Quantifying Visibility in Volumetric Rendering with Compute Shaders

by

Stroughair Michael, B.A.I.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

October 2017

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Stroughair Michael

September 7, 2017

Acknowledgments

Give many thanks and stuff...

Stroughair Michael

University of Dublin, Trinity College

October 2017

Quantifying Visibility in Volumetric Rendering with Compute Shaders

Stroughair Michael, M.Sc.
University of Dublin, Trinity College, 2017

Supervisor: Dingliana John

In Volumetric Rendering applications, understanding what aspects of the data are visible to the user is an important step in optimizing the expressiveness of the image, leading to a better final render.

The aim of this thesis is to design an algorithmic process by which visibility aspects of a Volumetrically Rendered object can be computed in real time. The two taken into consideration within this thesis are the saliency of the data set, and the relative visibility of the individual data points within the final rendered image.

Contents

List of Tables

List of Figures

Chapter 1

Overview

The contents of this thesis fall within the research area known as Direct Volume Rendering, or DVR. In the vast majority of cases, an object is rendered using a polygonal mesh to represent its surface geography, and textured using a static image. The lighting applied to this mesh can be described using a Bidirectional Reflectance Distribution function, or BRDF, a surface rendering algorithm that only solves for light from the surface points of an object. As such, when information from the internal structure of an object is required, this method is not suitable.

Direct Volume Rendering techniques allow for the internal structure of an object to be taken into account when rendering the surface image. In these cases, the information cannot be stored simply within a 2D image, but instead within a volumetric grid. This grid consists of image slices sampled from the original object at regular intervals, with each slice containing a regular pattern of elements. The elements of information contained within this grid are known as voxels, the 3D counterpart to pixels. At lower resolutions of data, volumetrically rendered objects appear to be made up of tiny cubes, each of which is a voxel.

Voxels are assigned what is known as an intensity or a material value. This range for this value can vary depending on the data set, but generally it is set between 0 and 255. Using what is known as a Transfer Function, these material values are mapped to an RGBA colour. In many cases, this transfer function doesn't assign each individual material a colour, but instead assigns a colours to ranges of materials.

This project deals with the question of visibility within this 3D dataset. With the data stored and rendered in this way, it is possible for important features to be hidden behind less important ones. While the alpha values of particular materials can be changed within the transfer function to better see or ignore it, this is a time consuming process, and automation is preferable. Many algorithms have been developed to deal with this problem with varying degrees of success.

This thesis aims to calculate information for a volumetric gridset on a per-voxel basis in real time, for the purpose of being used by other researchers to better automate and optimise their visibility algorithms. The two information types dealt with are voxel visibility and saliency.

The Visibilitity of a voxel is a measure of how influencial that voxel is to the final image. The visibility of materials within a volumetric grid is well established, through the creation of visibility histograms etc, but in many cases, researchers wish to know the visibility of individual voxels within the final rendered image. For this reason, visibility fields were introduced, which contain the visibility of each individual voxel within the grid. Saliency is a measure of how noticible a voxel is based on the colour assigned to it by the transfer function. To compute this, the idea of a saliency field was introduced. Unlike a visibility field, a saliency field is view independent, and so does not need to be recalculated each time the camera moves relative to the object.

Calculating these fields can be quite expensive, and while it is possible to do these calculations through the rendering pipeline, it is not made for this purpose. However, OpenGL introduced a new type of shader known as a Compute Shader. Compute Shaders are general purpose shaders, built for large calculations. This makes them perfect for the calculations required by this project. Unlike other solutions available such as CUDA and OpenCL, Compute Shaders are native to OpenGL and so do not require any external files to run. They also are coded through GLSL, making them ideal for new users with experience in shader design.

1.1 Structure

The structure of this thesis is as follows: Chapters 2 and 3 provide an overview of related work within the fields that this thesis is a part of. Chapter 2 talks about Volumetric Rendering, briefly talking about the history before moving into the main techniques used. Chapter 3 provides an explanation of Compute Shaders. Chapter 4 details the implementation of the project, and how each of the core components operate. Chapter 5 walks through the evaluation of the project, presenting the results that were obtained during testing. Chapter 6 is a summary of the work.

Chapter 2

Volumetric Rendering

This chapter discusses Volumetric Rendering.

2.1 Volume Data

Here, I want to make clear what exactly is being rendered in a volumetric rendering pipeline, and what makes it different to conventional rendering.

2.2 Direct Volume Rendering

Here, I want to give a quick overview of what is happening. Basically, I want to talk about how its mapping the intensity values contained within the dataset to colours and alpha values from the transfer function onto the image plane

2.2.1 Direct Volume Rendering Methods

Here, I'll go into how its all supposed to work at a low level, give a good amount of detail. I'll first talk about object order approaches, since I'll need to have readers well versed in the idea for when I describe my Compute Shader implementation. Then I'll chat about image order approaches, to help understand the raytracing aspect of the program.

Object-Order Approaches

Here's some words

Image-Order Approaches

Here's some more words

2.3 Visualisation

Chapter 3

Compute Shaders

In modern graphics programming, shaders are the process by which a programmer accesses the GPU to perform work. The most used shaders are the Vertex and Fragment shaders, which form part of the modern programmable graphics pipline. Vertex shaders perform work on individual vertices, while fragment shaders take the newly created fragments and perform operations on them. Other shaders such as the Geometry shader are also available to a programmer for their needs. Compute Shaders work outside of the pipeline, and work per job. The number of jobs is dependent on the size of the global work group required by the program, but they can be grouped into local work groups for better memory allocation. When a compute shader is dispatched, the workgroups are formed into a 3-dimensional array, with the positions in the local workgroup and in the global workgroup represented by a vector.

The strength of compute shaders lies in their flexibility. Their position as shaders allow them to take advantage of preexisting code written to create the rendering pipeline, making them a simple addition to most programs. Once set up properly, they can be used for a variety of tasks, from image processing to physics calculations. In the case where data written from a compute shader will immediately be used by the rendering pipeline, or by another compute shader, memory barriers must be placed into the code to prevent data hazards. Compute Shaders are able to take advantage of many of the strengths of other shaders, including the ability to share data between invocations through the *shared* keyword. This must be paired with memory barriers within the shader itself to ensure no data hazards.

There are of course, alternatives to Compute Shaders that can be used, such as OpenCL and CUDA, but Compute Shaders were chosen for this project for their simplicity. OpenCL requires a user to switch context to use it, but this is not the case when it comes to compute shaders. They are a core part of OpenGL from 4.3, meaning their use will only increase as more users upgrade to that level. They also do not require external libraries or drivers to be installed, making them beginner friendly. Since they are a type of shader, it is not necessary to worry about memory allocation in the same way as is required with CUDA.

Chapter 4

Implementation

This chapter explains the implementation of the project goals. A Volumetric Renderer was provided initially to allow for more time to be devoted to the project's implementation rather than for its creation. Many portions of this original code still remain, however much of it was refactored or replaced during the project, to allow for better readability and useability.

The chapter is divided into two main sections. The first section deals with the shaders created during the project period. The second section breaks down the code that runs on the CPU.

4.1 Shader Design

This section contains two subsections. The first looks at the work done by the Compute Shader in calculating the Visibility Field. The second looks at the work done in calculating the saliency field.

4.1.1 Volumetric Rendering Shader

This shader was provided as part of the Renderer supplied at the start of the project period. It implements a ray marching algorithm for calculating the colour for each fragment from the data volume provided.

The code works by first finding the normalised direction vector from the camera to the current fragment. The absorption for that fragment is set to 0.0, and the code enters the main loop. Within this loop, the colour for that position within the data volume is calculated, multiplied by its opacity value, and added to the total colour (set to $(0.0, 0.0, 0.0, 0.0)$ initially). The opacity of the colour is added to the absorption for the fragment. The next sampling position is calculated by moving along the direction vector.

At this point, the algorithm determines whether it needs to continue. If the new position is outside the bounds of the data volume, or if the absorption of the fragment is equal or greater than 1.0, then it can return the current colour value. Otherwise, it calculates the colour for that new position, multiplies it again by the opacity value, and adds it to the total colour.

A later addition to this code was the ability to draw a border along the edge of the volume, for aesthetic purposes. If the initial position of the fragment is within a tolerance of the edge of the volume, the final colour outputted is black. In each data volume used, there is no relevant information stored this close to the edge of the volume, so it has no effect on viewing the final image. It does however serve to better understand where the camera is with relation to the primary axes.

4.1.2 Visibility Field Shader

When developing this shader, the problem of coordinate space was a constant concern. The data volume and camera are always within world space, which centers the object on (0,0,0), and the camera relative to this. However, the data volume is only accessible via texture space, where (0,0,0) is a corner of the volume. This required a number of conversion functions to be written to accomodate this.

The calculations are performed per voxel, taking full advantage of the design of the compute shader by assigning each voxel to a shader invocation. The algorithm then inspiration from per-voxel volumetric rendering by raymarching from the voxel to the camera, calculating the opacity of the voxel relative to the voxels in front of it. This is achieved by subtracting the opacity of any voxels encountered during the raymarch from the current opacity of the voxel. If this opacity drops below 0, the voxel is not visible at all to the user, and the algorithm ends early. If the algorithm detects that it is outside of the limits of the volume, it also ends early.

The second problem that was encountered was one of sampling frequency. One issue with raymarching algorithms for volumetric rendering is how often should the ray sample the data to get one sample per voxel. Along any of the primary axes this is simply half the length of a voxel, but when projecting along any other line, this distance is different. The solution however to this was quite simple. Previously, the data volume and the output volume were passed to the shader using the $Image3D$ texture format. This allows data to be written to a variable and outputted back to the CPU, an important aspect to my solution. This limits the access of this data to integer values, meaning no interpolation. To negate the need for a variable sampling frequency, all that was required was to change the input of the data volume to a $Sampler3D$ format, and interpolate the opacity value required.

4.1.3 Saliency Field Shader

In most edge detection algorithms, a filter is applied to a greyscale version of the data. This would not be suitable in this case however, due to the importance of colour in depicting saliency. To mitigate this, the intensity of the colour for a voxel is determined by calculating the length of the 3D colour vector, and multiplying it by the opacity value. This is then used with the filter to determine the value of the gradient in an area.

The laplacian filter used here can be quite susceptible to noise, so a variable named $lowerLimit$ was included, whose value is user determined. In the case that the total value for a voxel is less than the value of $lowerLimit$, the voxel's value is set to 0.

4.2 CPU Code Design

This section deals with the code written to run on the CPU. The first section details how the Compute Shader is initialised and run. The second section explains the layout of the final rendered image.

4.2.1 Compute Shader Setup

Compute Shader setup is the same as the setup for Vertex and Fragment Shaders, but it differs in how it is called by the program. Rather than calling $DrawArrays()$, $DispatchCompute()$ is called instead, which requires the size of the local work group to be passed to it. Since the data being computed by the GPU during this time will be required later in the rendering pipeline, it is important to also call $glMemoryBarrier()$ to allow the Compute Shader to finish before continuing so as to ensure that no data hazards occur later on within the code.

4.2.2 Display Design

The code was built with the ability to orbit around each object independently. As such, each object is rendered separately to its own framebuffer object, each with its own camera with movement mapped to different keys. To fit both onto the screen, the horizontal screen size is halved, and each framebuffer is then rendered to a quad. The leftmost quad contains the original data volume, while the rightmost can either contain the visibility field volume or the saliency field volume, depending on the choice of the user.

Chapter 5

Evaluation

This chapter details the tests performed on the Visibility field computation code. The first test shows the difference in computation time between volumes of differing dimensions. The second test compared the computation times for the same volume but with the alpha values within the transfer function changed. The final test altered the size of the local work group within the compute shader to determine the impact on performance.

For each of these tests, the speed was computed as an average over 360 frames, where the volume was rotated 1 degree each time. The measurement was taken using the system clock, and measured only the time it took to setup, call the Compute Shader and then wait for it to return.

5.1 Data Used

The tests detailed below use five different data volumes. These are listed below.

The same transfer function was used for each test, with the exception of the second test. In this case, new transfer functions were created by reducing the opacities for the original transfer function by a fraction. The transfer function with the lowest opacity values had been assigned values $\frac{1}{4}$ of those from the original, the next with $\frac{1}{2}$, and the final with $\frac{3}{4}$.

Data Set	Dimensions (x * y * z)
Knee	379 * 229 * 305
Bonsai	256 * 256 * 256
Engine	256 * 256 * 110
Tooth	140 * 120 * 161
Nucleon	41 * 41 * 41

Table 5.1: *List of data sets used and their size*

5.2 Change in Computation Time against Volume Size

The aim of this test was to determine whether the computation time of the visibility field changed with a change in the size of the data set, and if so by how much. As shown in the table above, there was a good variety of sizes to test with, and the test resulted in the graph below. As would be expected, the graph suggests a linear relationship between average computation time and the size of the data set. The y-intercept is approximately 15ms, the time it took for the compute shaders to be set up and return the final values.

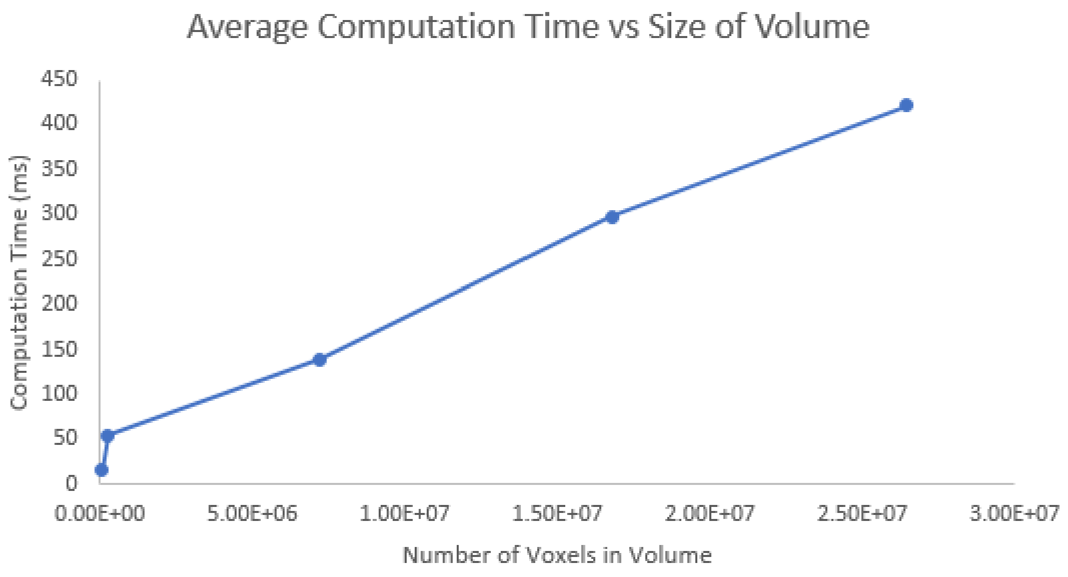

Figure 5.1: *A graph of the average computation times against the size of the data set being worked with*

5.3 Change in Computation Time against Opacity

Within the shader code, there is an early exit in the case where the code determines that the voxel is no longer visible behind all the other voxels in front of it. The aim of this test was to determine whether the computation time of the visibility field would change if the opacities of the voxels was changed, and perhaps cause the early exit to be used less often. This was achieved by measuring the speed of the computation against the same transfer function, but each copy of the transfer function having a successively lower opacity value for the most common intensity value over all the data sets. The results are shown below. The data suggests that there could be a small increase with lower opacities for certain intensity values, but the results are inconclusive, given how small the change is.

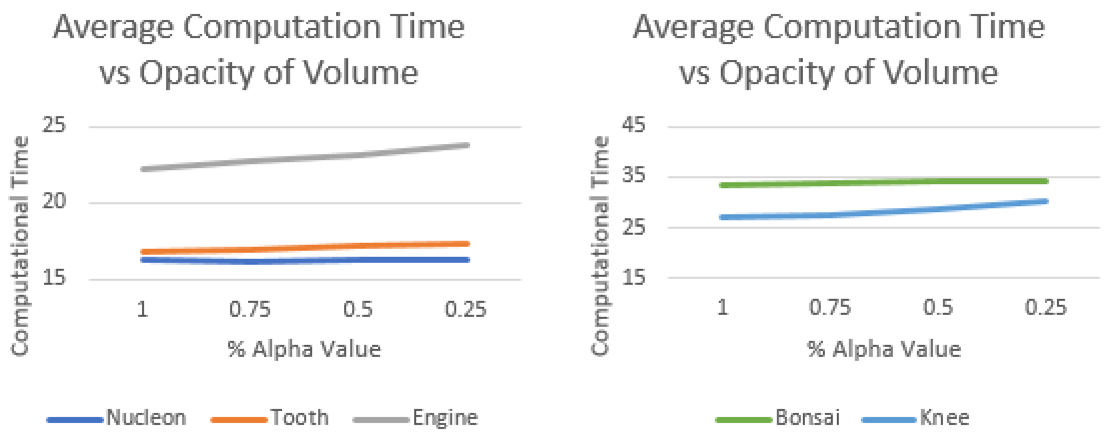

Figure 5.2: *A graph of the average computation times for each version of the transfer function over a total of 360 frames.*

5.4 Change in Computation Time against Local Work Group Size

The impact of the Local Work Group size on the Computation time was an unknown factor within this project. This test was designed to determine whether varying this size had any effect. It was evaluated by measuring the speed of the computation of the visibility field for different sizes of local group. The results are shown in figure 5.1 below. The data points to an exponential decrease in the computation time with an increase in workgroup size in powers of 2.

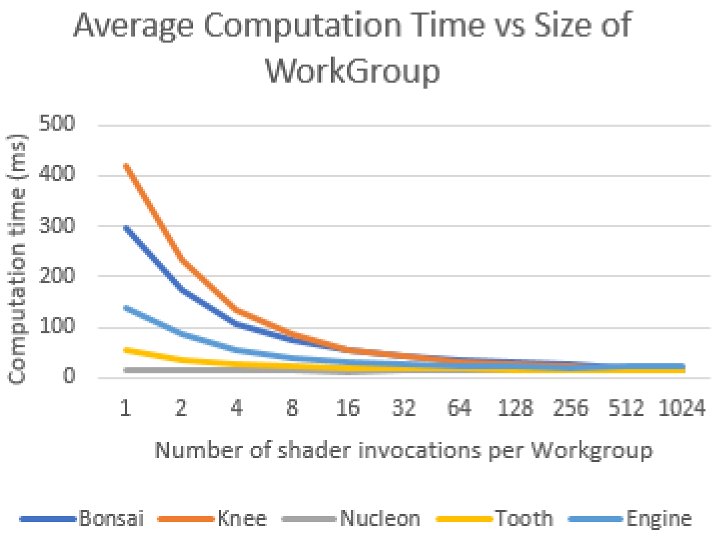

Figure 5.3: *A graph of the average computation times for each work group size over a total of 360 frames.*

One interesting observation that was made was on the errors that cropped up within the volume during evaluation during this test. While varying the size of each dimension disproportionally had no visible difference from varying each dimension equally, it was noticed that sometimes if a single dimension was large enough, errors would occur in the creation of the visibility field. In the case of the Engine data set, the volume would no longer output a visibility field if the value for the X dimension for the work group exceeded 256, while it would only output a partial visibility field for certain values of Z. This effect was also observed on the results from the saliency field compute shader.

The causes of thes errors was discovered when the sizes of each volume were determined. The tests were performed with volumes where each dimension was a power of 2. However, in table 5.1, very few of the volumes have power of 2 dimensions. In the case of the Engine, its length is 255 voxels, meaning that when it was no longer fully rendering, it meant that the compute shader had gotten too large.

The partial rendering error can also be explained with the size of the volumes. Since the Z dimension is the smallest, and the volume is aligned on that axis, the error was most noticeable there, but still present on the other axes. When assigning data points to a compute shader, the program divide the volume's dimensions by the size of the work group to get the number of data points per work group. However, since it rounds down, if there is any remainder from the division, those points will not be assigned to a workgroup. With a small workgroup or a large remainder, this effect is more noticeable, but will be present in any data sets whose dimensions won't have many numbers that divide into them evenly. Figure 5.3 was created using work group sizes that minimised this error as much as possible, but it will still be present in any volumes whose dimensions are not a power of 2.

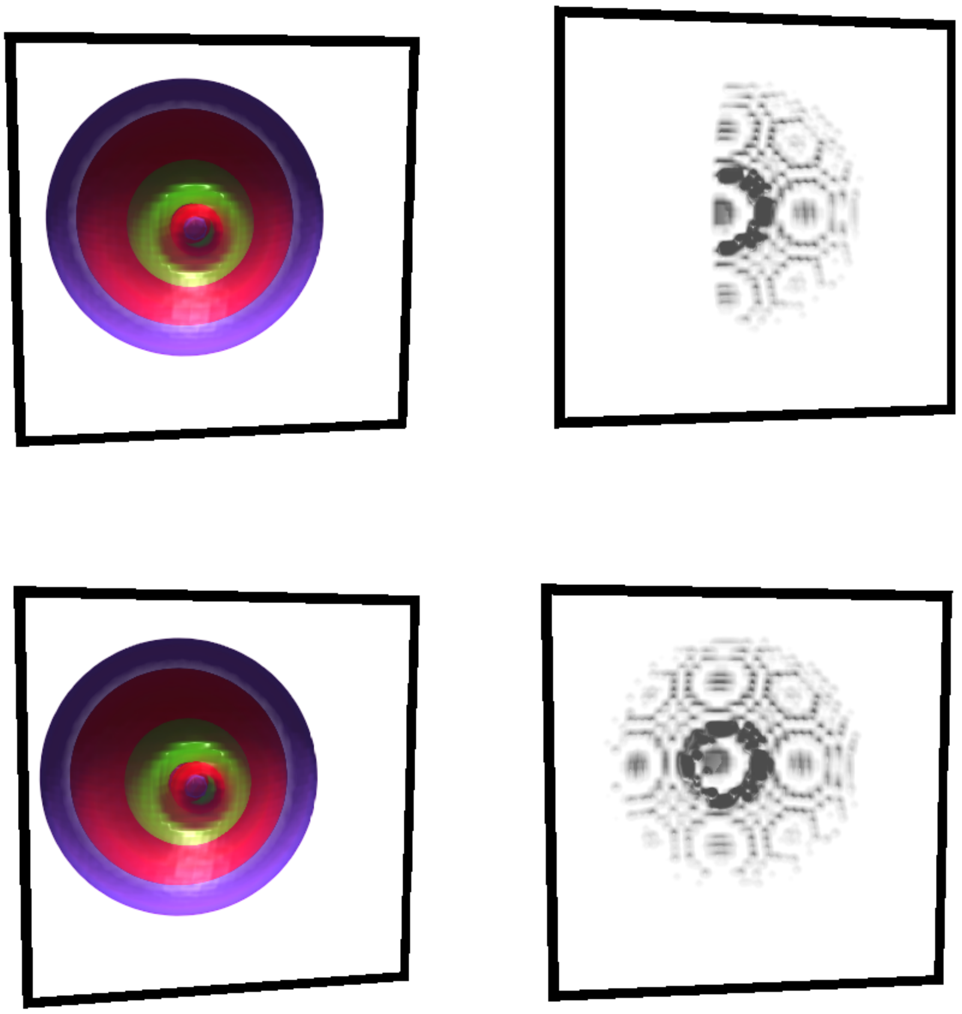

Figure 5.4: *Images showing the result of making the workgroup's x dimension equal to 20, thus cutting off most of the data, compared to making it equal to 41, which is the same as the volume's own dimension*

Chapter 6

Conclusions

And a fancy conclusion...

Appendix A

Abbreviations

Short Term	**Expanded Term**
DNS	Domain Name System
DHCP	Dynamic Host Configuration Protocol
...	...

Bibliography

[1] Laurie, Lundy-Ekman. (1998) Neuroscience: Fundamentals for Rehabilitation. W.B. Saunders Company, USA.

[2] Purves, D., Augustine, G., Fitzpatrick, D., Hall, W., LaMantia, A., McNamara, J.,Williamos, S. (2004) Neuroscience:Third Edition Sinauer Associates, Inc., USA.

[3] Longstaff, A. (2005) Neuroscience Second Edition. Taylor and Francis Group, USA.