Volume Raycasting Performance Using DirectCompute

Written by:
Håkan Johansson
Blekinge Institute of Technology
June 2012

Abstract

Volume rendering is quite an old concept of representing images, dating back to the 1980's. It is very useful in the medical field for visualizing the results of a computer tomography (CT) and magnet resonance tomography (MRT) in 3D. Apart from these two major applications for volume rendering, there aren’t many other fields of usage accept from tech demos.

Volumetric data does not have any limitations to the shape of an object that ordinary meshes can have. A popular way of representing volume data is through an algorithm that is called volume raycasting. There is a big disadvantage with this algorithm, namely that it is computationally heavy for the hardware. However, there have been vast improvements of the graphic cards (GPUs) in recent years and with the first GPU implementation of volume raycasting in 2003, how does this algorithm perform on modern hardware? Can the performance of the algorithm be improved with the introduction of GPGPU (DirectCompute) in Directx 11? The performance results of the basic version and the DirectCompute version was compared in this thesis and revealed significant improvement in performance. Speedup was indeed possible when using DirectCompute to optimize volume raycasting.
Table of contents

1 Introduction ................................................................................................................................. 3
   1.1 Question and Hypothesis ............................................................................................... 3
   1.2 Methodology .................................................................................................................... 4

2 GPGPU – General Purpose Computing On GPUs ................................................................. 4

3 DirectCompute .......................................................................................................................... 6
   3.1 Motivations for using DirectCompute ........................................................................... 8

4 Volume Rendering .................................................................................................................... 9
   4.1 What it is used for today .............................................................................................. 10

5 Volume RayCasting .................................................................................................................. 10
   5.1 Theory ............................................................................................................................. 10
   5.2 Implementation .............................................................................................................. 12
   5.3 Compute shader implementation ............................................................................... 19

6 Results ....................................................................................................................................... 20

7 Discussion ................................................................................................................................ 23

8 Conclusion ................................................................................................................................. 24

10 References .............................................................................................................................. 25
1 Introduction

The game industry and the number of people playing video games grow larger every day. With that the performance and the capacity of the hardware which drives video games improves year after year. The demands on the graphical aspect of a game has never been higher and rendering techniques that were impossible for the hardware to handle in real-time some years ago might be possible to use with modern hardware.

An important part of making a visually pleasing game is the particle effects. A particle effect is used in games to represent things like smoke, fire, rain, explosions etc. Particles are typically illustrated using billboards which in some cases can make the effect look quite “stiff” because it is not a real 3D volume; it's only a billboard with a texture on it. There are a number of techniques to make a particle effect look better, some more demanding than others. But what if it’s possible to make a real 3D volume effect, for example smoke? This can actually be done with an algorithm that is called volume raycasting, also known as volume raymarching. Effects can be made to look very realistic, but the problem is that this algorithm is very computationally heavy for the hardware and is not very suitable for a real-time video game. With recent potential to use the kernels of a GPU, i.e. a graphics card, I will test if this algorithm can be improved to the point that it is suitable for a real-time video game. In this paper, the theory behind volume ray casting and GPGPU (General Purpose Computing On GPUs) is described. It will be presented how a basic GPU version of volume ray casting is implemented and how performance was improved with the help of DirectCompute.

1.1 Question and Hypothesis

As mentioned in the introduction, the visual aspect of games has improved with a rapid pace over the years. As Zink, et al. (2011) describes, the capability of the GPU, which has until later years been purely reserved for graphical calculations, has become so powerful that it’s becoming more and more considered for general purpose computing.

The question is what possible gain in performance, i.e. rendering time and frame rate, can be achieved for the volume raycasting algorithm with the help of general purpose
computing on the GPU. And also, will the gain in performance be enough to make volume raycasting usable in a real-time video game.

The hypothesis is that, sufficient gain in performance can be achieved with the volume raycasting algorithm to be able to use it in real-time video games.

1.2 Methodology

To be able to get a good performance measurement I implemented the algorithm in two different ways. Firstly an implementation of a basic multi pass version was done as described in section 5.2, which this paper will refer to as the original version.

The second stage was to optimize the multi pass version with a compute shader, described in section 5.3, which this paper will refer to as the optimized version.

When these two versions were finished, data sets (described in section 5.2) of various sizes were collected that are known to be used for measurements like this from the Volvis\textsuperscript{1} website.

Using these data sets, comparisons were made between the two different implementations to see how much faster the optimized version was.

2 GPGPU – General Purpose Computing On GPUs

GPUs have been extremely improved over the years and are in some ways more powerful than a CPU. Luna (2012) describes:

“GPUs have been optimized to process a large amount of memory from a single location or sequential locations (so-called “streaming operations”); this is in contrast to a CPU designed for random memory access. Moreover, because vertices and pixels are independently processed, GPUs have been architected to be massively parallel; for example, the NVIDIA “Fermi” architecture supports up to 16 streaming multiprocessors of 32 CUDA cores for a total of 512 CUDA cores” (p.429).

\textsuperscript{1} volvis.org
Graphical computations benefit significantly from this architecture since GPU consists of a large number of cores and therefore each pixel can be computed in parallel. Luna (2012) describes, some non-graphical applications can also benefit from the computational power of the GPU with its parallel architecture. Using the GPU for processing that is normally done on the CPU is called general purpose GPU programming, i.e. GPGPU. A CPU today normally has 2, 3 or 4 cores, which is significantly less than the GPU, as seen in image 1.

Not all algorithms are ideal for the GPU. The graphics card needs a data parallel algorithm to really take advantage of the parallel architecture. For example, shading pixels is well suited for this architecture. Luna (2012) describes:

“We need a large amount of data elements that will have similar operations performed on them so that the elements can be processed in parallel” (p.429).

In GPGPU programming, the programmer often needs to fetch the result of the computation from the GPU back to the CPU, which requires copying the results from video memory and sending it to the system memory (as seen in image 2), which is a bottleneck (but the computation can be a lot faster than on the CPU so it's often worth it). However for graphics, results of the GPGPU computation are often used as an input to the rendering pipeline.
Therefore, there is no need to send anything to system memory. A good example of this is blurring textures with the compute shader and then sending the result as an input to another shader, instead of having to do the blurring directly in a pixel shader.

3 DirectCompute

Zink, et al. (2011) describes:

“DirectCompute introduces a new processing paradigm that attempts to make the massively parallel computation power of the GPU available for tasks outside of the normal raster-based rendering domain” (p.288).

DirectCompute allows the user to perform GPGPU tasks directly in the Direct3D 11 processing environment. It shares much with the existing framework from the rendering portion of the API and therefore is quite easy to learn. DirectCompute is not a direct part of any of the pipeline stages and it can execute at any time during runtime. Luna (2012) writes:

“In GPU programming, the number of threads desired for execution is divided up
into a grid of thread groups. A thread group is executed in a single multiprocessor”
(p.432).

For example solving a problem on a GPU with 16 multiprocessors, the work would be
divided into at least 16 thread groups so that all multiprocessors have work to do.

A thread group has access to shared memory, which is very fast, that all threads
can read and write to, but a group cannot access other groups shared memory. There are
also functions for synchronizing the threads in a group; however, the different groups
cannot be synchronized.

Here is an example of a simple compute shader written by Luna (2012) that sums
up two textures, just for showing the basic syntax and how similar it is to other HLSL
examples.

cbuffer cbSettings
{
    //Compute shaders can access values in constant buffers
}
// Data sources and outputs
Texture2D InputA;
Texture2D InputB;
RWTexture2D<float4> OutPut;

//The number of threads in this thread group.
//Max number of threads in a thread group is 1024.
[numthreads(16,16,1)]
void CS( int3 dispatchThreadID : SV_DispatchThreadID ) //Thread ID
{
    //Sum the xyth texels and store the result in the xyth texel of OutPut
    OutPut[ dispatchThreadID.xy] = InputA[dispatchThreadID.xy]+ InputB[dispatchThreadID.xy];
}
technique11 AddTextures
{
    pass p0
    {
        SetVertexShader(NULL);
        SetPixelShader(NULL);
        SetComputeShader(CompileShader(cs_5_0, CS()));:
}
There are two types of resources that can be bound to a compute shader: buffers and textures.

The output resource in the example has a prefix to its type “RW”, which stands for Read-Write. That means that this is a resource that the compute shader can both write to and read from; moreover if compared to the other resources: InputA and InputB one can see that they are normal textures and therefore are read-only. Binding resources in a compute shader is the same syntax as you would write a normal shader except from the RW resources. They need to be bound as a new view type that is called an unordered access view (UAV), instead of the shader resource view, (SRV).

3.1 Motivations for using DirectCompute

As explained in the previous section DirectCompute is embedded in Direct3D 11, which makes it easy for people who already have experience with Direct3D programming to get a compute shader up and running. A lot of the initialization code is similar to other initialization code in Direct3D. Zink, et al. (2011) writes:

“The resource model, execution paradigms, and general debugging process all leverage existing knowledge” (p.288).

“Another great benefit of using DirectCompute is that the performance of a particular algorithm can easily scale with a user’s hardware. If a user has a high-end gaming PC with two or more high-end GPUs, then an algorithm can easily provide additional complexity to a game without the need to rewrite any code. The threading model of the compute shader inherently supports parallel processing of resources, so adding additional work when more computational power is available is trivial. This is even more true for GPGPU applications, in which the user typically processes as much data as possible, as fast as possible. If an algorithm is implemented in the compute shader, it can easily scale to the current system’s capabilities” (p.288).
Perhaps the single most important benefit with DirectCompute is that it is so close to the rest of the rendering pipeline so it can easily be used to supply input for rendering operations. That fact alone makes it a very powerful tool for Direct3D programmers.

4 Volume Rendering

There are many effects in video games and in movies that are volumetric by nature and that can be very difficult to represent using geometric primitives (i.e. triangles) such as fluids, fire, smoke, and fog etc., seen in image 3. To achieve good looking effects like these it is better to represent them using volumetric data.

There are a few ways to implement volume rendering but this paper will be focusing on volume raycasting, or volume raymarching. Hayward (2009) describes:

“Volume rendering is a technique for directly displaying a three dimensional scalar field without first fitting an intermediate representation to the data, such as triangles.”
“There are two traditional ways of rendering a volume, either through slice-based rendering or volume raycasting. There are many advantages with using volume raycasting over slice-based rendering, such as empty space skipping, projection independence, simple to implement and it can be done in a single pass.”
4.1 What it is used for today

Pawasauskas (1997) describes, the major application area for volume raycasting currently is in medical imaging, where volume data is created from X-ray Computer Tomography (CT) scanners and Positron Emission Tomography (PET) scanners. CT scanners create three dimensional stacks of parallel plane images, each of which consist of an array of X-ray absorption coefficients. Typically X-ray CT images will have a resolution of 512*512*12 bits and there will be up to 50 slides in a stack. The slides are 1-5 mm thick, and are spaced 1-5 mm apart.

5 Volume RayCasting

This section explains the theory behind volume raycasting and the two different implementations of this project.

5.1 Theory

The basic idea of volume raycasting is that a volume (a cube) is created and rays are traced from the camera into this cube, computing the volume rendering integral along these rays, as seen in image 4. All of these rays are handled independently, which makes it flexible for optimization compared to slice-based volume rendering. It allows for acceleration techniques such as early ray-termination, adaptive sampling and empty-space leaping. Volume raycasting has from the early implementations in the 1980’s been done on the CPU, the GPU based volume raycasting is a rather new concept with its first implementation in 2003.
For each pixel on the screen a ray is cast from the camera into the volume. The ray then traverses step after step with a certain step size. For each step the ray travels through the volume a scalar value is sampled from a 3D texture (data set) of a certain sizes. When data at the current position is collected, it is composed with the data from the previous step using this compositing algorithm:

```cpp
//result = float4(0,0,0,0);
//src = float4(the data collected from the 3D texture)
result = (1.0f - result.a)*src + result;
```

The data-set is basically an array of bytes stored either in 8-bit or 16-bit RAW format, which is loaded into a 3D texture.

The following is a pseudo description of the volume raycasting algorithm with these major components:

1. **Ray set-up:** A viewing ray needs to be set up according to given camera parameters and the respective pixel position. Here, the ray’s entry position in the volume and its direction is determined.

2. **Traversal loop:** After a ray has been set up, a traversal loop is initiated in order to make the ray travel along its direction.

   2.1 **Data access:** Here, data is collected from the data set from which we have created a 3D texture.
2.1 **Compositing:** The previously accumulated color and opacity are updated according to the front to back compositing equation with the algorithm described earlier.

2.2 **Advance ray position:** The ray is advanced from its current position with a certain step size.

2.3 **Ray termination:** The traversal loop ends when the ray leaves the data set volume.

Following is another pseudo code description of the algorithm:

Determine volume entry position  
Compute ray direction  
While ( ray position inside volume )  
Access data value at current position  
Compositing of color and opacity  
Advance position along ray  
End while

The volume raycasting algorithm creates one ray per pixel which means that it very easily can be implemented with a GPU approach.

5.2 **Implementation**

For this project, the focus was mainly on getting a basic version of the volume raycasting algorithm up and running to see how much speed up that could be gained with a compute shader version of the algorithm. This section will first describe how one can go about implementing the important parts to get a volume raycasting project working and then how it was optimized with a compute shader. This project is made from scratch with the use of DirectX 11. The shaders used in this project are compiled with this shader flag:
DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS | D3D10_SHADER_OPTIMIZATION_LEVEL3;

All data sets which were used have been gathered from the Volvis website. These data-sets are either in 8-bit format or 16-bit RAW format, but this project is only implemented with the 8-bit data-sets. First of all in order to be able to get anything rendered on the screen, data from the data-sets need to be read and handled. The data in these sets are merely scalar values (BYTES) stored as slices [x,y,z], where x = width, y = height and z = depth. What needs to be done first is to store these BYTES in an array with the following code:

```c++
void Volume::LoadRAWFile( string filename )
{
    ifstream ifs( filename.c_str(), ios_base::in | ios_base::binary );

    ifs.read( (char*)this->m_volumeData, this->m_width*this->m_height*this->m_depth);

    ifs.close();
}
```

`m_volumeData` is the BYTE array and `m_width`, `m_height` and `m_depth` is the size of the current data-set, for example 256x256x256. Now that the reading of the data is finished, this information needs to be sent in to the shader and to do that, a 3D texture is created using the information gathered in `m_volumeData`. This is done like this:

```c++
HRESULT Volume::Create3DTexture()
{
    HRESULT hr = S_OK;

    D3D11_TEXTURE3D_DESC desc;
    desc.Width = this->m_width;
    desc.Height = this->m_height;
    desc.Depth = this->m_depth;
    desc.MipLevels = 1;
    desc.Format = DXGI_FORMAT_R8_UNORM;
    desc.Usage = D3D11_USAGE_IMMUTABLE;
```
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = 0;
desc.MiscFlags = 0;

D3D11_SUBRESOURCE_DATA initData;
initData.pSysMem = this->m_volumeData;
initData.SysMemPitch = this->m_width;
initData.SysMemSlicePitch = this->m_width*this->m_height;

if( FAILED( hr = this->m_device->CreateTexture3D( &desc, &initData,
 &this->m_volumeTexture ) ) )
  cout << "Failed to create Texture3D" << endl;

if( FAILED( hr = this->m_device->CreateShaderResourceView(
  this->m_volumeTexture, NULL, &this->m_volumeTextureRV ) ) )
  cout << "Failed to create Texture3D ShaderResourceView" << endl;

return hr;}

Now that the information from the data-set is read and a 3D texture is created from it, it can be sent as input to a shader.

In order to render this volume texture a bounding volume (a cube) is fitted to the volume data. The cube is rendered and the volume texture is sampled with rays that are traversed through the cube. To do this, a ray that starts at the camera and intersects the cube needs to be found. Hayward (2009) explains:

“We could always calculate the intersection of the ray from the eye to the current pixel position with the cube by performing a ray-cube intersection in the shader. But a better and faster way to do this is to render the positions of the front and back facing triangles of the cube to textures. This easily gives us the starting and end positions of the ray, and in the shader we simply sample the textures to find the sampling ray.”

Before the actual raycasting can be done, two passes need to execute and render two textures: the front positions of the cube and the back positions of the cube. With these two textures, the direction of the current ray can easily be calculated. To be able to do this I created a RenderTexture class which has its own RenderTargetView and ShaderResourceView which creates a texture that is used as an input for the volume
raycasting shader. The shader code for rendering the positions is very simple:

```cpp
PS_INPUT POSITION_VS(VS_INPUT vIn)
{
    PS_INPUT vOut;

    vOut.posH = mul(float4(vIn.posL, 1.0f), WVP);
    vOut.texC = vIn.posL;

    return vOut;
}

float4 POSITION_PS( PS_INPUT pIn ) : SV_TARGET
{
    return float4( pIn.texC, 1.0f );
}
```

For the front face position pass, this blendstate is applied:

```cpp
BlendState bsNoBlend
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = FALSE;
    RenderTargetWriteMask[0] = 0x0F;
};
```

For the back face position pass, this blendstate is applied:

```cpp
BlendState bsDirection
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = TRUE;
    BlendOP = SUBTRACT;
    SrcBlend = ONE;
    DestBlend = ONE;
    RenderTargetWriteMask[0] = 0x0F;
};
```

Here are the two textures created from the two first passes, which will be used for calculating the directions of the rays, illustrated by images 5 and 6:
With these two textures, the direction of the current ray can easily be calculated and rendered to a third texture for debugging purposes, as illustrated in image 7:

Now everything is prepared so that it is now possible to do the actual raycasting. Hayward (2009) explains:

“We render the front faces of the cube. In the shader we sample the front and back position textures to find the direction (back - front) and starting position (front) of the ray that will sample the volume. The volume is then iteratively sampled by advancing the current sampling position along the ray at equidistant steps. And we use front-to-back compositing to accumulate the pixel color.”
The following code is the actual algorithm written in the pixel shader using all the information explained so far in this section:

```cpp
float4 RAYCASTSIMPLE_PS( PS_INPUT pln ) : SV_TARGET 
{
    float3 f = front.Sample( linearSam, pln.texC ).xyz;
    float3 b = back.Sample( linearSam, pln.texC ).xyz;

    float3 dir = normalize( b - f );
    float4 pos = float4( f, 0 );

    float4 dst = float4(0,0,0,0);
    float4 src = 0;

    float3 Step = dir*StepSize;

    for(int i = 0; i < Iterations; i++)
    {
        pos.w = 0;
        src = (float4)volume.Load(pos).r;

        src.a *= alpha;

        src.rgb *= src.a;
        dst = (1.0f - dst.a)*src + dst;
        //break from the loop when alpha gets high enough
        if(dst.a >= 0.95f)
            break;

        //advance the current position
        pos.xyz += Step;

        //break if the position is greater than <width, height, depth>
        if(pos.x > width || pos.y > height || pos.z > depth)
            break;
    }
    return dst;
}
```
Images 8-13 demonstrate some visual results of this implementation.

Image 8, CT scan of an aneurysm
Image 9, Bonsai tree
Image 10, Engine
Image 11, CT scan of a foot
Image 12, Lobster in a teapot
Image 13, CT scan of a skull
First it seemed obvious how to optimize this algorithm with a compute shader. As soon as the algorithm had been implemented, it clearly seemed that the raycasting iteration needed to be parallelized. The initial idea was that since one can specify a maximum of 64 threads in z in a compute shader, the iterations where the raycasting is done just needed to split up in 64 parts and let one thread in z do its own part. When all the threads were finished and all of the iterations had stored all values in an array, the stored values simply needed to be accumulated with the compositing algorithm. This however, turned out to be extremely tough for the graphics card to handle. It led to a huge drop in the frame rate (the opposite of the goal). Some tweaking was made here and there but with no big results. A similar thing was implemented but instead of using the accumulation method, the interlockedadd function in HLSL was used instead of the original compositing algorithm. It worked better than the first version but there still was a huge loss in frame rate. In these early versions, as many thread groups as there were pixels on the screen was dispatched and each thread group had 1x1x64 threads.

A new version which was basically the same thing as the original pixel shader version was implemented. The thing that differed was the thread group dispatch and the number of threads per group. Here's an example of how many threads that were used. If the screen had a resolution of 800x600, then the dispatch call would dispatch 50x50x1 thread groups and each thread group would have 16x12x1. This means that all of the pixels on the screen is covered by a thread. 50*16 = 800 and 50*12 = 600. And if there was a resolution of 1920x1080, the dispatch call would dispatch 60*54*1 thread groups, each with 32*20*1 threads so that each pixel is covered. 60*32 = 1920 and 54*20 = 1080.

Instead of letting the compute shader generate a texture, which would serve as an input to a pixel shader, the compute shader directly outputted to the backbuffer. So in other words, the computations that the compute shader did were immediately rendered on screen.

Here's the compute shader implementation of volume raycasting, where output is bound to the backbuffer:

```cpp
//Output directly to the backbuffer
RWTexture2D<float4> output;

[numthreads(16, 12, 1)]
```
void RAYCASTSIMPLE_CS( uint3 threadID : SV_DispatchThreadID, uint3 grpThreadID : SV_GroupThreadID )
{
    float3 f = front[threadID.xy].xyz;
    float3 b = back[threadID.xy].xyz;

    float3 dir = normalize( b-f );

    float3 stepSize = float3(1,1,1);
    float3 step = dir*stepSize;

    float4 pos = float4(f,0);
    float4 src = 0;
    float4 result = float4(0,0,0,0);

    for( int i = 0; i < 512; i++ )
    {
        src = (float4)volume.Load(pos).r;

        src.a *= 0.1f;
        src.rgb *= src.a;

        result = (1.0f - result.a)*src + result;

        pos.xyz += step;
    }
    output[threadID.xy] = result;
}

Compared to the original version in section 5.2 it clearly shows that these two implementations do not differ a lot from each other.

6 Results

The performance of the two different implementations was measured on two different things: frames per second (fps) and how long it took to calculate one frame in milliseconds (ms/frame). The measurements were done with two different resolutions: 800x600 and 1920x1080. The performance data was collected at the worst case scenario,
i.e. when the frame rate was at its lowest. The frame rate differed depending on where the camera was positioned. This applied for the original unoptimized version, since the compute shader version maintained its frame rate at all times. Here are the specifications of the computer that these measurements were made on:

Windows 7 professional 64-bit
Intel(R) Core(TM) i5 CPU 760 @ 2.80 GHz (4 CPUs)
Memory: 4 GB RAM
Video: ATI Radeon HD 5850

This project was implemented with DirectX 11 and the data sets on which the measuring were done are all 8-bit RAW format and have been collected from volvis.org.

Table 1 presents the measurement results of both the original version of the algorithm and the compute shader optimized version with a resolution of 800x600. It shows frames per second (FPS) and the time it took to calculate one frame in milliseconds (Ms/frame).

Table 1, Results of measurements of both versions with resolution of 800x600

<table>
<thead>
<tr>
<th>Data-set</th>
<th>Size 8-bit</th>
<th>FPS, Original</th>
<th>FPS, Compute Shader</th>
<th>Ms/frame, Original</th>
<th>Ms/frame, Compute Shader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neghip</td>
<td>64x64x64</td>
<td>~204</td>
<td>~570</td>
<td>~4.9</td>
<td>~1.8</td>
</tr>
<tr>
<td>Hydrogen Atom</td>
<td>128x128x128</td>
<td>~106</td>
<td>~326</td>
<td>~9.4</td>
<td>~3.0</td>
</tr>
<tr>
<td>Foot</td>
<td>256x256x256</td>
<td>~55</td>
<td>~129</td>
<td>~18.2</td>
<td>7.8</td>
</tr>
<tr>
<td>Head Aneurysm</td>
<td>512x512x512</td>
<td>~27</td>
<td>~66</td>
<td>~37.0</td>
<td>~15.2</td>
</tr>
</tbody>
</table>

As it show there was quite a big speed up with the compute shader (optimized) version compared to the original one. Table 2 summarizes the speedup which was gained. The column on the right shows how many times faster the compute shader version was compared to the original.
Table 2, Presentation of the speedup with resolution of 800x600

<table>
<thead>
<tr>
<th>Data-set</th>
<th>Size 8-bit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neghip</td>
<td>64x64x64</td>
<td>~2.8</td>
</tr>
<tr>
<td>Hydrogen Atom</td>
<td>128x128x128</td>
<td>~3.0</td>
</tr>
<tr>
<td>Foot</td>
<td>256x256x256</td>
<td>~2.4</td>
</tr>
<tr>
<td>Head Aneurysm</td>
<td>512x512x512</td>
<td>~2.4</td>
</tr>
</tbody>
</table>

Table 3 presents the measurement results of both the original version of the algorithm and the compute shader optimized version with a resolution of 1920x1080. It shows frames per second (FPS) and the time it took to calculate one frame in milliseconds (Ms/frame).

Table 3, Results of measurements of both versions with resolution of 1920x1080

<table>
<thead>
<tr>
<th>Data-set</th>
<th>Size 8-bit</th>
<th>FPS, Original</th>
<th>FPS, Compute Shader</th>
<th>Ms/frame, Original</th>
<th>Ms/frame, Compute Shader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neghip</td>
<td>64x64x64</td>
<td>~50</td>
<td>~145</td>
<td>~20.0</td>
<td>~6.9</td>
</tr>
<tr>
<td>Hydrogen Atom</td>
<td>128x128x128</td>
<td>~25</td>
<td>~78</td>
<td>~40.0</td>
<td>~12.8</td>
</tr>
<tr>
<td>Foot</td>
<td>256x256x256</td>
<td>~12</td>
<td>~30</td>
<td>~83.3</td>
<td>~33.3</td>
</tr>
<tr>
<td>Head Aneurysm</td>
<td>512x512x512</td>
<td>~3</td>
<td>~9</td>
<td>~333.3</td>
<td>~111.1</td>
</tr>
</tbody>
</table>

Table 4 summarizes the speedup which was gained. The column on the right shows how many times faster the compute shader version was compared to the original.
Table 4, Presentation of the speedup with resolution of 1920x1080

<table>
<thead>
<tr>
<th>Data-set</th>
<th>Size 8-bit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neghip</td>
<td>64x64x64</td>
<td>~2.9</td>
</tr>
<tr>
<td>Hydrogen Atom</td>
<td>128x128x128</td>
<td>~3.1</td>
</tr>
<tr>
<td>Foot</td>
<td>256x256x256</td>
<td>~2.5</td>
</tr>
<tr>
<td>Head Aneurysm</td>
<td>512x512x512</td>
<td>~3.0</td>
</tr>
</tbody>
</table>

7 Discussion

After analyzing the results one can clearly see that it is possible improve the performance of volume raycasting with the help of compute shaders quite extensively. With the hardware that was used for this test, the performance improved about three times in some cases. As shown in the results section, this test was not implemented on the cutting edge technology GPU and therefore it would be possible for even more speedup with better GPUs since this algorithm was GPU based.

After researching the subject of volume rendering one can clearly see that the volume raycasting was very computationally heavy for the hardware which is very noticeable in the results. Moreover, the frame rate of the original version of the algorithm was actually higher than one would first expect.

Judging from the information that was gathered from these tests one can see that the lower resolution data sets can actually perform very well but the larger data sets have a performance that is totally unacceptable for real-time video games. I would say, based on the results, that resolution of 64x64x64 and smaller are reasonable for video games; depending on the video-game type of course. One can be confident that the performance of the lower resolution data sets is high enough for certain video games.

Regarding the implementations there are some acceleration methods that can be implemented to get even better performance. They are sadly outside of the scope of this thesis, but further future research should implement these and evaluate the performance benefits they introduce. Three of these acceleration methods are described by Engel, et al. (2006):
**Early ray termination:** Early ray termination allows us to stop light rays as soon as we know that volume elements further away from the camera are hidden. Ray traversal can be stopped when the accumulated opacity is high enough.

**Adaptive sampling:** Another advantage of raycasting is that the step sizes for one ray can be chosen independently from other rays, e.g., empty regions can be completely skipped or quickly traversed with a large step size.

**Empty-space skipping:** This is useful when volume visualization contains large portions of completely transparent space. An additional data structure is used to find empty space regions. For example, an octree can be used to store the scalar data values within a node. In combination with the transfer function, these min/max values allow us to determine completely transparent nodes, which then can be skipped.

## 8 Conclusion

After analyzing the results it became clear that it is possible to improve the performance of volume raycasting with the compute shader. The algorithm did not have to be altered particularly much compared to the ordinary volume raycasting algorithm. The computation for volume raycasting is still heavy for the hardware but with the help of new technology for GPGPU programming, such as DirectCompute, a significant gain can be made in performance. This makes volume raycasting on smaller data sets suitable for certain games.

## 9 Acknowledgments

I would like to thank volvis.org for their volumetric data sets that I have been using throughout this project and my supervisor Petar Jercic for helping me write this paper. For additional support and encouragement I’d like to thank Stefan Petersson. I’d also like to thank Maria Albinsson for valuable input regarding language and layout. Finally, thanks to Blekinge Institute of Technology.
10 References


