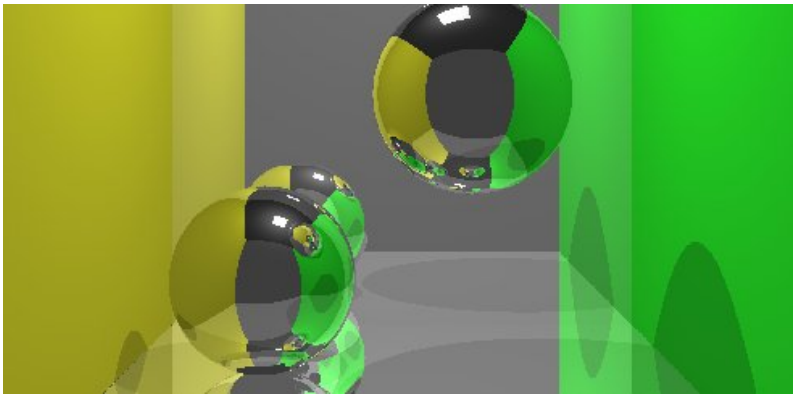
-



Real time GPU Ray tracer

SPECIAL TECHNIQUES FOR GRAPHICS AND ANIMATION

Shanaka Senevirathne

Contents

Abstract

This documentation provides an insight to the implementation of a real-time GPU based Ray tracer on current generation consumer level GPUs. It is implemented using DirectX 9, Direct3D High Level Shading Language (HLSL) and C++.

This documentation will also briefly discuss an introduction to what is Ray tracing, the state-of-the-art of real-time GPU based ray traces, the design strategy used to construct the Ray tracer, description of the implementation and followed by future improvements that could be made to this implementation.

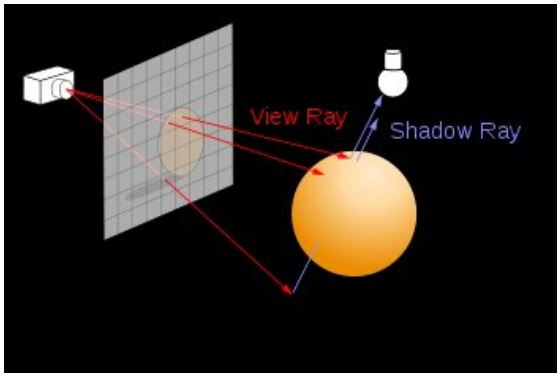
Introduction

In the field of computer graphics, mimicking the real world attributes as close as possible inside the computer has been a benchmarking standard. And in video games particularly, making things as realistic as possible has been a major motivation for developers to find new and existing ways to represent real world attributes and lighting inside their programs.

Ray tracing was traditionally done on CPUs and is heavily used in Computer Animation field. But it is highly processor intensive and to model a moderate scene with some geometric primitives on the CPU will give you very low frame rate which is unacceptable where games are concerned. Thus, the idea of implementing Ray tracing on a GPU (Graphical Processing Unit) was explored. This yielded some pretty amazing frame rates and also freed up the CPU to do other tasks such as doing AI routes or calculating physics in a gaming environment.

What is Ray tracing?

Ray tracing is basically an attempt to imitate nature. Light is cast by a light source, bounces around the scene and finally hits the viewer's eye. But calculating each and every ray coming into the scene from the light source is impractical because there are too many rays to consider and not all of them do affect the objects in the scene. Those rays are the reason that a ray tracer does not trace rays from light source to a camera but vice versa. Below image shows this concept in a diagram.



By using Ray tracing, a much more effective kind of photorealism can be archived with realistic reflections, refractions, depth of field, motion blur and etc.

What is the state of the art in Ray tracing?

When I was doing my initial research on the subject of real time ray tracing, I came across several research papers which discusses the benefits of ray tracing over rasterization-based algorithms.

And one of those benefits is that when you go to extremes, ray tracing is actually faster than rasterizing. And they prove it. Imagine a huge scene, consisting of, say, 50 million triangles. Toss it at a recent GeForce graphics card with enough memory to store all those triangles, and write down the frame rate. It will be in the vicinity of 2-5 fps. Now, ray trace the same scene. The test report 8 frames per second on a dual PIII/800. Make that a quad PIII/800 and the speed doubles. Ray tracing scales linearly with processing power, but only logarithmically with scene complexity. And on top of this, the advantage and ability to archive much more photorealistic reflections, shadows, refractions, motion blur, depth of field effects, Global illumination and all sorts of other effects are staggering.

Although the current generation of games does not uses heavy use of ray tracing, with the introduction of DirectX 11 which is suppose to have ray tracing component in it will surely grab the attention of the next generation game programmers. And thus it will lead to more photorealistic graphics, effects and games.

Programming on GPU

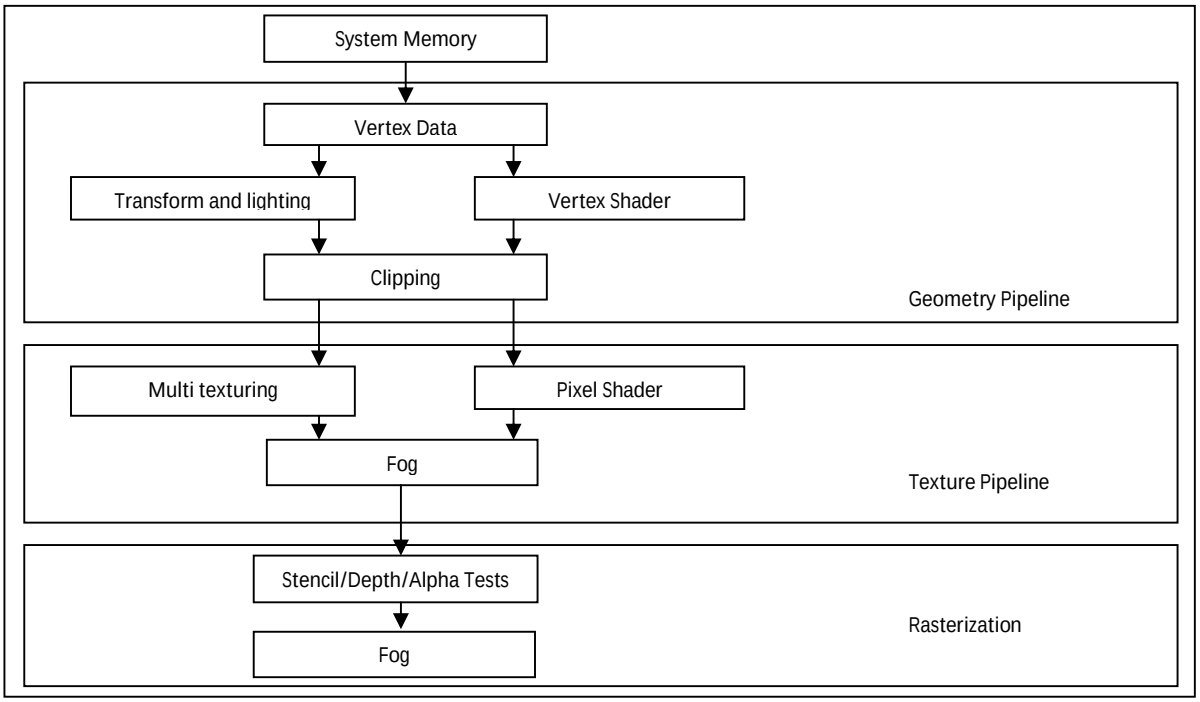
Traditionally graphics programmers were programming to the CPU and they had no control over how the vertices and pixels were processed under the hood after it was send to be drawn. This was called Fixed-function pipeline. Programmers wanted much more control over how things happen and drawn, so in reply to that, graphics APIs like DirectX and OpenGL supplied the programmers with a Programmable Graphics pipeline, where the programmer has much more control over how the vertices and pixels were rendered.

This was done using shaders (vertex shaders and pixel shaders). And they introduced it through High Level Shading Language (HLSL) for DirectX and OpenGL Shading Language (GLSL) for OpenGL. The syntax is much similar to the syntax of C and can also be coded in assembly if the programmer wishes.

These shaders do not reside in the source code of the program but outside of it. Shaders are a way to instruct the Graphical Processing Unit (GPU) to how to and what to do with the data that is passed into it. I have focused on DirectX and HLSL for this implementation.

Programmable Graphics Hardware Pipeline

A pipeline is a sequence of steps operating in a parallel and a fixed order. Each stage receives its input from the prior stage and sends its outputs to the subsequent stage.



Design Strategy

Classical recursive Ray tracer algorithm

```
For each pixel in image
{
  Create ray from eyepoint passing through this pixel
  Initialize NearestT to INFINITY and NearestObject to NULL

  For every object in scene
  {
     If ray intersects this object
     {
        If t of intersection is less than NearestT
        {
           Set NearestT to t of the intersection
           Set NearestObject to this object
        }
     }
  }

  If NearestObject is NULL
  {
     Fill this pixel with background color
  }
  Else
  {
     Shoot a ray to each light source to check if in shadow
     If surface is reflective, generate reflection ray: recurse
     If surface is transparent, generate refraction ray: recurse
     Use NearestObject and NearestT to compute shading function
     Fill this pixel with color result of shading function
  }
}
```

Implementation

There is two parts to this project

1. Main source file which runs on CPU (Raytracer.cpp)

   This is where all the setting up of the camera and geometry happens.

2. Shader / Effect file which runs on GPU (Raytracer.fx)

   This is where the actual ray tracing is done with the support of vertex and pixel shaders.

Main source file which runs on CPU (Raytracer.cpp)

I used the DXUT utility framework provided by Microsoft to make the initial setting up of a DirectX application easy. But I did not use any specific help from that like making use of full screen mode and such things.

- Then I have a couple of custom structures defined. One to hold vertex information.

```
struct CustomVertex
{
        float x, y, z;
        float u, v;
};
```

- Another to hold texture information.

```
struct TextureFormat
{
        float r, g, b, a;
};
```

- Finding and creation of the shader from the effect file (Raytracer.fx) happens inside the DXUT callback method `OnD3D9CreateDevice`.
- I'm creating the vertex buffer inside DXUT callback method `OnD3D9ResetDevice`. It is also where the projection parameters of the camera is been set.
- Most of the action happens onside DXUT callback method OnFrameMove. This is where I create the spheres and rectangles for my scene and every frame the two spheres animate accordingly. This is also where I pass the values from the .cpp file into the .fx file. I only pass 9 parameters all together. For example this is how I set the values

```
g_pEffect9->SetInt( "g_nSphereNum", nSphereNum )
```

Like wise I pass along the World matrix, View Projection Inverse matrix, camera position, number of rectangles, number of spheres, ambient color, light position, light color and scene texture.

- Since I need to access the vertex and pixel information of the whole screen inside the .fx file and there are a lot of data to be passed on if done manually, I put all the Rectangle (floors, walls) and

sphere information into a texture and pass that to the .fx file. That way I could access that information pixel by pixel inside the fx file. This method has been used a lot in a number of GLSL implementations of ray traces and is encouraged to do so rather than passing hundreds of parameters across.

Shader / Effect file which runs on GPU (Raytracer.fx)

- Number of iteration the ray tracer is traced is limited by this value

```
#define MAX_TRACE_LEVEL 2
```

- Vertex shader output structure

```
struct VS_OUTPUT
{
	float4 Position : POSITION; // vertex position
	float4 Diffuse : COLOR0; // vertex diffuse color
	float2 TextureUV : TEXCOORD0; // vertex texture coords
	float3 worldPos : TEXCOORD1; // world position
};
```

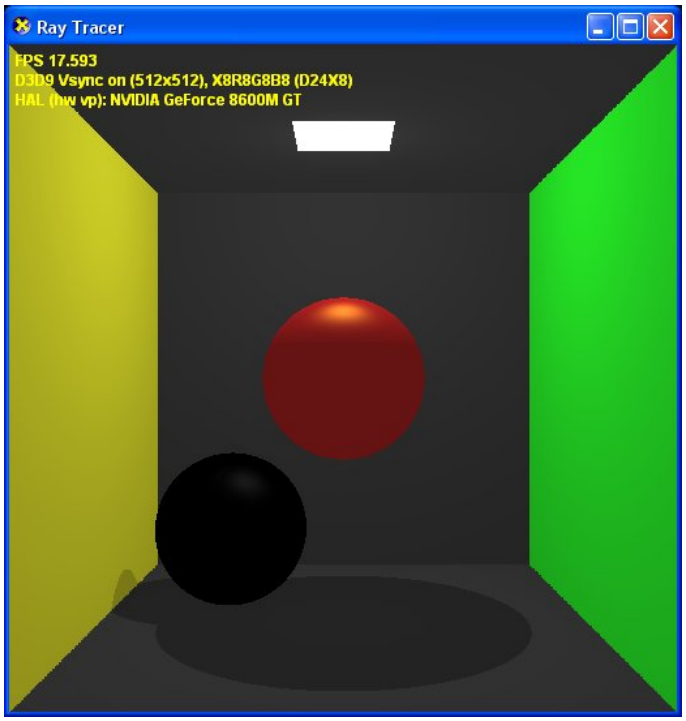
- Pixel shader output structure

```
struct PS_OUTPUT
{
	float4 RGBColor : COLOR0; // Pixel color
};
```

- I have `intersectRectangle` and `intersectSphere` methods to calculate the intersection of the rays send by the point of view of the camera to the rectangles and the spehers.
- Most of the action happpens inside the `trace` method. This is where all the rest of the functions get called from like `intersectRectangle, intersectSphere` and etc.
- Finally the scene is renderd by calling `RenderScenePS` inside the `RenderScene` techniques first pass.

Screenshots

2 rays

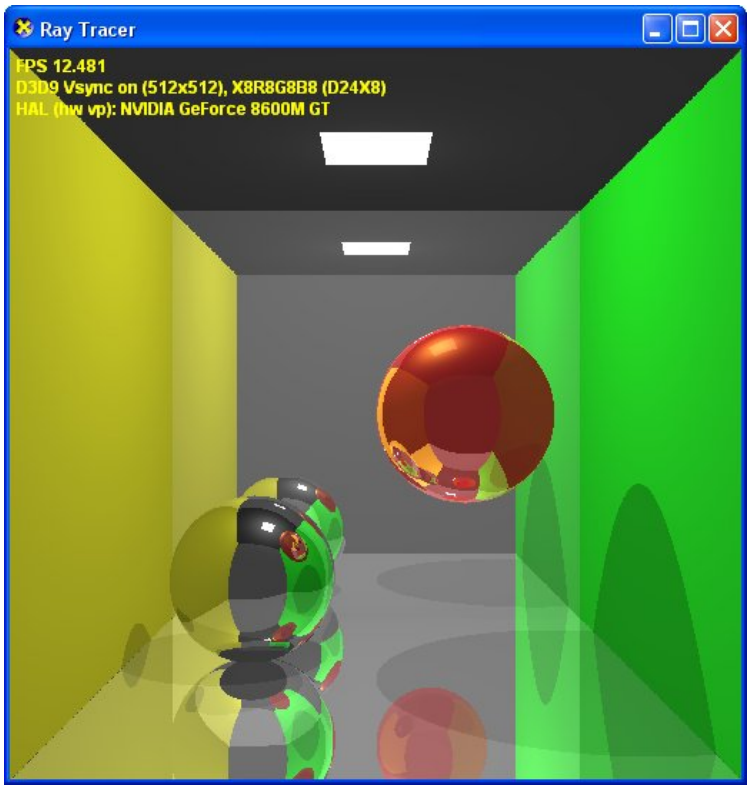


3 rays

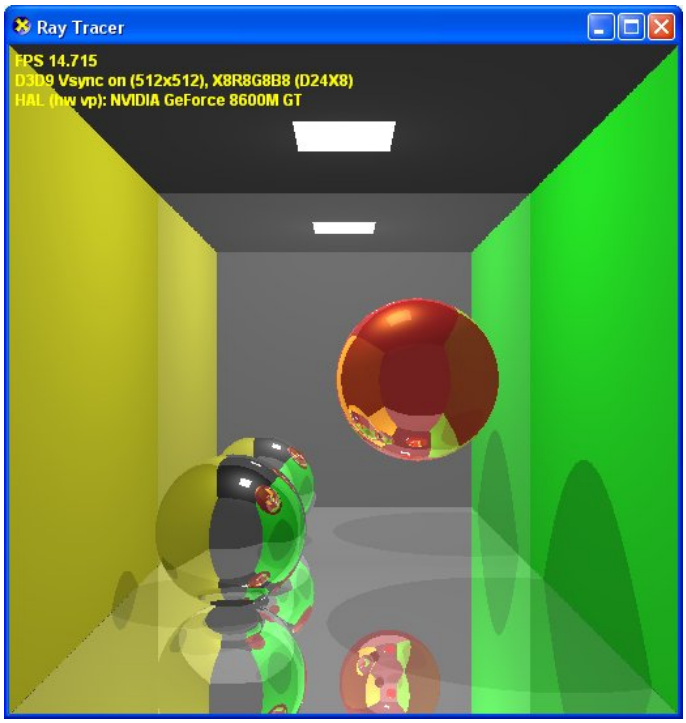
4 rays

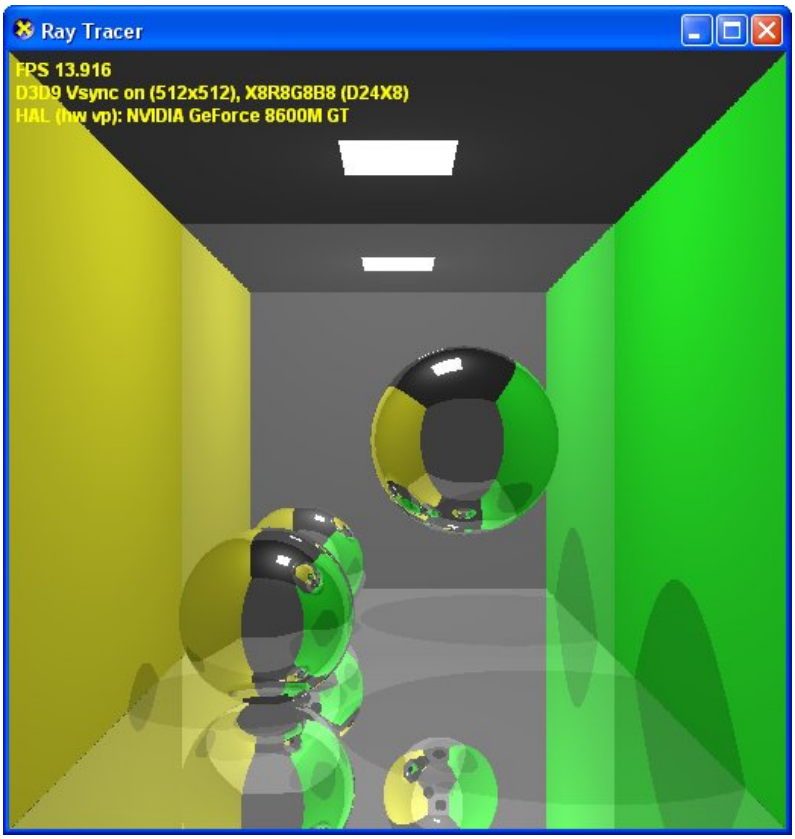

Ray Tracer
FPS 17.212
D3D9 Vsync on (512x512), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce 8600M GT

5 rays


Ray Tracer
FPS 12.481
D3D9 Vsync on (512x512), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce 8600M GT

6 rays


Ray Tracer
FPS 14.715
D3D9 Vsync on (512x512), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce 8600M GT


Ray Tracer
FPS 13.916
D3D9 Vsync on (512x512), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce 8600M GT

Future improvements

- Adding sliders like GUI to control the lighting.
- Adding click and move interaction for spheres.
- Adding multi target / multi mesh rendering.
- Implementing a suitable acceleration structure.

References

http://www.devmaster.net/articles/raytracing_series/part1.php

http://en.wikipedia.org/wiki/Ray_tracing_(graphics)#Disadvantages

http://www.groovyvis.com/other/raytracing/index.html

http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm

http://www.clockworkcoders.com/oglsl/rt/index.html

http://www.geocities.com/jamisbuck/raytracing.html

http://www.flipcode.com/archives/Raytracing_Topics_Techniques-Part_1_Introduction.shtml

http://www.masm32.com/board/index.php?PHPSESSID=181c40a5b8c4e197d0acddb7738182c8&topic=9993.0

http://www.cs.utah.edu/~reinhard/papers/egwr2k.pdf

http://graphics.stanford.edu/papers/gpu_kdtree/kdtree.pdf

http://igad.nhtv.nl/~bikker/

http://www.coniserver.net/wiki/index.php/HLSL_Basics

Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach (Wordware Game and Graphics Library)

Real Time Rendering Tricks and Techniques in DirectX 8 (Premier Press Game Development)