint;
            Data.Distance = Data.TestDistance;
            Data.Sphere = Sphere;
        }
    }

    for(CQuad *Quad = Quads; Quad < LastQuad; Quad++)
    {
        if(Quad == Object) continue;

        if(Quad->Intersect(Origin, Ray, Data.Distance, Data.TestDistance, Data.TestPoint))
        {
            Data.Point = Data.TestPoint;
            Data.Distance = Data.TestDistance;
            Data.Quad = Quad;
        }
    }

    for(CLight *Light = Lights; Light < LastLight; Light++)
    {
        if(Light->Sphere)
        {
            if(Light->Sphere->Intersect(Origin, Ray, Data.Distance, Data.TestDistance,
Data.TestPoint))
            {
                Data.Point = Data.TestPoint;
                Data.Distance = Data.TestDistance;
                Data.Light = Light;
            }
        }
        else
        {
            if(Light->Quad->Intersect(Origin, Ray, Data.Distance, Data.TestDistance,
Data.TestPoint))
            {
                Data.Point = Data.TestPoint;
                Data.Distance = Data.TestDistance;
                Data.Light = Light;
            }
        }
    }
    if(Data.Light)
    {
        Data.Color = Data.Light->Sphere ? Data.Light->Sphere->Color :
Data.Light->Quad->Color;
    }
    else if(Data.Quad)
    {
        Data.Color = Data.Quad->Color;

        if(Textures && Data.Quad->Texture)
        {
            float s = dot(Data.Quad->T, Data.Point) - Data.Quad->O.x;
            float t = dot(Data.Quad->B, Data.Point) - Data.Quad->O.y;

            Data.Color *= Data.Quad->Texture->GetColorBilinear(s, t);
        }

        IlluminatePoint(Data.Quad, Data.Point, Data.Quad->N, Data.Color);

        if(Data.Quad->Reflection > 0.0f)
        {
            vec3 ReflectedRay = reflect(Ray, Data.Quad->N);

            Data.Color = mix(Data.Color, RayTrace(Data.Point, ReflectedRay, Depth + 1,
Data.Quad), Data.Quad->Reflection);
        }

        /*if(Data.Quad->Refraction > 0.0f)
        {
            float Angle = -dot(Data.Quad->N, Ray);

            vec3 Normal;
            float Eta;

            if(Angle > 0.0f)
            {
                Normal = Data.Quad->N;
                Eta = Data.Quad->ODEta;
            }
            else
            {
                Normal = -Data.Quad->N;
                Eta = Data.Quad->Eta;
            }

            vec3 RefractedRay = refract(Ray, Normal, Eta);

            if(RefractedRay.x == 0.0f && RefractedRay.y == 0.0f && RefractedRay.z == 0.0f)
            {
                RefractedRay = reflect(Ray, Normal);
            }

            Data.Color = mix(Data.Color, RayTrace(Data.Point, RefractedRay, Depth + 1, Data.Quad), Data.Quad->Refraction);
        }*/
    }
    else if(Data.Sphere)
    {
        Data.Color = Data.Sphere->Color;

        vec3 Normal = (Data.Point - Data.Sphere->Position) * Data.Sphere->ODRadius;

        if(Textures && Data.Sphere->Texture)
        {
            float s = atan2(Normal.x, Normal.z) * M_1_PI_2 + 0.5f;
            float t = asin(Normal.y < -1.0f ? -1.0f : Normal.y > 1.0f ? 1.0f : Normal.y) * (float)M_1_PI + 0.5f;

            Data.Color *= Data.Sphere->Texture->GetColorBilinear(s, t);
        }

        IlluminatePoint(Data.Sphere, Data.Point, Normal, Data.Color);

        if(Data.Sphere->Refraction > 0.0f)
        {
            vec3 RefractedRay = refract(Ray, Normal, Data.Sphere->ODEta);

            vec3 L = Data.Sphere->Position - Data.Point;
            float LdotRR = dot(L, RefractedRay);
            float D2 = length2(L) - LdotRR * LdotRR;
            float Distance = LdotRR + sqrt(Data.Sphere->Radius2 - D2);

            vec3 NewPoint = RefractedRay * Distance + Data.Point;

            vec3 NewNormal = (Data.Sphere->Position - NewPoint) * Data.Sphere->ODRadius;
				RefractedRay = refract(RefractedRay, NewNormal, Data.Sphere->Eta);

				Data.Color = mix(Data.Color, RayTrace(NewPoint, RefractedRay, Depth + 1,
Data.Sphere), Data.Sphere->Refraction);
			}

			if(Data.Sphere->Reflection > 0.0f)
			{
				vec3 ReflectedRay = reflect(Ray, Normal);

				Data.Color = mix(Data.Color, RayTrace(Data.Point, ReflectedRay, Depth + 1,
Data.Sphere), Data.Sphere->Reflection);
			}
		}

		return Data.Color;
}

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

bool CMyRayTracer::InitScene()
{
	bool Error = false;

	Error |= !Floor.CreateTexture2D("floor.jpg");

	Error |= !Cube.CreateTexture2D("cube.jpg");

	Error |= !Earth.CreateTexture2D("earth.jpg");

	if(Error)
	{
		return false;
	}

	if(0) // Textured cube
	{
		QuadsCount = 6;

		Quads = new CQuad[QuadsCount];

		Quads[0] =  CQuad(vec3(-0.5f, -0.5f,  0.5f), vec3( 0.5f, -0.5f,  0.5f), vec3( 0.5f,
0.5f,  0.5f), vec3(-0.5f,  0.5f, 0.5f), vec3(1.0f, 1.0f, 1.0f), &Cube);
        Quads[1] =  CQuad(vec3( 0.5f, -0.5f, -0.5f), vec3(-0.5f, -0.5f, -0.5f), vec3(-0.5f,
0.5f, -0.5f), vec3( 0.5f,  0.5f,-0.5f), vec3(1.0f, 1.0f, 1.0f), &Cube);
        Quads[2] =  CQuad(vec3( 0.5f, -0.5f,  0.5f), vec3( 0.5f, -0.5f, -0.5f), vec3( 0.5f,
0.5f, -0.5f), vec3( 0.5f,  0.5f, 0.5f), vec3(1.0f, 1.0f, 1.0f), &Cube);
        Quads[3] =  CQuad(vec3(-0.5f, -0.5f, -0.5f), vec3(-0.5f, -0.5f,  0.5f), vec3(-0.5f,
0.5f,  0.5f), vec3(-0.5f,  0.5f,-0.5f), vec3(1.0f, 1.0f, 1.0f), &Cube);
        Quads[4] =  CQuad(vec3(-0.5f,  0.5f,  0.5f), vec3( 0.5f,  0.5f,  0.5f), vec3( 0.5f,
0.5f, -0.5f), vec3(-0.5f,  0.5f,-0.5f), vec3(1.0f, 1.0f, 1.0f), &Cube);
        Quads[5] =  CQuad(vec3(-0.5f, -0.5f, -0.5f), vec3( 0.5f, -0.5f, -0.5f), vec3( 0.5f, -
0.5f,  0.5f), vec3(-0.5f, -0.5f, 0.5f), vec3(1.0f, 1.0f, 1.0f), &Cube);
    }
    else if(0) // Textured sphere
    {
        SpheresCount = 1;

        Spheres = new CSphere[SpheresCount];

        Spheres[0] = CSphere(vec3(0.0f, 0.0f, 0.0f), 0.5f, vec3(1.0f, 1.0f, 1.0f), &Earth);
    }
    else if(0) // 100 random spheres
    {
        SpheresCount = 100;

        Spheres = new CSphere[SpheresCount];

        for(int i = 0; i < SpheresCount; i++)
        {
            #define rnd (float)rand() / (float)RAND_MAX

            vec3 Position = (vec3(rnd, rnd, rnd) * 2.0f - 1.0f) * 2.5f;
            float Radius = 0.125f + rnd * 0.25f;
            vec3 Color = vec3(rnd, rnd, rnd) * 2.0f;
            float Reflection = 0.25f + rnd * 0.5f;

            Spheres[i] = CSphere(Position, Radius, Color, NULL, Reflection);
        }
    }
    else if(0) // 50 random cubes
    {
        QuadsCount = 300;

        Quads = new CQuad[QuadsCount];
        for(int i = 0; i < QuadsCount; i += 6)
        {
            #define rnd (float)rand() / (float)RAND_MAX

            float S = 0.5f + rnd * 0.5f;
            vec3 T = (vec3(rnd, rnd, rnd) * 2.0f - 1.0f) * 2.5f;
            vec3 Color = vec3(rnd, rnd, rnd) * 2.0f;
            float Reflection = 0.25f + rnd * 0.5f;

            Quads[i + 0] =  CQuad(vec3(-0.5f, -0.5f,  0.5f) * S + T, vec3( 0.5f, -0.5f,  0.5f) * S + T, vec3( 0.5f,  0.5f,  0.5f) * S + T, vec3(-0.5f,  0.5f, 0.5f) * S + T, Color, NULL, Reflection);
            Quads[i + 1] =  CQuad(vec3( 0.5f, -0.5f, -0.5f) * S + T, vec3(-0.5f, -0.5f, -0.5f) * S + T, vec3(-0.5f,  0.5f, -0.5f) * S + T, vec3( 0.5f,  0.5f,-0.5f) * S + T, Color, NULL, Reflection);
            Quads[i + 2] =  CQuad(vec3( 0.5f, -0.5f,  0.5f) * S + T, vec3( 0.5f, -0.5f, -0.5f) * S + T, vec3( 0.5f,  0.5f, -0.5f) * S + T, vec3( 0.5f,  0.5f, 0.5f) * S + T, Color, NULL, Reflection);
            Quads[i + 3] =  CQuad(vec3(-0.5f, -0.5f, -0.5f) * S + T, vec3(-0.5f, -0.5f,  0.5f) * S + T, vec3(-0.5f,  0.5f,  0.5f) * S + T, vec3(-0.5f,  0.5f,-0.5f) * S + T, Color, NULL, Reflection);
            Quads[i + 4] =  CQuad(vec3(-0.5f,  0.5f,  0.5f) * S + T, vec3( 0.5f,  0.5f,  0.5f) * S + T, vec3( 0.5f,  0.5f, -0.5f) * S + T, vec3(-0.5f,  0.5f,-0.5f) * S + T, Color, NULL, Reflection);
            Quads[i + 5] =  CQuad(vec3(-0.5f, -0.5f, -0.5f) * S + T, vec3( 0.5f, -0.5f, -0.5f) * S + T, vec3( 0.5f, -0.5f,  0.5f) * S + T, vec3(-0.5f, -0.5f, 0.5f) * S + T, Color, NULL, Reflection);
        }
    }
    else if(0) // Cornell box
    {
        SpheresCount = 1;

        Spheres = new CSphere[SpheresCount];

        Spheres[0] = CSphere(vec3(0.0f, -1.5f, 1.0f), 0.5f, vec3(0.0f, 0.5f, 1.0f), NULL, 0.125f, 0.875f, 1.52f);

        QuadsCount = 17;

        Quads = new CQuad[QuadsCount + LightsCount];

        mat4x4 R = RotationMatrix(22.5f, vec3(0.0f, 1.0f, 0.0f));
        vec3 V = vec3(1.0f, 0.0f, 0.0f);
```

```
	Quads[0] =  CQuad(R * vec3(-0.5f, -2.0f,  0.5f) + V, R * vec3( 0.5f, -2.0f,  0.5f) + V, R * vec3( 0.5f, -1.0f,  0.5f) + V, R * vec3(-0.5f, -1.0f,  0.5f) + V, vec3(1.0f, 1.0f, 1.0f), &Cube);
	Quads[1] =  CQuad(R * vec3( 0.5f, -2.0f, -0.5f) + V, R * vec3(-0.5f, -2.0f, -0.5f) + V, R * vec3(-0.5f, -1.0f, -0.5f) + V, R * vec3( 0.5f, -1.0f, -0.5f) + V, vec3(1.0f, 1.0f, 1.0f), &Cube);
	Quads[2] =  CQuad(R * vec3( 0.5f, -2.0f,  0.5f) + V, R * vec3( 0.5f, -2.0f, -0.5f) + V, R * vec3( 0.5f, -1.0f, -0.5f) + V, R * vec3( 0.5f, -1.0f,  0.5f) + V, vec3(1.0f, 1.0f, 1.0f), &Cube);
	Quads[3] =  CQuad(R * vec3(-0.5f, -2.0f, -0.5f) + V, R * vec3(-0.5f, -2.0f,  0.5f) + V, R * vec3(-0.5f, -1.0f,  0.5f) + V, R * vec3(-0.5f, -1.0f, -0.5f) + V, vec3(1.0f, 1.0f, 1.0f), &Cube);
	Quads[4] =  CQuad(R * vec3(-0.5f, -1.0f,  0.5f) + V, R * vec3( 0.5f, -1.0f,  0.5f) + V, R * vec3( 0.5f, -1.0f, -0.5f) + V, R * vec3(-0.5f, -1.0f, -0.5f) + V, vec3(1.0f, 1.0f, 1.0f), &Cube);
	Quads[5] =  CQuad(R * vec3(-0.5f, -2.0f, -0.5f) + V, R * vec3( 0.5f, -2.0f, -0.5f) + V, R * vec3( 0.5f, -2.0f,  0.5f) + V, R * vec3(-0.5f, -2.0f,  0.5f) + V, vec3(1.0f, 1.0f, 1.0f), &Cube);

	Quads[6] =  CQuad(vec3(-2.0f, -2.0f,  2.0f), vec3( 2.0f, -2.0f,  2.0f), vec3( 2.0f, -2.0f, -2.0f), vec3(-2.0f, -2.0f, -2.0f), vec3(1.0f, 1.0f, 1.0f), &Floor, 0.0625f);
	Quads[7] =  CQuad(vec3(-2.0f,  2.0f, -2.0f), vec3( 2.0f,  2.0f, -2.0f), vec3( 2.0f,  2.0f,  2.0f), vec3(-2.0f,  2.0f,  2.0f), vec3(1.0f, 1.0f, 1.0f));
	Quads[8] =  CQuad(vec3(-2.0f, -2.0f, -2.0f), vec3( 2.0f, -2.0f, -2.0f), vec3( 2.0f,  2.0f, -2.0f), vec3(-2.0f,  2.0f, -2.0f), vec3(1.0f, 1.0f, 1.0f));
	Quads[9] =  CQuad(vec3( 2.0f, -2.0f,  2.0f), vec3(-2.0f, -2.0f,  2.0f), vec3(-2.0f,  2.0f,  2.0f), vec3( 2.0f,  2.0f,  2.0f), vec3(1.0f, 1.0f, 1.0f));
	Quads[10] = CQuad(vec3( 2.0f, -2.0f, -2.0f), vec3( 2.0f, -2.0f,  2.0f), vec3( 2.0f,  2.0f,  2.0f), vec3( 2.0f,  2.0f, -2.0f), vec3(0.0f, 1.0f, 0.0f));
	Quads[11] = CQuad(vec3(-2.0f, -2.0f,  2.0f), vec3(-2.0f, -2.0f, -2.0f), vec3(-2.0f,  2.0f, -2.0f), vec3(-2.0f,  2.0f,  2.0f), vec3(1.0f, 0.0f, 0.0f));

	Quads[12] = CQuad(vec3(-0.5f,  1.875f,  0.5f), vec3( 0.5f,  1.875f,  0.5f), vec3( 0.5f,  1.875f, -0.5f), vec3(-0.5f,  1.875f, -0.5f), vec3(1.0f, 1.0f, 1.0f));

	Quads[13] = CQuad(vec3(-0.5f,  1.875f - 0.125f,  0.5f), vec3( 0.5f,  1.875f - 0.125f,  0.5f), vec3( 0.5f,  2.0f - 0.125f,  0.5f), vec3(-0.5f,  2.0f - 0.125f,  0.5f), vec3(1.0f, 1.0f, 1.0f));
	Quads[14] = CQuad(vec3( 0.5f,  1.875f - 0.125f, -0.5f), vec3(-0.5f,  1.875f - 0.125f, -0.5f), vec3(-0.5f,  2.0f - 0.125f, -0.5f), vec3( 0.5f,  2.0f - 0.125f, -0.5f), vec3(1.0f, 1.0f, 1.0f));
	Quads[15] = CQuad(vec3(-0.5f,  1.875f - 0.125f, -0.5f), vec3(-0.5f,  1.875f - 0.125f,  0.5f), vec3(-0.5f,  2.0f - 0.125f,  0.5f), vec3(-0.5f,  2.0f - 0.125f, -0.5f), vec3(1.0f,
1.0f, 1.0f));
        Quads[16] = CQuad(vec3( 0.5f,  1.875f - 0.125f,  0.5f), vec3( 0.5f,  1.875f -
0.125f, -0.5f), vec3( 0.5f,  2.0f - 0.125f, -0.5f), vec3( 0.5f,  2.0f - 0.125f,  0.5f), vec3(1.0f,
1.0f, 1.0f));

        LightsCount = 1;

        Lights = new CLight[LightsCount];

        Lights[0].Quad = new CQuad(vec3(-0.5f,  1.875f - 0.125f, -0.5f), vec3( 0.5f,
1.875f - 0.125f, -0.5f), vec3( 0.5f,  1.875f - 0.125f,  0.5f), vec3(-0.5f,  1.875f - 0.125f,
0.5f), vec3(3.0f, 3.0f, 3.0f));
        Lights[0].Ambient = 0.25f;
        Lights[0].Diffuse = 0.75f;
        Lights[0].QuadraticAttenuation = 0.0625f;
    }
    else
    {
        SpheresCount = 3;

        Spheres = new CSphere[SpheresCount];

        Spheres[0] = CSphere(vec3(-2.0f, -1.0f,  2.0f), 0.5f, vec3(0.0f, 0.5f, 1.0f), NULL,
0.875f);
        Spheres[1] = CSphere(vec3( 0.0f, -1.5f,  2.0f), 0.5f, vec3(0.0f, 0.5f, 1.0f), NULL,
0.125f, 0.875f, 1.52f);
        Spheres[2] = CSphere(vec3( 2.0f, -1.5f, -2.0f), 0.5f, vec3(1.0f, 1.0f, 1.0f), &Earth);

        QuadsCount = 21;
        // QuadsCount = 31;

        Quads = new CQuad[QuadsCount];

        LightsCount = 1;
        // LightsCount = 2;
        // LightsCount = 3;

        Lights = new CLight[LightsCount];

        mat4x4 R = RotationMatrix(22.5f, vec3(0.0f, 1.0f, 0.0f));
        vec3 V = vec3(2.0f, 0.0f, 2.0f);

        Quads[0] =  CQuad(R * vec3(-0.5f, -2.0f,  0.5f) + V, R * vec3( 0.5f, -2.0f,  0.5f) +
V, R * vec3( 0.5f, -1.0f,  0.5f) + V, R * vec3(-0.5f, -1.0f,  0.5f) + V, vec3(1.0f, 1.0f, 1.0f),
&Cube);
		Quads[1] =  CQuad(R * vec3( 0.5f, -2.0f, -0.5f) + V, R * vec3(-0.5f, -2.0f, -0.5f) +
V, R * vec3(-0.5f, -1.0f, -0.5f) + V, R * vec3( 0.5f, -1.0f, -0.5f) + V, vec3(1.0f, 1.0f, 1.0f),
&Cube);
		Quads[2] =  CQuad(R * vec3( 0.5f, -2.0f,  0.5f) + V, R * vec3( 0.5f, -2.0f, -0.5f) +
V, R * vec3( 0.5f, -1.0f, -0.5f) + V, R * vec3( 0.5f, -1.0f,  0.5f) + V, vec3(1.0f, 1.0f, 1.0f),
&Cube);
		Quads[3] =  CQuad(R * vec3(-0.5f, -2.0f, -0.5f) + V, R * vec3(-0.5f, -2.0f,  0.5f) +
V, R * vec3(-0.5f, -1.0f,  0.5f) + V, R * vec3(-0.5f, -1.0f, -0.5f) + V, vec3(1.0f, 1.0f, 1.0f),
&Cube);
		Quads[4] =  CQuad(R * vec3(-0.5f, -1.0f,  0.5f) + V, R * vec3( 0.5f, -1.0f,  0.5f) +
V, R * vec3( 0.5f, -1.0f, -0.5f) + V, R * vec3(-0.5f, -1.0f, -0.5f) + V, vec3(1.0f, 1.0f, 1.0f),
&Cube);
		Quads[5] =  CQuad(R * vec3(-0.5f, -2.0f, -0.5f) + V, R * vec3( 0.5f, -2.0f, -0.5f) +
V, R * vec3( 0.5f, -2.0f,  0.5f) + V, R * vec3(-0.5f, -2.0f,  0.5f) + V, vec3(1.0f, 1.0f, 1.0f),
&Cube);

		Quads[6] =  CQuad(vec3(-0.0f, -2.0f,  4.0f), vec3( 4.0f, -2.0f,  4.0f), vec3( 4.0f, -
2.0f, -4.0f), vec3(-0.0f, -2.0f, -4.0f), vec3(1.0f, 1.0f, 1.0f), &Floor, 0.0625f);
		Quads[7] =  CQuad(vec3(-4.0f, -2.0f,  4.0f), vec3( 0.0f, -2.0f,  4.0f), vec3( 0.0f, -
2.0f,  0.0f), vec3(-4.0f, -2.0f,  0.0f), vec3(1.0f, 1.0f, 1.0f), &Floor, 0.0625f);
		Quads[8] =  CQuad(vec3( 0.0f,  2.0f, -4.0f), vec3( 4.0f,  2.0f, -4.0f), vec3( 4.0f,
2.0f,  4.0f), vec3( 0.0f,  2.0f,  4.0f), vec3(1.0f, 1.0f, 1.0f));
		Quads[9] =  CQuad(vec3(-4.0f,  2.0f,  0.0f), vec3( 0.0f,  2.0f,  0.0f), vec3( 0.0f,
2.0f,  4.0f), vec3(-4.0f,  2.0f,  4.0f), vec3(1.0f, 1.0f, 1.0f));
		Quads[10] = CQuad(vec3(-0.0f, -2.0f, -4.0f), vec3( 4.0f, -2.0f, -4.0f), vec3( 4.0f,
2.0f, -4.0f), vec3(-0.0f,  2.0f, -4.0f), vec3(1.0f, 1.0f, 1.0f));
		Quads[11] = CQuad(vec3( 4.0f, -2.0f,  4.0f), vec3(-4.0f, -2.0f,  4.0f), vec3(-4.0f,
2.0f,  4.0f), vec3( 4.0f,  2.0f,  4.0f), vec3(1.0f, 1.0f, 1.0f));
		Quads[12] = CQuad(vec3( 4.0f, -2.0f, -4.0f), vec3( 4.0f, -2.0f,  4.0f), vec3( 4.0f,
2.0f,  4.0f), vec3( 4.0f,  2.0f, -4.0f), vec3(0.0f, 1.0f, 0.0f));
		Quads[13] = CQuad(vec3(-4.0f, -2.0f,  4.0f), vec3(-4.0f, -2.0f, -0.0f), vec3(-4.0f,
2.0f, -0.0f), vec3(-4.0f,  2.0f,  4.0f), vec3(1.0f, 0.0f, 0.0f));
		Quads[14] = CQuad(vec3(-4.0f, -2.0f,  0.0f), vec3( 0.0f, -2.0f,  0.0f), vec3( 0.0f,
2.0f,  0.0f), vec3(-4.0f,  2.0f,  0.0f), vec3(1.0f, 1.0f, 1.0f));
		Quads[15] = CQuad(vec3( 0.0f, -2.0f,  0.0f), vec3( 0.0f, -2.0f, -4.0f), vec3( 0.0f,
2.0f, -4.0f), vec3( 0.0f,  2.0f,  0.0f), vec3(1.0f, 1.0f, 1.0f));

		vec3 S = vec3(-2.0f, 0.0f, 2.0f);

		Quads[16] = CQuad(vec3(-0.5f,  1.875f,  0.5f) + S, vec3( 0.5f,  1.875f,  0.5f) + S,
vec3( 0.5f,  1.875f, -0.5f) + S, vec3(-0.5f,  1.875f, -0.5f) + S, vec3(1.0f, 1.0f, 1.0f));
		Quads[17] = CQuad(vec3(-0.5f,  1.875f - 0.125f,  0.5f) + S, vec3( 0.5f,  1.875f -
0.125f,  0.5f) + S, vec3( 0.5f,  2.0f - 0.125f,  0.5f) + S, vec3(-0.5f,  2.0f - 0.125f,  0.5f) +
S, vec3(1.0f, 1.0f, 1.0f));
            Quads[18] = CQuad(vec3( 0.5f,    1.875f - 0.125f, -0.5f) + S, vec3(-0.5f,    1.875f -
0.125f, -0.5f) + S, vec3(-0.5f,    2.0f - 0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f, -0.5f) + S,
vec3(1.0f, 1.0f, 1.0f));
            Quads[19] = CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(-0.5f,    2.0f - 0.125f,    0.5f) + S, vec3(-0.5f,    2.0f - 0.125f, -0.5f) +
S, vec3(1.0f, 1.0f, 1.0f));
            Quads[20] = CQuad(vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3( 0.5f,    1.875f -
0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f,    0.5f) + S,
vec3(1.0f, 1.0f, 1.0f));

            Lights[0].Quad = new CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,
1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(3.0f, 3.0f, 3.0f));
            // Lights[0].Quad = new CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,
1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(1.0f, 1.0f, 1.0f));
            Lights[0].Ambient = 0.25f;
            Lights[0].Diffuse = 0.75f;
            // Lights[0].LinearAttenuation = 0.09375f;
            // Lights[0].LinearAttenuation = 0.125f;

            // Lights[1].Sphere = new CSphere(vec3( 2.0f, 0.0f, 2.0f), 0.03125f, vec3(1.0f, 1.0f,
1.0f));
            // Lights[1].Ambient = 0.25f;
            // Lights[1].Diffuse = 0.75f;
            // Lights[1].LinearAttenuation = 0.125f;

            /* Lights[0].Quad = new CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,
1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(3.0f, 0.0f, 0.0f));
            // Lights[0].Quad = new CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,
1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(1.0f, 1.0f, 1.0f));
            Lights[0].Ambient = 0.25f;
            Lights[0].Diffuse = 0.75f;
            Lights[0].LinearAttenuation = 0.09375f;
            // Lights[0].LinearAttenuation = 0.125f;

            S.x += 2.0f;

            Quads[21] = CQuad(vec3(-0.5f,    1.875f,    0.5f) + S, vec3( 0.5f,    1.875f,    0.5f) + S,
vec3( 0.5f,    1.875f, -0.5f) + S, vec3(-0.5f,    1.875f, -0.5f) + S, vec3(1.0f, 1.0f, 1.0f));
            Quads[22] = CQuad(vec3(-0.5f,    1.875f - 0.125f,    0.5f) + S, vec3( 0.5f,    1.875f -
0.125f,    0.5f) + S, vec3( 0.5f,    2.0f - 0.125f,    0.5f) + S, vec3(-0.5f,    2.0f - 0.125f,    0.5f) +
S, vec3(1.0f, 1.0f, 1.0f));
            Quads[23] = CQuad(vec3( 0.5f,    1.875f - 0.125f, -0.5f) + S, vec3(-0.5f,    1.875f -
0.125f, -0.5f) + S, vec3(-0.5f,    2.0f - 0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f, -0.5f) + S,
vec3(1.0f, 1.0f, 1.0f));
            Quads[24] = CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(-0.5f,    2.0f - 0.125f,    0.5f) + S, vec3(-0.5f,    2.0f - 0.125f, -0.5f) +
S, vec3(1.0f, 1.0f, 1.0f));
            Quads[25] = CQuad(vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3( 0.5f,    1.875f -
0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f,    0.5f) + S,
vec3(1.0f, 1.0f, 1.0f));

            Lights[1].Quad = new CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,
1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(0.0f, 3.0f, 0.0f));
            // Lights[1].Quad = new CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,
1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(1.0f, 1.0f, 1.0f));
            Lights[1].Ambient = 0.25f;
            Lights[1].Diffuse = 0.75f;
            Lights[1].LinearAttenuation = 0.09375f;
            // Lights[1].LinearAttenuation = 0.125f;

            S.x += 2.0f;

            Quads[26] = CQuad(vec3(-0.5f,    1.875f,    0.5f) + S, vec3( 0.5f,    1.875f,    0.5f) + S,
vec3( 0.5f,    1.875f, -0.5f) + S, vec3(-0.5f,    1.875f, -0.5f) + S, vec3(1.0f, 1.0f, 1.0f));
            Quads[27] = CQuad(vec3(-0.5f,    1.875f - 0.125f,    0.5f) + S, vec3( 0.5f,    1.875f -
0.125f,    0.5f) + S, vec3( 0.5f,    2.0f - 0.125f,    0.5f) + S, vec3(-0.5f,    2.0f - 0.125f,    0.5f) +
S, vec3(1.0f, 1.0f, 1.0f));
            Quads[28] = CQuad(vec3( 0.5f,    1.875f - 0.125f, -0.5f) + S, vec3(-0.5f,    1.875f -
0.125f, -0.5f) + S, vec3(-0.5f,    2.0f - 0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f, -0.5f) + S,
vec3(1.0f, 1.0f, 1.0f));
            Quads[29] = CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(-0.5f,    2.0f - 0.125f,    0.5f) + S, vec3(-0.5f,    2.0f - 0.125f, -0.5f) +
S, vec3(1.0f, 1.0f, 1.0f));
            Quads[30] = CQuad(vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3( 0.5f,    1.875f -
0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f, -0.5f) + S, vec3( 0.5f,    2.0f - 0.125f,    0.5f) + S,
vec3(1.0f, 1.0f, 1.0f));

            Lights[2].Quad = new CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,
1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(0.0f, 0.0f, 3.0f));
            // Lights[2].Quad = new CQuad(vec3(-0.5f,    1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,
1.875f - 0.125f, -0.5f) + S, vec3( 0.5f,    1.875f - 0.125f,    0.5f) + S, vec3(-0.5f,    1.875f -
0.125f,    0.5f) + S, vec3(1.0f, 1.0f, 1.0f));
        Lights[2].Ambient = 0.25f;
        Lights[2].Diffuse = 0.75f;
        Lights[2].LinearAttenuation = 0.09375f;
        // Lights[2].LinearAttenuation = 0.125f; */
    }

    Camera.LookAt(vec3(0.0f), vec3(0.0f, 0.0f, 8.75f), true);
    // Camera.LookAt(vec3(0.0f), rotate(vec3(0.0f, 0.0f, 8.75f), 45.0f, vec3(0.0f, 1.0f, 0.0f)),
true);
    // Camera.LookAt(vec3(0.0f), rotate(vec3(0.0f, 0.0f, 8.75f), 135.0f, vec3(0.0f, 1.0f, 0.0f)),
true);
    // Camera.LookAt(vec3(0.0f), rotate(vec3(0.0f, 0.0f, 8.75f), 67.5f, vec3(0.0f, 1.0f, 0.0f)),
true);

    return true;
}

void CMyRayTracer::DestroyTextures()
{
    Floor.Destroy();
    Cube.Destroy();
    Earth.Destroy();
}

CMyRayTracer RayTracer;

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

CWnd::CWnd()
{
    char *moduledirectory = new char[256];
    GetModuleFileName(GetModuleHandle(NULL), moduledirectory, 256);
    *(strrchr(moduledirectory, '\\') + 1) = 0;
    ModuleDirectory = moduledirectory;
    delete [] moduledirectory;
}

CWnd::~CWnd()
{
}
```

```
bool CWnd::Create(HINSTANCE hInstance, char *WindowName, int Width, int Height)
{
    WNDCLASSEX WndClassEx;

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

    WndClassEx.cbSize = sizeof(WNDCLASSEX);
    WndClassEx.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
    WndClassEx.lpfnWndProc = WndProc;
    WndClassEx.hInstance = hInstance;
    WndClassEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    WndClassEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    WndClassEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    WndClassEx.lpszClassName = "Win32CPURayTracerWindow";

    if(RegisterClassEx(&WndClassEx) == 0)
    {
        ErrorLog.Set("RegisterClassEx failed!");
        return false;
    }

    this->WindowName = WindowName;

    this->Width = Width;
    this->Height = Height;

    DWORD Style = WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN;

    if((hWnd = CreateWindowEx(WS_EX_APPWINDOW, WndClassEx.lpszClassName,
WindowName, Style, 0, 0, Width, Height, NULL, NULL, hInstance, NULL)) == NULL)
    {
        ErrorLog.Set("CreateWindowEx failed!");
        return false;
    }

    if((hDC = GetDC(hWnd)) == NULL)
    {
        ErrorLog.Set("GetDC failed!");
        return false;
    }

    return RayTracer.Init();
}
```

```
void CWnd::RePaint()
{
	Line = 0;
	InvalidateRect(hWnd, NULL, FALSE);
}

void CWnd::Show(bool Maximized)
{
	RECT dRect, wRect, cRect;

	GetWindowRect(GetDesktopWindow(), &dRect);
	GetWindowRect(hWnd, &wRect);
	GetClientRect(hWnd, &cRect);

	wRect.right += Width - cRect.right;
	wRect.bottom += Height - cRect.bottom;

	wRect.right -= wRect.left;
	wRect.bottom -= wRect.top;

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

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

	ShowWindow(hWnd, Maximized ? SW_SHOWMAXIMIZED : SW_SHOWNORMAL);
}

void CWnd::MsgLoop()
{
	MSG Msg;

	while(GetMessage(&Msg, NULL, 0, 0) > 0)
	{
		TranslateMessage(&Msg);
		DispatchMessage(&Msg);
	}
}

void CWnd::Destroy()
{
	RayTracer.Destroy();

	DestroyWindow(hWnd);
}

void CWnd::OnKeyDown(UINT Key)
{
	switch(Key)
	{
		case '1':
			if(RayTracer.SetSamples(1)) RePaint();
			break;

		case '2':
			if(RayTracer.SetSamples(2)) RePaint();
			break;

		case '3':
			if(RayTracer.SetSamples(3)) RePaint();
			break;

		case '4':
			if(RayTracer.SetSamples(4)) RePaint();
			break;

		case VK_F1:
			RayTracer.Textures = !RayTracer.Textures;
			RePaint();
			break;

		case VK_F2:
			RayTracer.SoftShadows = !RayTracer.SoftShadows;
			RePaint();
			break;

		case VK_F3:
			RayTracer.AmbientOcclusion = !RayTracer.AmbientOcclusion;
			RePaint();
			break;
	}

	if(Camera.OnKeyDown(Key))
	{
		RePaint();
	}
}
```

```
void CWnd::OnMouseMove(int cx, int cy)
{
	if(GetKeyState(VK_RBUTTON) & 0x80)
	{
		Camera.OnMouseMove(LastCurPos.x - cx, LastCurPos.y - cy);

		LastCurPos.x = cx;
		LastCurPos.y = cy;

		RePaint();
	}
}

void CWnd::OnMouseWheel(short zDelta)
{
	Camera.OnMouseWheel(zDelta);

	RePaint();
}

void CWnd::OnPaint()
{
	PAINTSTRUCT ps;

	BeginPaint(hWnd, &ps);

	static DWORD Start;
	static bool RayTracing;

	if(Line == 0)
	{
		RayTracer.ClearColorBuffer();

		// CQuad::Intersections = 0;
		// CSphere::Intersections = 0;

		Start = GetTickCount();

		RayTracing = true;
	}

	DWORD start = GetTickCount();

	while(Line < Height && GetTickCount() - start < 250)
```

```
    {
        RayTracer.RayTrace(Line++);
    }

    RayTracer.SwapBuffers(hDC);

    if(RayTracing)
    {
        if(Line == Height)
        {
            RayTracing = false;
            RayTracer.MapHDRColors();
        }

        DWORD End = GetTickCount();

        CString text = WindowName;

        text.Append(" - %dx%d", Width, Height);
        text.Append(", Supersampling %dx", RayTracer.GetSamples());
        text.Append(", Time: %.03f s", (float)(End - Start) * 0.001f);
        // text.Append(", %.02f / %.02f mil. (quad / sphere) intersection tests", (float)CQuad::Intersections * 0.000001f, (float)CSphere::Intersections * 0.000001f);

        SetWindowText(hWnd, text);

        InvalidateRect(hWnd, NULL, FALSE);
    }

    EndPaint(hWnd, &ps);
}

void CWnd::OnRButtonDown(int cx, int cy)
{
    LastCurPos.x = cx;
    LastCurPos.y = cy;
}

void CWnd::OnSize(int Width, int Height)
{
    this->Width = Width;
    this->Height = Height;

    RayTracer.Resize(Width, Height);
    RePaint();
}

CWnd Wnd;

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

LRESULT CALLBACK WndProc(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM
lParam)
{
    switch(uiMsg)
    {
        case WM_CLOSE:
            PostQuitMessage(0);
            break;

        case WM_MOUSEMOVE:
            Wnd.OnMouseMove(LOWORD(lParam), HIWORD(lParam));
            break;

        case 0x020A: // WM_MOUSWHEEL
            Wnd.OnMouseWheel(HIWORD(wParam));
            break;

        case WM_KEYDOWN:
            Wnd.OnKeyDown((UINT)wParam);
            break;

        case WM_PAINT:
            Wnd.OnPaint();
            break;

        case WM_RBUTTONDOWN:
            Wnd.OnRButtonDown(LOWORD(lParam), HIWORD(lParam));
            break;

        case WM_SIZE:
            Wnd.OnSize(LOWORD(lParam), HIWORD(lParam));
            break;

        default:
            return DefWindowProc(hWnd, uiMsg, wParam, lParam);
```

```
	}

	return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR sCmdLine,
int iShow)
{
	SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);

	if(Wnd.Create(hInstance, "CPU ray tracer", 800, 600))
	{
		Wnd.Show();
		Wnd.MsgLoop();
	}
	else
	{
		MessageBox(NULL, ErrorLog, "Error", MB_OK | MB_ICONERROR);
	}

	Wnd.Destroy();

	return 0;
}
```