RAY TRACING ON GPU

Performance comparison between the CPU and the Compute Shader with DirectX 11

By:
Gustav Persson
Jonathan Udd

University Advisor:
Olle Lindeberg

2010

Contact Information:
Authors:
Gustav Persson
Address: Fogdevägen 4B 371 40 Karlskrona
E-mail: gustav.e.persson@gmail.com

Jonathan Udd
Address: Gyllenstjärnas Väg 5B 371 40 Karlskrona
E-mail: jonathan.udd@gmail.com

University advisor:
Olle Lindeberg
School of Computing
Address: Campus Gräsvik, Karlskrona
Phone: +46 455 38 58 30

Abstract

The game industry have always looked for rendering techniques that makes the games as good looking and realistic as possible. The common approach is to use triangles built up by vertices and apply many different techniques to make it look as good as possible. When triangles are used to draw objects, there is always edges and those edges often make the objects look less realistic than desired. To reduce these visible edges the amount of triangles for an object have to be increased, but with more triangles more processing power from the graphics cards is needed.

Another way to approach rendering is ray tracing which can render an extremely photo realistic image but to the cost of unbearable low performance if you would use it in a realtime application. The reason raytracing is so slow is the massive amount of calculations that needs to be made. In DirectX 11 a few new shaders where announced and one of them were the compute shader, the compute shader allows you to calculate data on the graphics card which is not bound to the pipeline. The compute shader allows you to use the hundreds of cores that the graphic card has and is therefore well suited for a raytracing algorithm.

One application is used to see if the hypothesis is correct. A flag is used to define if the application runs on the CPU and the GPU. The same algorithm is used in both versions. Three test where done on each processing unit to confirm the hypothesis. Three more tests where done on the GPU to see how the performance scaled on the GPU depending on the number of rendered objects. The tests proved throughout that the compute shader performs considerably better than the CPU when running our ray tracing algorithm.

Keywords

Ray Tracing, Compute shader, DirectX, Direct3D, Rendering, Real-time

TABLE OF CONTENTS

INTRODUCTION

Background

Ray Tracing is a graphics rendering algorithm that can be used to render incredible photorealistic images, although at a rather steep performance cost. Because of this, Ray Tracing has so far mostly been used with great results in the movie and TV industry to create special effects, environments and other things that would, for various reasons, not be feasible to create in reality. We will explain the algorithm in further detail in a later section.

The Computer Gaming industry has long been known to push the boundaries of what's possible to do with real time applications on main stream computers. With the new edition of Microsoft's graphics library DirectX it is possible to use the Graphics Processing Unit (GPU) to conduct computations normally handled by the Central Processing Unit (CPU). This might mean that the industry will move to a more Ray Tracing based rendering base, as opposed to the regular raster rendering.

This thesis' purpose is to test if a simple ray tracer can be implemented with the use of the compute shader, and to compare it with the same algorithm running on the CPU, to see if modern graphic cards can perform real time application rendering with ray tracing better than modern CPUs.

Methodology

The focus of this thesis is to see how the performance of a ray tracing algorithm is changed when executed on the GPU compared to when executed on the CPU. We will analyze the algorithm in real time to see how it performs.

The benchmarking algorithm is developed by us to work on both the CPU and GPU with minimum differences between the two.

Hypothesis

The Compute Shader will significantly increase ray tracing performance over the CPU.

The hypothesis has been reached by researching threading capabilities of the GPU versus capabilities of the CPU, and the extremely threading-friendly ray tracing algorithm.

Research Objectives

This thesis aims to examine performance and result differences between ray tracing algorithms executed on the CPU and GPU, to see if, and if so how much, the new DirectX 11 compute shader increases performance.

Delimitations

The algorithm has been kept extremely simple. We only cast primary rays, we will get to what this means later, and therefore ignore shadows, refractions and so on. We also only test against one kind of primitives, spheres. Focus stays on seeing if it's possible to ray trace at all with the requirements specified.

All tests are implemented with the latest rendering software, Direct3D 11, which means that solutions that other rendering software, such as OpenGL, is not covered at all.

All rays cast are tested for collisions with every object in the scene, and therefore not exactly as optimized as they could be, and performance can be expected to increase if the algorithm were to be fully optimized.

Acknowledgements

We would like to thank Stefan Peterson at BTH for giving us the idea to try out the Compute Shader. We would also like to thank our advisor Olle Lindeberg for all his help and advice.

RAY TRACING

How it works

Introduction

Ray tracing has long been used to create photorealistic sceneries and effects, see image 1, mostly in movies [Christensen, et. al. 2006] and TV-shows and so on. The algorithm is, however, very computationally expensive, but the actual theory is quite simple. This chapter will give a short description about the ray tracing algorithm and how it works.

Oyonale.com, Scene with glasses: Ray traced scene.

Approaches

The first ray tracing algorithm was developed by Turner Whitted, who in 1980 wrote a paper detailing the first version of the algorithm [Whitted, 1980]. The algorithm simulates how rays of light pass through a scene and how it reacts to the objects in it, and tries to behave as rays of light would in real life.

Forward Ray Tracing

Forward Ray Tracing is the term for what happens in nature. A number of light rays are emitted from a light source, in this case we'll say the sun. These rays will travel in a straight line until they hit something that interrupts it's progress. At this point, any combination of the following four things might happen: absorption, reflection, refraction and fluorescence. The surface might reflect the ray, or just a part of it, in one or more directions. It can also absorb a faction of it, resulting in the loss of intensity in the light ray. If the surface is transparent or

translucent, it refracts a portion of the ray to pass through it in a new direction, and possibly also with a new color. Eventually some of these rays will hit the viewer's eye, allowing him to see the surface. Because of the massive amounts of rays you need to spawn if you would use Forward ray tracing in rendering and as an extension of that the massive amounts of calculations to check for collisions, this is obviously not an alternative.

Backward ray tracing

In *Backward Ray Tracing* the rays spawn from the eye, in this case the camera, and travel outwards towards the light source, giving the method it's name since photons travel the other way in real life. This guarantees that the ray will always hit the "eye" and vastly reduces calculations needed. The algorithm spawns one ray through each pixel, called *Primary Rays,* which are then tested for collisions against all objects in the scene. When a collision is found, you sample the impact point's color, and spawn a number of different kinds of secondary rays, depending on the material. There are three commonly used kinds of secondary rays, called reflective, refractive and shadow rays.

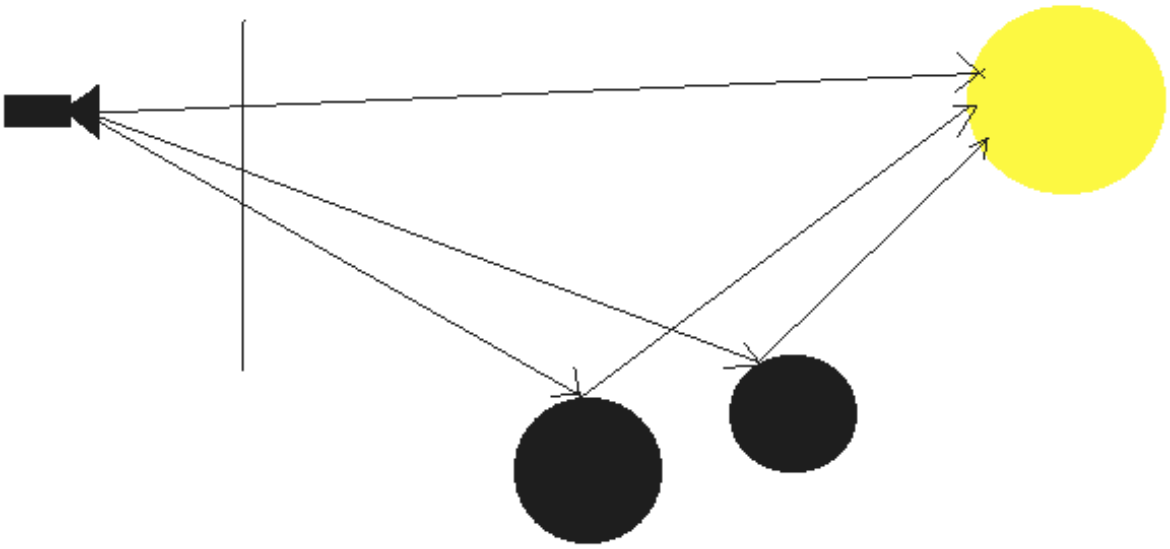


Image 2: Illustration of backward ray tracing, with shadow rays and view plane

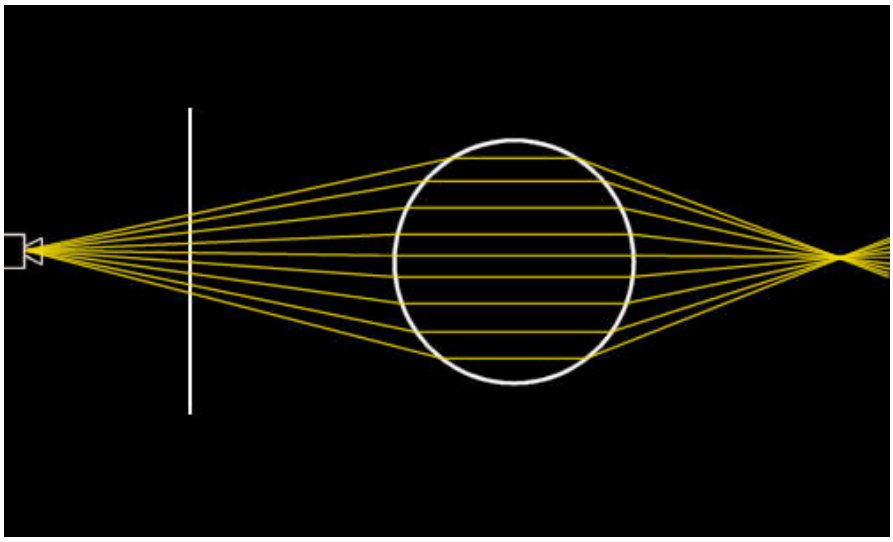
Shadow Rays

Shadow Rays are used to determine if the surface the primary ray hit is shadowed by another object. The theory behind it is quite simple; for each intersection point for a ray, you spawn one or more rays traveling towards the scene's light sources. If the shadow ray hits an opaque object before it reaches the light source, then the original impact point must be in shadow. If the ray hits nothing, the original point is lighted. This approach will, however, create shadows with so called "sharp" edges, which does not look very realistic. If you take a look at the edges of shadows in real life, you will notice that they are blurred. This is called *Soft Shadows* {Bikker,

2005} and is achieved by not limiting yourself to sending one shadow ray per primary ray, but by instead sending multiple ones, directed at different parts of the light source. This way, you will be able to determine if the impact point is partly shaded, by taking the amount of shadow rays reaching they're destination and comparing that to the total amount of shadow rays. Of course, other solutions for this problem does exist.

Refractive Rays

Refractive Rays are rays that are created when the surface that's hit has some kind of transparency. The secondary ray travels through the surface in a new direction, which is decided by the surface's refraction index. When the refractive ray hits another surface, the new point is rendered where the refractive ray originated from, sometimes with a new color (see fluids in image 1), allowing you to see through the first surface.



Bicco, J: Refraction through a sphere

Reflective Rays

Reflective Rays are quite simply rays that are reflected by the surface in a mirrored direction. After the ray is reflected, it checks for collisions with other objects and, with the original surface's reflectiveness value, decides how much of the hit object will be shown on the original surface, creating a mirror image. For example, mirrors have a high degree of reflectivity, allowing you to see objects reflected in detail on it, while brick walls have a very low degree, reflecting only very rough shapes.

THE COMPUTE SHADER

In order to compare the ray tracing algorithm's performance between the CPU and GPU, we need to implement code for the Direct3D Compute shader, allowing it to execute on the GPU.

The compute shader is a shader stage, introduced in Direct3D 11, which allows the programmer to use the GPU for a more common purpose then what's been possible to do before now. The GPU has hundreds of processor cores, compared to the CPU which only has a few, and is therefore very well suited for parallel programming.

The ability to make your own shader stages, called the Programmable Pipeline, was first introduced in DirectX 9 which allowed you to choose between the programmable and the fixed pipeline. In DirectX 10, the fixed pipeline was removed, and all shaders now had to be programmed by the developer. A shader is a small program, written in High Level Shading Language (HLSL), which runs on the GPU as opposed to the CPU. The pipeline consists of different shaders, executed in a specific order.

In DirectX 11 there are 6 shader stages, where most run in a fixed order:

Vertex shader → Hull shader → Domain shader → Geometry shader → Pixel shader

Then there's the compute shader, which is not a fixed part of the actual pipeline, but rather used in different places to compute data for the application. You can even use it in an application only to compute data, even if it has nothing to do with rendering.

To send data to the shader programs on the GPU, buffers are used. A buffer is a data structure based on, for example, a struct that is defined both in the normal application code and in the HLSL shader code. To let the GPU gain access to the buffer you bind it in different ways, by creating Buffer Description-variables and assigning different flags, depending on the purpose of the buffer. Usually the only usage is sending data to the GPU, and the buffer is therefore bound to the shader in a way that it can only be read, not written. With the compute shader, this would be pointless, since you want to be able to write back to the main memory, so the buffer has to be bound with read and write privileges.

TESTING AND RENDERING SPHERES WITH RAY TRACING

This chapter describes how we developed the application for rendering spheres through raytracing, and how we tested the performance of the algorithm.

To make sure that the tests are fair, we used the same raytracing algorithm for both the CPU and the GPU version of the application and the same vertex and pixel shaders.

Rendering

To be able to render the spheres to the screen easily, we decided to render a single texture in front of the camera, that the ray tracing algorithm writes directly to. The vertex shader then generates a quad, and the pixel shader samples from the texture to paint that quad to the screen.

On the CPU

For rendering with the CPU, the ray tracing algorithm stores all pixel color values it receives when running in a byte array, which is then mapped to a texture that is then sampled by the pixel shader for the actual rendering. The CPU version of the ray tracing algorithm is divided up into eight threads, which each process an equal amount of pixels, and therefore an equal amount of rays. Following is the basic order of events on the CPU algorithm:

> SPHERE ARRAY → CALCULATE ON CPU → PIXEL COLOR BYTE ARRAY → TEXTURE → VERTEX SHADER → PIXEL SHADER

In order to get the optimal number of threads for the algorithm, we did some testing to see when the algorithm performed at its best. The first thing we tried was to let one thread handle one row of pixels, which resulted in as many pixels as the height of the screen, which in this case were 768. This proved to be too many for the CPU to handle, and made it perform worse than without any threading. After this we reduced the number of threads to only four, since that's the number of cores in the CPU. This proved to be a lot more effective, and we started increasing the number of threads up to ten, and from this came to the conclusion that eight threads would give the best performance result, since there would always be a thread ready to run on every core.

On the GPU

For rendering with the GPU, we use the compute shader to compute the pixel colors, and can write directly to the texture. The GPU has a lot more threading capability, containing about 800 cores, compared to the CPU's four, although not all are usable for all kinds of calculations. This allowed us to have one thread for each pixel, meaning that each thread computes only one ray and one pixel color. Following is the basic order of events on the GPU algorithm:

> SPHERE ARRAY → BUFFER → CALCULATE ON GPU → TEXTURE → VERTEX SHADER → PIXEL SHADER

When using the compute shader, you dispatch a number of thread groups, which each has a number of threads, and each thread executes a method defined with High Level Shading Language (HLSL). Both the groups and the threads in the groups are defined in three dimensions, so there are X, Y, Z groups and each group has X, Y, Z threads.

At the beginning of development for the GPU, a window with the resolution of 1024*768 pixels was used, and we decided to try having one thread for each pixel. We started out with groups of (2, 2, 1) threads, which comes to four threads per group. To get the number of groups needed, we divided the screen width and screen height by two, which came to (width/2, height/2, 1) groups, which comes to a total of 196,608 groups.

To find the optimum amount of groups and threads, we started to increase the number of threads per group and reduce the number of groups, making sure that the total amount of threads were equal to the number of pixels.

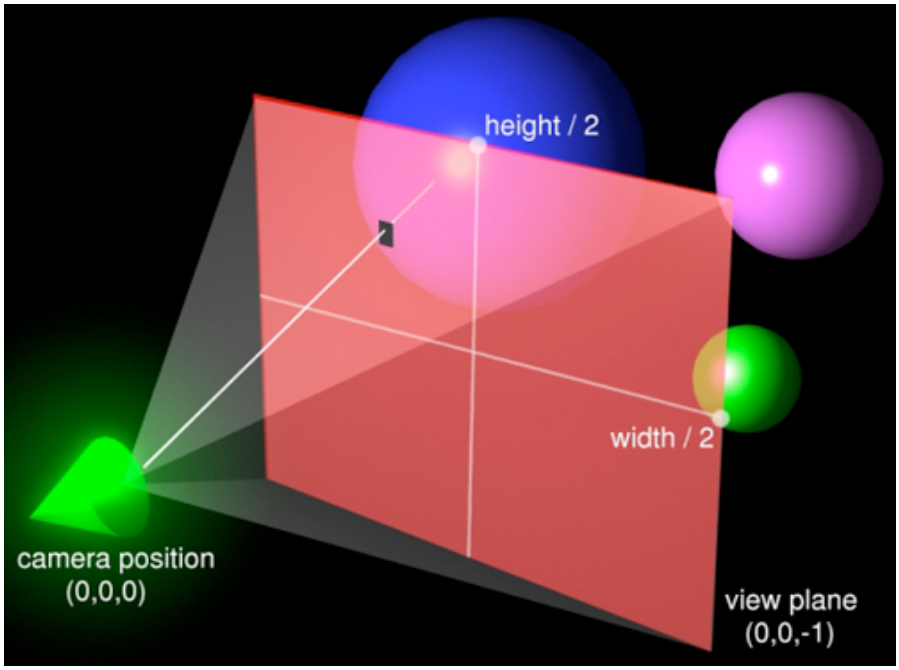
We stepped from (2, 2, 1) threads and (width/2, height/2, 1) groups to (4, 4, 1) threads and (width/4, height/4, 1) groups and so on, until reaching (32, 32, 1) and (width/32, height/32, 1) groups. At this point, the total number of threads in each group is 1024, which is the maximum number that the compute shader allows. Performance increased with each step, so we settled on the maximum amount of threads in each group.

The Camera

To generate the rays, we need a camera. The camera is generated by using a camera position, a view direction and a view plane. The plane is placed at a fixed offset from the position, in the direction defined. With the camera defined, we shoot the rays from the camera position along the view direction towards the plane, using the screen width and height to make sure that a ray passes through every screen pixel.

When a ray collides with something, we check if it's the closest collision so far for that ray, and if it is, save the distance and color. When we have iterated through all the spheres in the scene,

we have the nearest collision and the color we will need to write to the texture at the right pixel position.



Schmidt, R. Camera and View Plane: Illustration of how the camera works.

Results

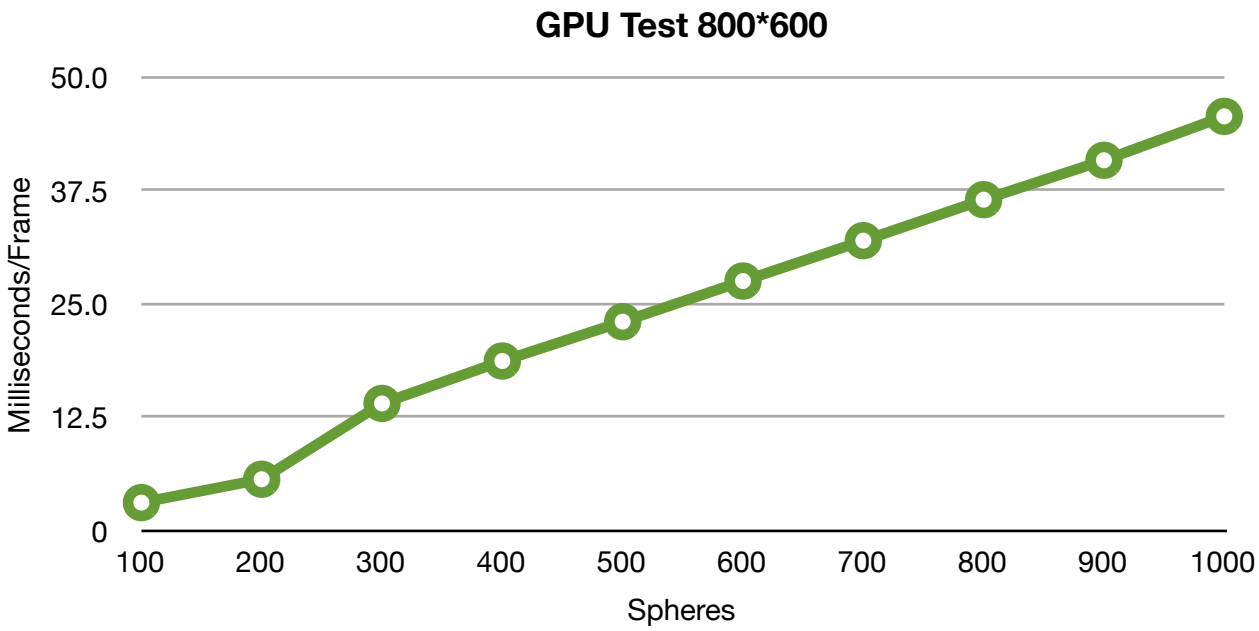
For the testing, we used a personal computer running Windows 7, with a minimal amount of other programs running in the background. The computer was rebooted before the tests.

Performance was tested by measuring the rendering speed of each frame in milliseconds, with a varying amount of spheres, from 100 to 1000, on three different resolutions, 800*600, 1280*800 and 1680*1050 running for ten seconds and taking the average frame rendering time. The test computer specifications was as follows:

Component	Description
CPU	Intel Core i5 750, 2.67 GHz
Memory	4 GB, Corsair Dominator, 1600 MHz
GPU	Radeon HD 5770 1GB GDDR5, 850 MHz
Operating System	Windows 7 Ultimate 64bit
DirectX	DirectX 11 (February 2010)

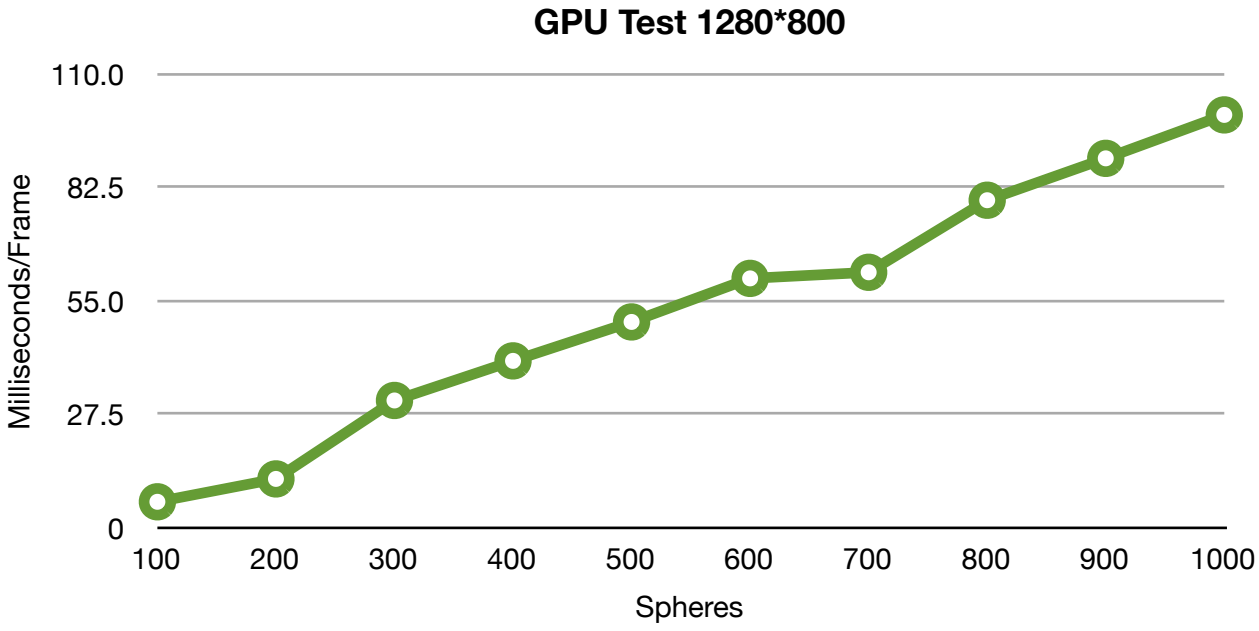
Test 1

Rendering with the GPU at a resolution of 800*600 pixels, resulting in 480,000 rays being tested against the specified number of spheres.



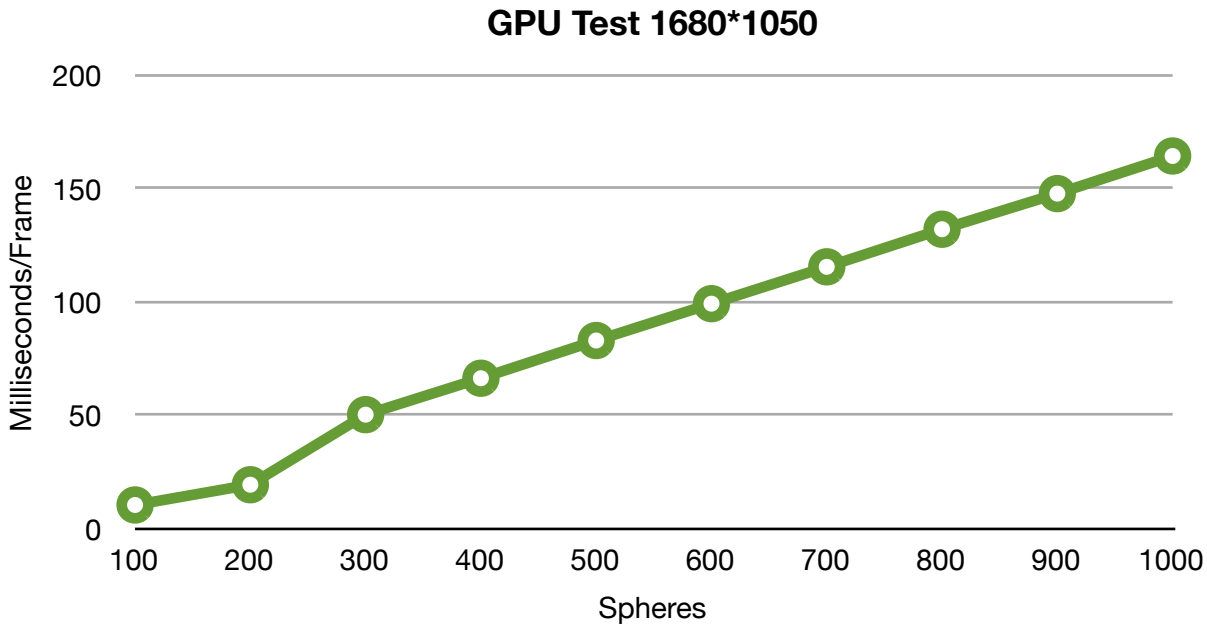
Test 2

Rendering with the GPU at a resolution of 1280*800 pixels, resulting in 1,024,000 rays being tested against the specified number of spheres.



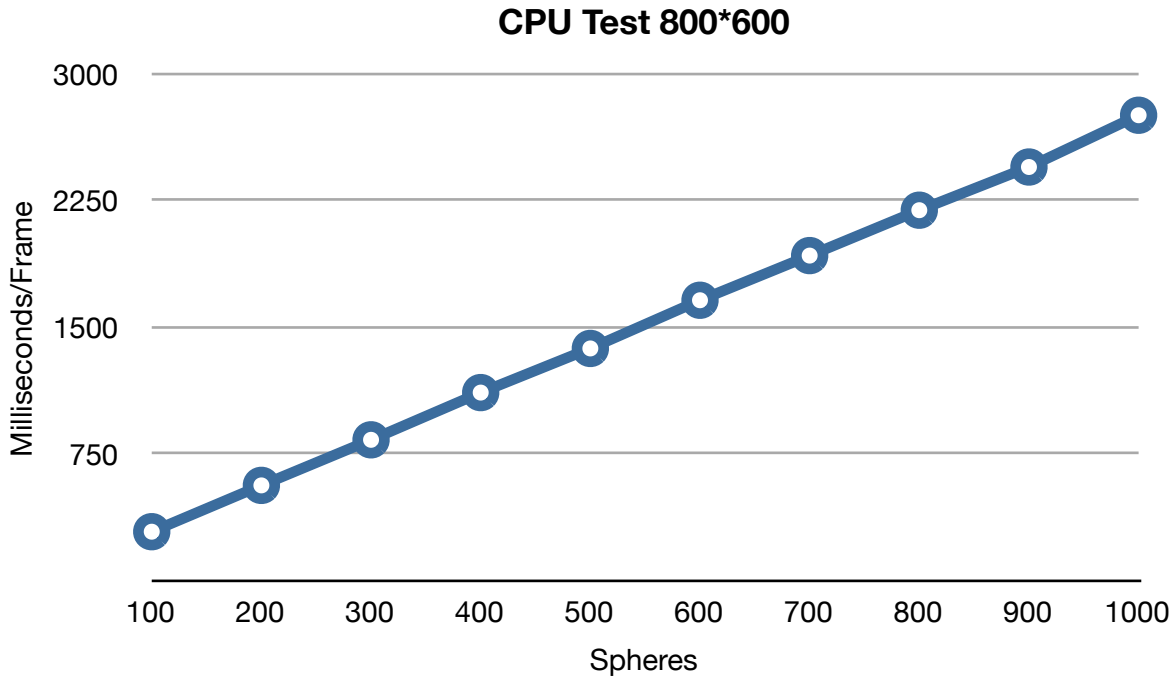
Test 3

Rendering with the GPU at a resolution of 1680*1050 pixels, resulting in 1,764,000 rays being tested against the specified number of spheres.



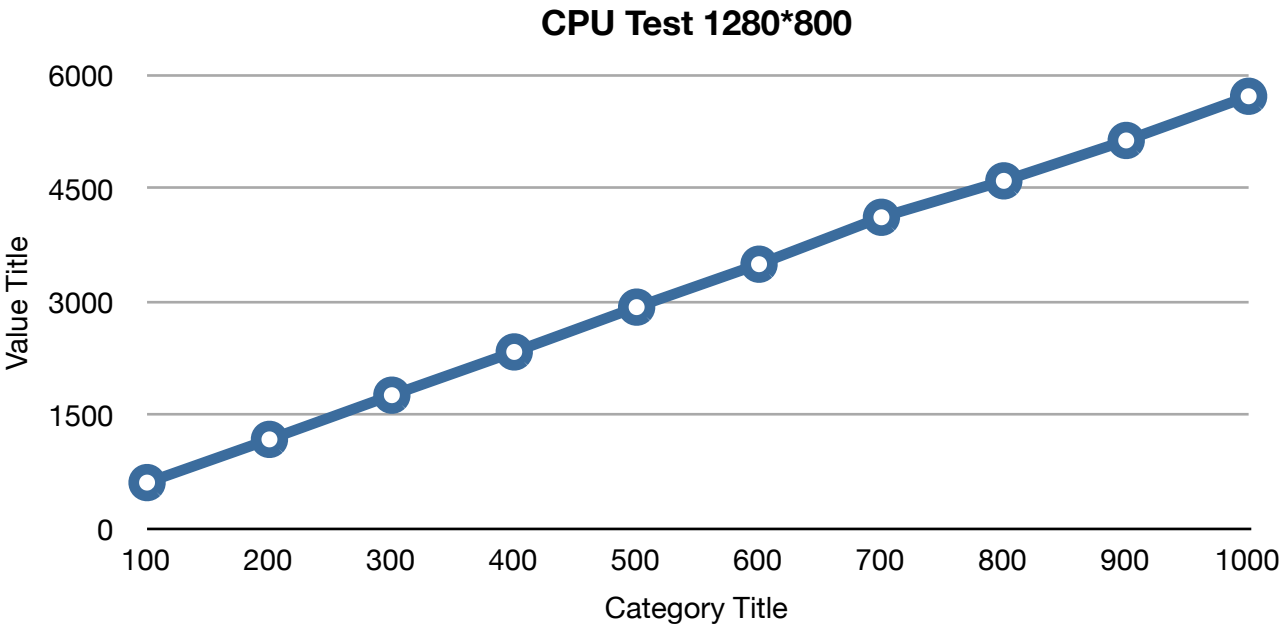
Test 4

Rendering with the CPU at a resolution of 800*600 pixels, resulting in 480,000 rays being tested against the specified number of spheres.



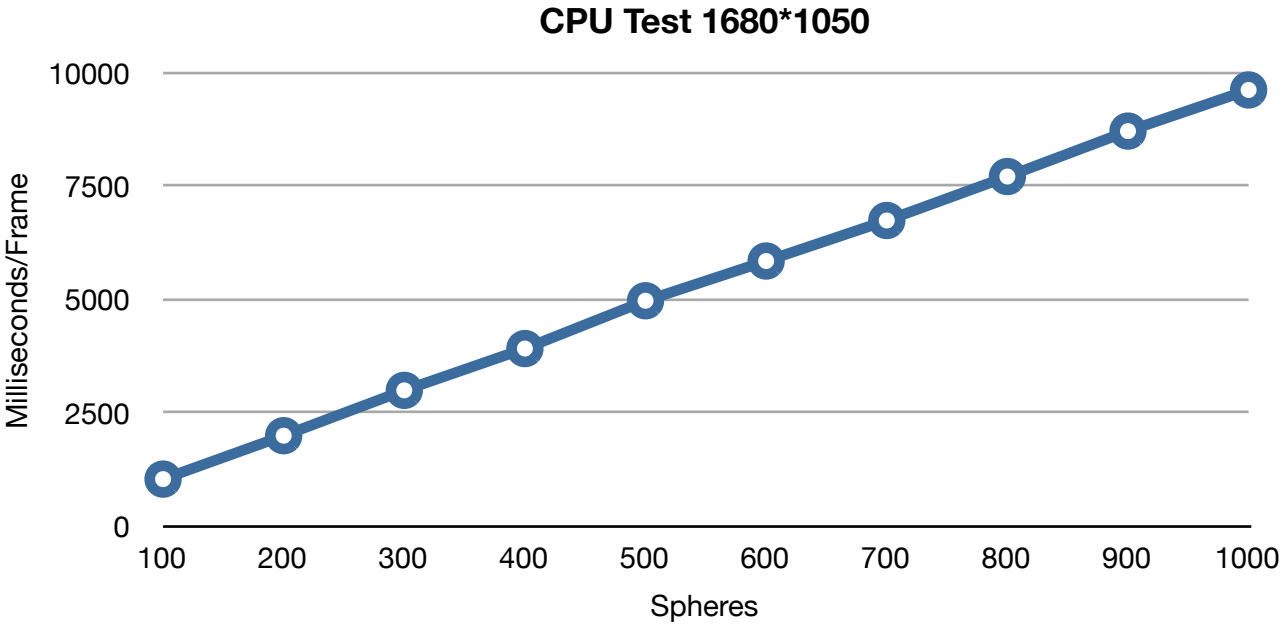
Test 5

Rendering with the CPU at a resolution of 1280*800 pixels, resulting in 1,024,000 rays being tested against the specified number of spheres.



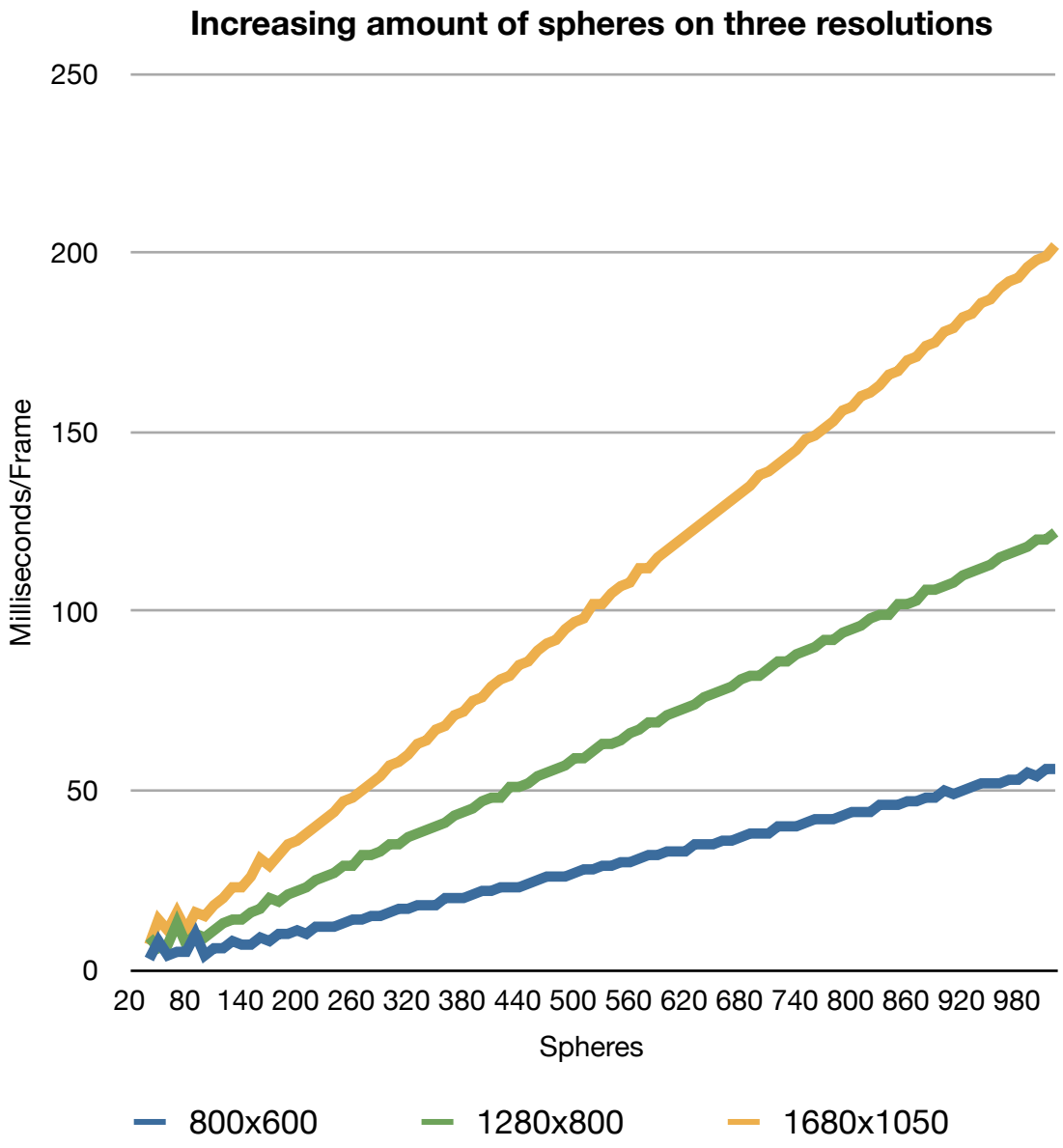
Test 6

Rendering with the CPU at a resolution of 1680*1050 pixels, resulting in 1,764,000 rays being tested against the specified number of spheres.

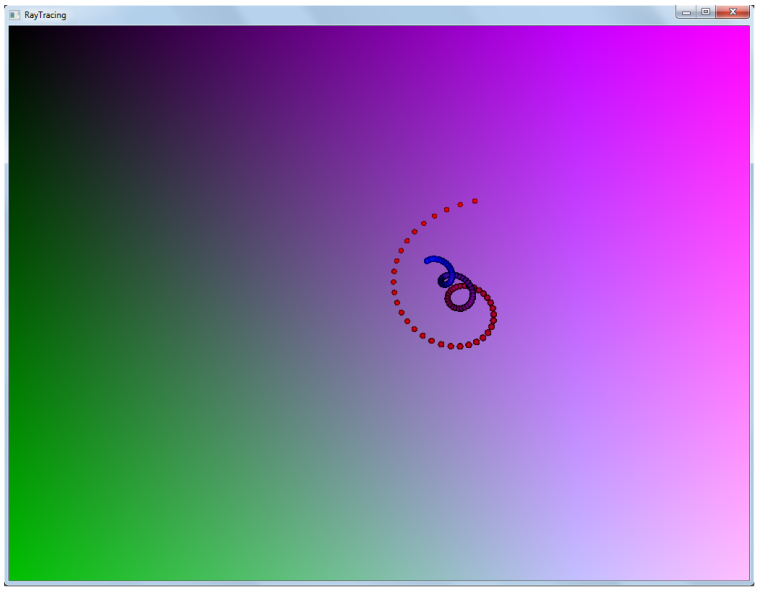


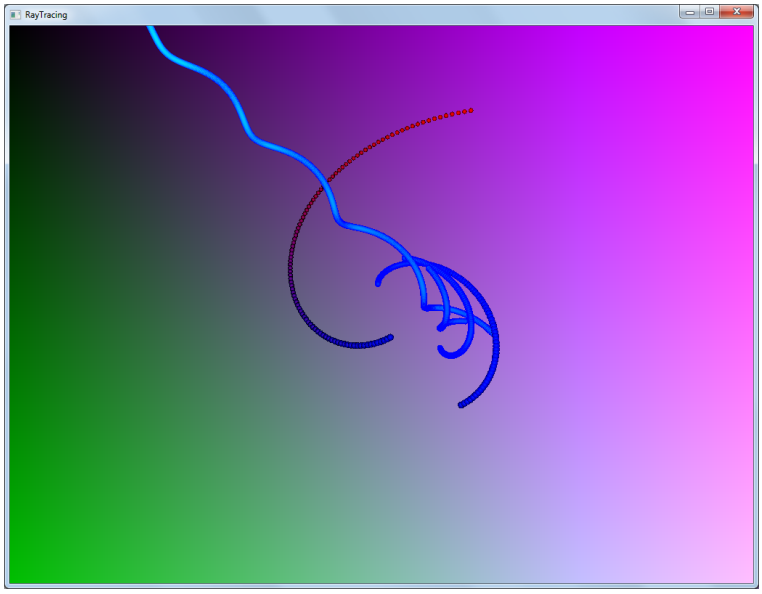
Test 7

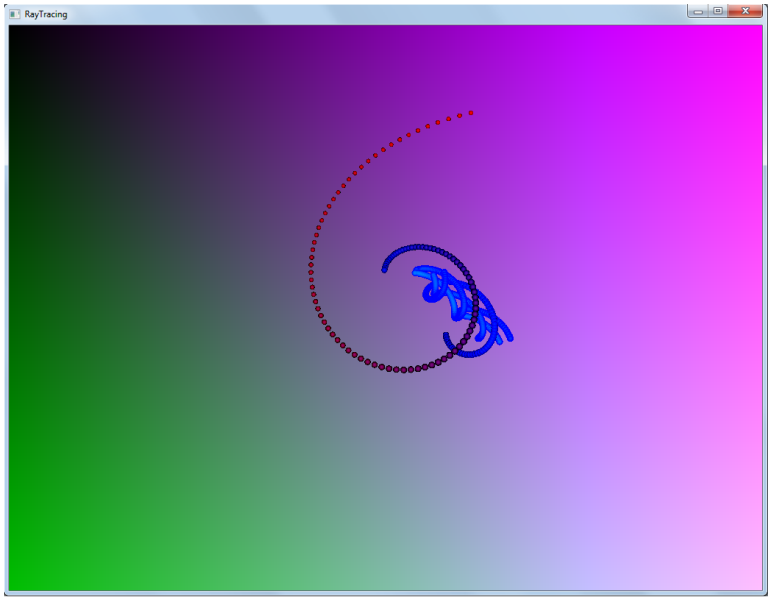
In this test, we started the application with ten spheres, and then increased the number of spheres in the scene by ten each frame, until there were a thousand spheres. This test only ran on the GPU. Note that in this chart the first frame is ignored, because it is the frame where everything is initialized and set up, and is therefore not a fair representation of the ray tracing algorithm's performance. Note that the small variations in the beginning seem to be normal, based on our testing.



Test Screenshots







DISCUSSION AND CONCLUSIONS

Discussion

This thesis focus on ray tracing using the compute shader versus using the CPU. The tests performed were however very simple, and can be optimized to further increase performance, and the algorithm is not complete, since shadow rays and other secondary rays are not implemented. The tests should therefore only be used as a starting point for future research of real-time ray tracing on the compute shader.

The test results showed that what the hypothesis claims, that the compute shader will always perform faster, was correct, with the compute shader greatly outperforming the CPU in every test. Theoretically if secondary rays and different kinds of scene objects were included, the compute shader would still perform better than the CPU, although not necessarily at a level suited for real time applications such as games. During the tests the GPU reached activity levels between 85 and 99%, with the higher screen resolutions having higher activity levels compared to the lower resolutions.This theoretically means that the screen resolution, and therefore the amount of threads, causes a heavier workload, keeping the GPU busy and might leave little computing power for other calculations, leading to longer rendering times. GPU activity was measured using the ATI Catalyst Control Center and it's performance tools.

The goal was to test the compute shader and see if it would always outperform the CPU at simple ray tracing. The main goal has been reached, but further investigations have to be done to get a complete answer to the question.

Further investigations could test to see if the same is true for complete and optimized versions of the ray tracing algorithm, and if the performance stays at the same level in more complicated scenes with, for example, more complicated primitives and how performance is affected when the other shaders have more complicated work to do.

Conclusion

As you can see in the test result graphs, all the results are linear depending on the amount of spheres in the scene. This makes it easy to calculate the max amount of spheres you can have in a scene while still meeting a certain rendering time. The CPU tests seem to have a more constant linear performance curve than the GPU ones. This might be because the compute shader is still new, and might have some infancy problems that makes performance slightly unreliable. In the tests the GPU tended to render at speeds between 58 and 60 times faster than the CPU, theoretically meaning that the GPU is 60 times better at rendering with ray tracing. Looking at the computer specifications, you can see that the combined computing power of the CPU is 4*2.67 and the GPU 800*0.85, showing that the GPU has about 64 times

the computing power of CPU. This shows that the GPU rendering about 60 times faster than the CPU fits very well with the hardware and further proves how much threading improves ray tracing performance, and shows how linear the performance gains are.

In Test 7 we use a variable called nrOfSpheres in the compute shader, which increases with 10 every frame, that the application uses to decide how many spheres to draw. In the other tests we use a constant number that's set before compiling and running the program. With this variable being used in a normal for-loop, also in the compute shader, even if it's just set once at the beginning of runtime, performance will decrease as compared to the GPU when using a constant value. This is most likely a limitation in the compute shader, and could be a good thing to put further research into, since it might hurt performance.

Ray tracing has always been very computationally expensive, compared to the main rendering technique, raster rendering, which only takes a pixel at face value, and often doesn't require intersection tests to get it's results.

Looking at the tests, one can easily see that a ray tracer, even if it only uses primary rays, requires a great number of calculations to render a single image. This, together with the very linear increase in time it takes to render a frame when increasing the amount of scene objects, it's easy to see that for a real time ray tracer to succeed, it needs to be highly optimized to reduce or speed up intersection tests.

A very promising point is that Intel in 2008 developed a version of Enemy Territory: Quake Wars that renders by ray tracing {Valich, 2008} that runs on four Intel Tigerton CPUs which have 187.6 GFLOPS each, which comes to a combined computational power of 750.4 GFLOPS {Barker, et al. 2008}. Our test computer's GPU has a total computing power of 1360 GFLOPS {Wasson, 2009}, which makes it safe to say that with a good algorithm, it's very likely that it already is possible to make real time ray tracers work on current generation consumer hardware.

BIBLIOGRAPHY

Oyonale.com, Scene with glasses, http://www.oyonale.com/modeles.php?lang=en&page=40, link worked on 17 of May 2010

Bikker, Jacco. Refraction through a Sphere. Devmaster.net, 2005. Retrieved from http://www.devmaster.net/articles/raytracing_series/part3.php on 19 of May 2010

Schmidt, Ryan. Camera and View Plane, UnknownRoad.com, retrieved from http://www.unknownroad.com/rtfm/graphics/rt_eyerays.html on 18 of May 2010

Bikker, Jacco. Raytracing: Theory and Implementation. Devmaster.net, 2005. Retrieved from http://www.devmaster.net/articles/raytracing_series/part1.php on 12 of April 2010.

Whitted, Turner. An Improved Illumination Model for Shaded Display. Holmdel, New Jersey: Bell Laboratories. Published in Communications of the ACM Volume 23, Number 6 in June 1980

Valich, Theo. Intel converts ET: Quake Wars to ray-tracing, TGDaily June 12, 2008. Retrieved from http://www.tgdaily.com/trendwatch-features/37925-intel-converts-et-quake-wars-to-ray-tracing on 19 of May 2010

Barker, K. Davis, K. Hoisie, A. Kerbyson, D. Lang, M. Pakin, S. Sancho, J.C. Experiences in scaling scientific applications on current-generation quad-core processors. Published on pages 1-8 of Parallel and Distributed Processing, 2008 IPDPS 2008 on 14-18 April 2008

Wasson, Scott. AMD's Radeon HD 5770 and 5750 graphics cards, The Tech Report, 2009. Retrieved from http://techreport.com/articles.x/17747/5 on May 19, 2010

Christensen, P.H. Fong, J. Laur, D.M. Batali, D. Ray Tracing for the Movie 'Cars'. Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006, pages 1-6. Salt Lake City, USA, September 2006

APPENDIX A

HLSL Source Code, Compute Shader

```
///------------------------------------------------------------
///                 ComputeShader.fx
///                 RayTracer Demo
///                 Direct3D 11 Shader Model 5.0
///
///                 Copyright(c) Jonathan Udd,
///                 Gustav Persson, 2010
///------------------------------------------------------------

///------------------------------------------------------------
///                 Raytracing structs
///------------------------------------------------------------
struct Ray
{
        float3 origin;
        float3 direction;
};

struct Sphere
{
        float3 center;
        float radius;
        float4 color;
};

struct IntersectData
{
        float dist;
        float4 color;
};

///------------------------------------------------------------
///                 ConstantBuffer
///------------------------------------------------------------
cbuffer CBPerFrame : register(b0)
```

```
{
	int nrOfSpheres;
	float3 cameraPos;
	float3 cameraDir;
	float3 cameraUp;
	float3 cameraLeft;
	float3 cameraUpperLeft;
	Sphere Spheres[1000];
};

///------------------------------------------------------------
///		ReadWrite Texture
///------------------------------------------------------------
RWTexture2D<float4> mOutBuff;

IntersectData rayVsSphere(Sphere sphere, Ray ray)
{
	IntersectData data;

	float3 EO = sphere.center - ray.origin;
	float v = dot(EO, ray.direction);
	float disc = (sphere.radius * sphere.radius) - (dot(EO, EO) - (v * v));
	if(disc < 0.0f)
	{
		data.dist = -1.0f;
		return data;
	}
	else
	{
		float d = sqrt(disc);
		float vd = v - d;
		data.dist = vd;
		float3 P = ray.origin + float3(ray.direction.x * vd, ray.direction.y * vd, ray.direction.z
* vd);
		float colorMod = (sphere.radius - d) / sphere.radius;
		data.color = float4(sphere.color.x - colorMod, sphere.color.y - colorMod,
sphere.color.z - colorMod, 1.0f);
	}
	return data;
}
```

```
#define MAX_DISTANCE 9999999

[numthreads(32, 32, 1)]
void CSMain( uint3 threadID : SV_DispatchThreadID )
{
	uint width = 0;
	uint height = 0;
	mOutBuff.GetDimensions(width, height);
	float4 finalColor = float4(threadID.xy / 1024.0f, threadID.x / 786.0f, 1.0f);



	float3 left = cameraLeft * threadID.x;
	float3 up = cameraUp * threadID.y;
	float3 offsetToPos = -(left + up);
	float3 pixelPos = cameraUpperLeft + offsetToPos;

	Ray ray;
	ray.origin = cameraPos;
	ray.direction = normalize(pixelPos - cameraPos);

	float closestDist = MAX_DISTANCE;
	for(int i = 0; i < nrOfSpheres; ++i)
	{

		IntersectData data = rayVsSphere(Spheres[i], ray);
		if(data.dist < closestDist && data.dist > 0.0f)
		{
			closestDist = data.dist;
			finalColor = data.color;
		}
	}
	mOutBuff[uint2(threadID.x, height - 1  - threadID.y)] = finalColor;
}
```