NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK

MASTEROPPGAVE

Kandidatens navn: Kristian Eide

Fag: Datateknikk

Oppgavens tittel (norsk):

Oppgavens tittel (engelsk): Use of GPU Functionality in Volume Rendering

Oppgavens tekst:

Modern graphics cards include a GPU which is capable of executing small vertex and fragment shader program directly on the card itself. This unit's primary purpose is to enable real-time shading, however, the instruction set available has proved sufficiently general for performing other types of computations as well.

This thesis explores ways of taking advantage of the GPU in the field of volume rendering; specifically the ray casting technique and the optimizations it enables. In addition, it also investigates how transparent polygonal geometry embedded in a volume can be rendered correctly.

Oppgaven gitt:	03. september 2004
Besvarelsen leveres innen:	28. januar 2005
Besvarelsen levert:	28. januar 2005
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veileder:	Torbjørn Hallgren og Morten Eriksen

Trondheim, 28. januar 2005

Torbjørn Hallgren
Faglærer

Preface

This document is the Master thesis of Kristian Eide, prepared in the Fall of 2004, as the final part of his Master of Science degree from the Norwegian University of Science and Technology. He would like to extend his thanks to his research mentors, Torbjørn Hallgren of NTNU and Morten Eriksen of Systems in Motion, as well as Øystein Handegard and Peder Blekken for their input on volume rendering and OpenGL.

Summary

Volume rendering describes the processes of creating a 2D projection of a 3D discretely sampled data set. This field has a number of applications, most notably within medical imaging, where the output of CT and MRI scanners is a volume data set, as well as geology where seismic surveys are visualized as an aid when searching for oil & gas.

Rendering a volume is a computationally intensive task due to the large amount of data that needs to be processed, and it is only recently, with the advent of commodity 3D accelerator cards, that interactive rendering of volumes has become possible.

The latest versions of 3D graphics cards include a *Graphics Processing Unit*, or GPU, which is capable of executing small code fragments at very high speed. These small programs, while not as flexible as traditional programming, still represent a significant improvement in what is possible to achieve with the added computational ability provided by the graphics card.

This thesis explores how volume rendering can be enhanced by the use of a GPU. In particular, it shows an improvement to the GPU-based ray-casting approach presented in [1] and also a method for integrating the "depth peeling" technique [6] with a volume renderer for correctly rendering transparent geometry embedded in the volume.

In addition, an introduction to volume rendering and GPU programming is given, and a rendering of a volume with the Phong illumination model is shown.

Table of Contents

Chapter 1

Introduction

Volume rendering is the process of transforming a set of discrete sample points in three dimensions to an image in two dimensions which can be viewed on a computer display. Since the size of a volume data set increases with the cube of the length of its sides even relatively small volumes contain a significant number of samples, more commonly called *voxels* for volume elements, and for this reason are computationally expensive to render.

While volume data has typically been rendered in software running on a general-purpose CPU at speeds well below that required for interactive visualization, 3D accelerator cards support *texturing*, projecting an image onto the surface of geometry, which is fast enough to allow it to be used for interactive volume rendering. This is accomplished by drawing what is called "proxy geometry", usually planes, onto which the graphics accelerator draws texture slices from the volume. By drawing a large enough number of slices, a very realistic rendering of the volume can be realized. The first generations of 3D cards only supported 2D textures, which have some drawbacks when used for volume rendering as we shall see in chapter 3, while newer cards also support 3D textures which allow for a more natural way to organize the volume data.

The latest generations of 3D accelerator cards include a *Graphics Processing Unit* (GPU*),* essentially a co-processor specialized for performing graphics tasks. This specialization gives it less flexibility compared to the CPU, but also allows it to execute instructions in parallel at a far higher rate than what CPUs are capable of. It also allows them to increase in speed much more rapidly than general-purpose processors, which makes them an especially attractive platform for developing computation-intensive programs.

In this thesis I explore how the GPU can be utilized to improve various aspects of volume rendering. An approach to performing ray casting, the preferred way of rendering volumes in software, by taking advantage of the GPU has been proposed previously, and I improve on this approach by integrating near neighbor skipping, also a technique well-known in software-based ray casting. I also show how "depth peeling", a technique for correctly rendering scenes with transparent geometry independently of the order in which geometric primitives arrive, can be integrated into a volume renderer.

The rest of this thesis is organized as follows. Chapter 2 presents the original thesis proposal, as well as the final version. In chapters 3 and 4 an introduction to volume rendering and GPU programming, respectively, is given. A paper describing the near neighbor technique can be found in chapter 5, while chapter 6 contains a paper on integrating correct rendering of transparent geometry embedded in the volume. Results are presented in chapter 7, discussion in chapter 8, suggestions for future work in chapter 8, conclusions in chapter 9 and finally in chapter 10 ideas for future work is given.

Chapter 2

Project description

2.1 Thesis mandate

Thesis title: Use of GPU Functionality in Volume Rendering

Mandate: The purpose of this project is to investigate how the GPU of modern 3D graphics hardware can be used to enhance volume rendering. Prototypes of proposed methods should be implemented in order to evaluate their effectiveness.

2.2 Problem text

2.2.1 Original thesis proposal

Modern graphics cards include a GPU which is capable of executing small vertex and fragment shader program directly on the card itself. This unit's primary purpose is to enable real-time shading, however, the instruction set available has proved sufficiently general for performing other types of computations as well. The thesis will make a closer investigation into which possibilities exist for making use of the GPU in the field of volume rendering, with the goal of increasing performance, visual quality, or both.

2.2.2 Final thesis text

Modern graphics cards include a GPU which is capable of executing small vertex and fragment shader program directly on the card itself. This unit's primary purpose is to enable real-time shading, however, the instruction set available has proved sufficiently general for performing other types of computations as well.

This thesis explores ways of taking advantage of the GPU in the field of volume rendering; specifically the ray casting technique and the optimizations it enables. In addition, it also investigates how transparent geometry embedded in the volume can be correctly rendered.

Chapter 3

Volume rendering methods

3.1 Volume rendering explained

The process of volume rendering consists of visualizing a 2D projection of a 3D discretely sampled data set. Each data element is known as a volume element, or *voxel*, and is represented as a single value obtained by sampling the immediate area around the voxel location. The voxel value is usually not displayed directly, but used as input to a *transfer function* which maps each possible voxel value to a color, as in Figure 1.

Figure 1: Example of a volume rendered with a transfer function. Although this volume data set, of a CT scan of a human brain, is only 8-bit grayscale itself, a transfer function, which maps each of the 256 possible voxel values to a color, makes the rendering much more useful

Traditional 3D computer graphics uses primitives, such as points, lines and polygons, to build up a scene. These primitives can be combined in arbitrary size, shapes, colors and orientations, and, when textures are applied to surfaces, strikingly realistic images can be created. To create the image, a camera and viewport is defined and placed in the scene, and the objects in the scene are then projected onto the viewport. The process of projecting the objects to form a 2D image is known as *rendering*.

Volume rendering is similar in that the volume is projected onto a 2D viewport. However, unlike traditional rendering where objects are empty and consist only of a surface, volume data sets require rendering of the inside of objects as well. Indeed, the most useful feature of volume rendering is the ability to see inside of objects.

The rendering of a volume is a complex process and is composed of a number of individual steps. The next subsections of Chapter 3.1 explain

the steps involved in roughly the same order as they are applied.

3.1.1 Classification

Classification is a pre-processing step which assigns an *opacity* to each voxel. This is a value between 0 and 1 which describes how much of the incoming light is absorbed by the voxel, or how easy it is to see through that voxel. This classification step allows the user to emphasize structures in the volume by assigning voxels belonging to the structure a high opacity, and thus making it visible. Structure of little interest are made transparent.

3.1.2 Clipping planes

Another way of looking inside the volume is the use of *clipping planes*. A clipping plane, as the name implies, removes the section of the volume located in one of the half-spaces into which the scene is divided by the plane. By moving and rotating the clipping plane, arbitrary slices of the volume can be displayed. Other clip shapes than a plane are also possible, as we shall see in Chapter 6.

3.1.3 Compositing

Since there are several voxels projected onto the same pixel on the viewport, some of which might be transparent, these voxels need to be combined to form the final pixel color. The combining step is called *compositing* and is in general a complex, nonlinear operation; it is not easy to predict the final value of a pixel after combining the color values obtained from a set of voxel values. Even a slight change in the transfer function, which takes a single voxel value as input and gives a color value as output, can result in a completely different final color. There are several compositing operators available, each of which describes a way of combining a set of color values into a single such value, and the compositing of the voxels can be done either in front-to-back or back-to-front order, both of which have advantages and drawbacks.

The most used compositing model in use today is based on the following ray casting integral:

$$I(a,b)=\int_b^a g(s)e^{-\int t(x)dx}ds$$

Where *I(a,b)* is the intensity of a single pixel; *ds* is the ray direction,

with the ray running from *a* to *b*; *g(s)* is the source term which essentially describes the light model in use; and *t(x)* is the extinction coefficient which defines the rate of which light is occluded per unit of length due to scattering or extinction of light. This is the transparency of each voxel. *g(s)* and *t(x)* are the transfer functions which maps a voxel value to its intensity and opacity, respectively.

Since a computer cannot evaluate a continuous integral, an approximation has to be used; the Riemann sum is the simplest and is also widely used.

$$\int_d^0 h(x)\,dx \sum_{i=0}^{n} h(x_i)\,\Delta x$$

The Riemann sum gives a way to approximate the integral given only a discrete set of samples from the original function. The integral is divided into *n* equal intervals of length $\Delta(x)$, and the function's value is assumed to have a constant value *h(x)* over the interval. The discrete front-to-back version of the integral can now be written as:

$$I(a,b)=\sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} T_j$$

I_i is the intensity at position *i* along the ray; T_j is the transparency, a number between 0 and 1 which represents how much light pass through a point. If we instead of transparency use opacity the integral becomes:

$$I(a,b)=\sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} (1-\alpha_j)$$

This equation tells us that *I*, the total intensity accumulated so far along a ray, is the intensity I_j multiplied with all the transparencies, $(1 - \alpha_j)$, preceding it.

Now, in order to compute *I(a,b),* the ray casting equation must be evaluated recursively:

$$\sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} T_j = I_0 + I_1(1-\alpha_0) + I_2(1-\alpha_0)(1-\alpha_1) + \ldots + I_n(1-\alpha_0)\ldots(1-\alpha_{n-1})$$
$$= I_0 ov\, I_1 ov\, I_2 ov \ldots ov\, I_n$$

The *over* operator was first introduced by Porter and Duff in a SIGGRAPH paper on digital imaging in 1984 [13]; compositing thus reduces to applying the over operator on all samples along a ray. This recurrence can also be written:

$$I_{out} = I_{in} + T_{in} I_j$$
$$T_{out} = T_{in} T_i$$

I_{out} and T_{out} are the total accumulated intensity and transparency just after the ray hit the current sample point; I_{in} and T_{in} just before. I_i and T_i represent the current sample point. This can be implemented as follows:

```
Trans = 1.0;
Inten = I[0];
for (i = 1; i <= n; i++) {
     Trans = Trans * T[i-1];
     Inten = Inten + Trans * I[i];
}
```

When doing front-to-back compositing it is necessary to keep track of the accumulated transparency separately, which incurs some overhead. This also makes it possible to terminate, or stop processing, the current ray when the transparency falls below a given threshold and further samples will not contribute more toward the final result. The corresponding code for back-to-front compositing is:

```
Inten = I[0];
for (i = 1; i <= n; i++) {
  Inten = Inten + T[i] * I[i];
}
```

Since the accumulated transparency is not available, it is no longer possible to perform early ray termination.

3.1.4 Partial ray compositing

One important property of both front-to-back and back-to-front compositing is that partial rays can be composited. This means that a ray can be divided into smaller rays which can then be processed independently and the individual ray segments later combined to form the final pixel color.

Since ray tracing is a computationally intensive task, this is a potentially very useful property if hardware, such as a GPU, is available which can process multiple ray segments in parallel. After each ray segment is processed, the results are composited with the over operator.

3.1.5 Other compositing operations

Although front-to-back and back-to-front are the most important compositing operations there are several others. Two in particular deserve mention.

Maximum intensity projection is conceptually simple; it finds the maximum intensity value along the ray and this value becomes the final pixel value. This is useful in MRA (Magnetic Resonance Angiography) to visualize blood vessels which have very bright intensity values and thus will show up clearly. Doctors can use this to determine if a blood vessel is partially blocked and then make a decision on whether surgery is necessary or not.

X-ray projection generates images similar to conventional X-ray images. Instead of using the *over* operator to composite sample values along each ray the intensities are added. For this reason this compositing mode is also called SUM.

3.2 Challenges in volume rendering

Volume rendering is in many ways more challenging than traditional polygonal rendering. The most important reason for this is the large size of typical volume data sets, which can reach hundreds of megabytes, or even tens of gigabytes for geological surveys. The requirements for quality of rendering and the size of data sets only increase as more powerful hardware becomes available, and clever solutions have to be employed to give satisfactory results.

Before the advent of commodity hardware 3D accelerator cards volume rendering was typically done by ray casting implemented in software.

As a single rendering could take several minutes, or even hours, this technique was not suitable for interactive rendering. Specialized volume rendering hardware existed, but it was both very expensive and not very flexible, thus preventing wide-spread use.

Even early consumer 3D accelerators supported fast 2D texture mapping, which can be used for interactive volume rendering by rendering on axis-aligned proxy geometry. This method is described in more detail below. More recent 3D accelerators support 3D texturing in hardware, which permits loading volume data sets as single textures. This enabled the employment of more advanced techniques, such as non axis-aligned proxy geometry and GPU accelerated ray casting.

Another significant challenge is the correctness of the visualization. Since volume rendering is often used to inspect details in a data set it is important that even small details are visible and that no artifacts are introduced which are not actually present in the data.

3.3 Applications of volume rendering

Perhaps the most well-known users of volume rendering are the medical imaging industry and oil companies doing geological surveying in their search for new oil & gas reservoirs. There are many other usages as well, however, and its use increases as sampling technology improves and visualization hardware becomes more affordable and easier to use. The following presents an overview of some of the uses of volume visualization.

Medical imaging was one of the first applications of volume rendering and is still perhaps its most important use. CT, and later MRI, scanners produce 3D imaging data, and doctors need to be able to rotate and zoom, as well as color the data set, to distinguish different types of tissue and examine details. In addition to these features volume rendering also allows for making parts of the volume transparent to better focus on the relevant details.

A more recent development is surgical planning, where the surgeon can examine and virtually operate on a volume scan of the patient before the actual operation, to better prepare and also discover potential problems ahead of time. The surgeon can also have the visualization available during the operation, and even view data scanned in real time. With the addition of haptics the surgeon can operate on a patient from a remote

location with a robot mimicking his or her movements.

Modeling of different physical phenomena, such as ocean turbulence, solar magnetic storms, the ozone layer, hurricanes and typhoons, using volume rendering, can provide added insight into how these develop. Computational fluid dynamics is often used to model the flow of air over car bodies or aircraft fuselages to minimize resistance, and when using volume rendering to view the data in 3D areas of high vorticity (the rotational component of flow) can quickly be identified and the user can also get an overall feel for the flow in the system.

Nondestructive testing can be used when there is a need to examine the inside of an object without disassembling it. Since the very act of disassembly can disturb the object enough to invalidate the examination, or there is a wish to preserve the object intact, this is often the only way of gathering the data of interest. Examples of application range from looking for stress fractures or other flaws in failed integrated circuit dies or looking at the inside of engine blocks to examining the contents of dinosaur eggs.

Oil exploration is an application of volume rendering with the potential to save the oil companies a large amount of money from getting a higher "hit rate" on drilling. Today the majority of drilled wells are dry, i.e. there is not enough oil to justify production, but it has been estimated that hit rates as high as 80% [8] could be achieved be using volume rendering to examine the subsurface structures of potential sites before drilling.

3.4 Volume rendering methods

For the implementation of a volume renderer, several methods are in use, mostly depending on the available hardware. If a 3D accelerator card is available, interactive rendering is possible using 2D texture mapping, or, if the hardware is recent enough to support it, 3D texture mapping, which, while slightly more computationally intensive, requires less texture memory (thus larger volumes can be rendered) and gives better visual quality. Finally the ray casting method is introduced, however, its implementation on the GPU is only described in Chapter 5.

3.4.1 2D texture-based volume rendering

Early 3D accelerator cards only supported 2D textures in hardware, and a compromise had to be made to make use of this feature for volume

rendering. 2D textures are first made from slices of the volume data set, perpendicular to each of the three principal axes; the three resulting stacks of textures are painted onto "proxy geometry", which consists of stacks of planes parallel to the principal axes. Which of the three stacks is used depends on the camera position; the stack which gives the lowest angle between the vector from the center of the volume to the camera and the axis of the stack is used.

A disadvantage to this method is that three times the amount of texture memory is required to store the three stacks as compared to using a single 3D texture.

3.4.2 3D texture-based volume rendering

Modern graphics hardware supports 3D textures directly in hardware and arbitrary points inside the texture can be sampled. Trilinear interpolation is also supported in hardware, which makes this a very attractive options for volume rendering.

Instead of using axis-aligned planes as proxy geometry, the planes are oriented perpendicular to the viewing direction. Since the 3D texture supports arbitrary sampling, the plane can have any orientation and the hardware will correctly paint the appropriate slice of the volume onto the plane. When using this approach it is necessary to clip the planes against the volume bounding box, however. See Figure 2.

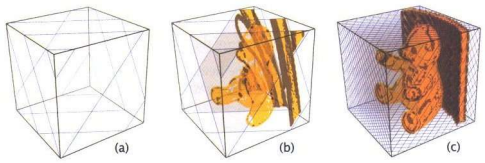

Figure 2: 3D texture based volume rendering. Planes oriented perpendicular to the viewing vector are drawn, clipped by the volume bounding box, and slices of the volume are drawn onto these planes; the image of the volume is built up layer-by-layer

3.4.3 Ray casting

Ray casting is the most natural approach to volume rendering, and, when implemented in software, this approach is most often used. A ray is cast from the camera location to each visible pixel on the surface of the volume bounding box and values are sampled at regular intervals along the segment of each ray inside the volume. See Figure 3.

Ray casting is similar to ray tracing, a common, but very computationally intensive technique used for rendering very realistic images. The high cost comes from the fact that the closest object with which each ray intersects must be determined, and each ray can also spawn two new rays – one reflected and one refracted. Neither of these properties apply to ray casting, but it is still computationally expensive as a large number of samples have to be taken along each ray to get acceptable quality.

Chapter 4

GPU Programming

4.1 Evolution of graphics hardware

Early graphics cards were dumb frame buffers whose only task was to display the contents of the buffer on the computer screen. The host computer had to do all the work of determining the color of each pixel before handing them to the graphics card. As the cost and size of transistors has been reduced, it has been a trend in the industry to implement in hardware functionality previously handled in software, as specialized hardware always requires less transistors and is usually faster than general hardware. For computer graphics this has meant a transition of an increasing number of stages in the graphics pipeline from the host CPU to the graphics card, see Figure 3.

The first 3D accelerator cards had a fixed-function pipeline, meaning that the card was like a black box in which vertices and textures were input in one end and pixels came out in the other. Various options could be set to change a number of constants used in the pipeline, but the steps performed were essentially fixed.

This changed with the introduction of the *Graphics Processing Unit*, or *GPU*, which allows program to be inserted at two points in the pipeline: the *vertex transformation* and the *rasterization*. The programs are known as *vertex shaders* and *fragment shaders,* respectively. These programs can perform mathematical calculations and logical operations and are free to transform their input in any way, or even discard it, before writing the result as output. Additional data can be provided by the application program as variables. The use of GPUs provides for much greater flexibility as compared to the fixed pipeline it replaced, and demanding algorithms can be implemented in software while still being hardware accelerated and able to run interactively.

At the time GPUs were introduced they had a significant drawback, however, in that the implementation of their programs had to be written in an archaic assembly-like language. This made it very time-consuming to experiment with different algorithms, especially taking into account vendor-specific instructions and different performance characteristics between graphics cards, and it limited usefulness somewhat. A solution to this problem came with the introduction of *Cg*, a high level programming language for GPUs.

Cg, or *C for Graphics*, is similar to the C programming language in

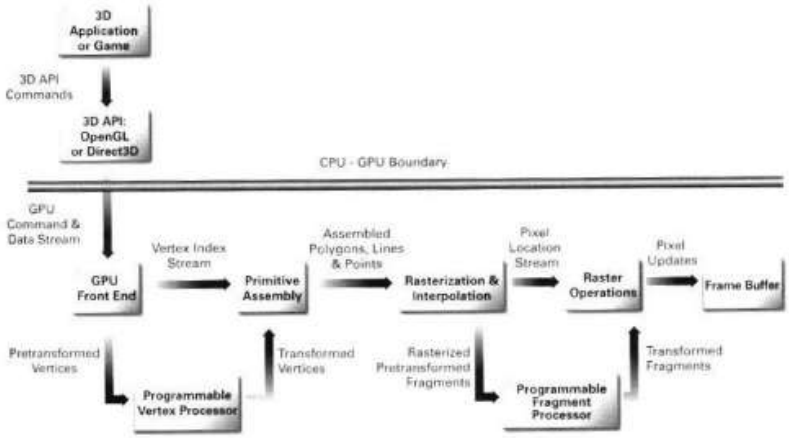

Figure 3: The graphics pipeline of a modern 3D card

syntax and provides the developer with high-level constructs such as vector and matrix variables, if-statements, for-loops and function calls. Library functions are provided for common functionality, such as calculating a dot product or matrix multiplication, which often run faster than a direct implementation since they can take advantage of special instructions in the hardware. The Cg program is compiled and optimized, usually at run-time, for the specific GPU hardware on which it is to be executed. The need for multiple implementations of the same program is thus greatly lessened.

4.2 GPU programming explained

As previously mentioned, Cg is a C-like high-level language for the development of vertex- and fragment shader programs. Two other such languages also exist, High Level Shading Language, HLSL, from Microsoft, which is identical to Cg, but implemented for the Direct3D API rather than OpenGL, and OpenGL Shading Language, OGSL, the shading language defined in the OpenGL 2.0 specification. Cg has been used exclusively during the research described here.

Although Cg is C-like in syntax it it quite different from writing normal C code, as the language has been adapted specifically for graphics processing. One of the most significant difference is the introduction of a number of additional data types supported directly in the language. This includes vector and matrix types of different sizes, and support for common operators working on these types. Depending on the available hardware there may also be limitations on the number of temporary registers and instructions, and, as branching is generally not supported, loops have to be unrolled by the compiler; the number of iterations must

thus be known at compile time.

Every Cg program must write an output, and may also take one or more inputs. In the case of vertex shaders the input is the vertex coordinate, and additional per-vertex data, such as vertex color, texture coordinates, and any data provided by the host application, may also be accessed. The output is a new vertex coordinate, transformed from local space to clip space, and new color and texture coordinates may also be included.

The input to the fragment shader is the interpolated output from the vertex shader as well as any variables and textures supplied by the host application. The fragment shader may then access the textures and perform calculations necessary to determine the final pixel color. Memory access can be simulated by *dependent texture lookups*, where the texture value from one texture is used as a coordinate to look up a value in a second texture. The output from the fragment shader, a color value, is written to the framebuffer, unless the fragment fails one of the OpenGL tests, such as the *depth test*.

By far the most floating-point calculation power lies in the fragment engine, and applications wishing to take advantage of this GPU capability often do not make use of a vertex shader; a simple pass-through shader is used. This is the case for this research as well, and only fragment shaders will be described in the following.

4.2.1 Phong shading

One of the most important aspects of producing realistic renderings is the lighting model used. One of the most popular such models is the Phong illumination model, first described by Bui-Toung Phong at the University of Utah [12], which, although it has no physical basis, gives very realistic-looking images. When the Phong equation is evaluated on a per-vertex basis and the intermediate pixel values interpolated from the vertices, the technique is known as *Gourad shading*; evaluating the equation for each fragment is known as *Phong shading*; both use the Phong illumination model. Phong shading is much more computationally expensive, however, often as much as two orders of magnitude, and has not been feasible for interactive rendering until the advent of fragment shaders.

The Phong illumination equation is relatively simple and its implementation is straight-forward. While an in-depth explanation of

Phong shading is not within the scope of this text, a brief description is given. The equation is a sum of four light components:

$$I = k_e + k_a + k_d(\bar{N} \cdot \bar{L}) + k_s(\bar{R} \dot{\bar{V}})^n$$

where k_e is the emissive component, describing light produced by the object; k_a is the ambient light, which is independent of direction; k_d is the directional light, coming from a point light source; N is the normal vector; L is the light vector; k_s the specular component; R the reflected ray vector; V the viewing vector and n the specular reflection exponent.

The specular component is essentially the shininess of the object, which produces the well-known specular highlight as seen from a position along the direction of reflected light illuminating a surface. The specular contribution is highest in this direction, and drops off quickly when the angle changes, which is why this highlight can be seen more clearly on a sphere than a flat surface.

The following is an example of Phong shading implemented in Cg:

```
// Get the position and interpolated normal
float3 P = position.xyz;
float3 N = normal;

// The emissive component
float3 emissive = Ke;

// Calculate the ambient component
float3 ambient = Ka * globalAmbient;

// Calculate the diffuse component
float3 L = normalize(lightPosition – P);
float diffuseLight = max(dot(N, L), 0);
float3 diffuse = Kd * lightColor * diffuseLight;

// Calculate the specular component
// H is used as an approximation to R
float3 V = normalize(eyePosition – P);
float3 H = normalize(L + V);
float specularLight = pow(max(dot(N, H), 0), shininess);

// If the diffuse component is zero the specular
// component is zero as well
```

```
if (diffuseLight <= 0) specularLight = 0;
float3 specular = Ks * lightColor * specularLight;

// Sum the components to form the final pixel color
color.xyz = emissive + ambient + diffuse + specular;
```

A rendering of two spheres using the above fragment shader is shown in Figure 4.

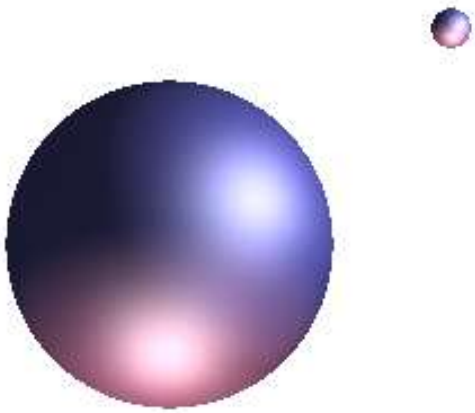

Figure 4: Two spheres rendered with Phong shading. There are two light sources in the scene, one blue and one red, giving the famous specular highlights.

Although the fragment shader code is relatively short, it implements a complete and realistic-looking lighting model. As a matter of fact, fragment shaders are usually short; their power comes from the GPU's ability to execute them very quickly. Simple, but computationally expensive, algorithms which previously were too slow to be considered for interactive rendering may well be feasible when implemented on the GPU.

4.3 A more complex example: Environment mapping

Environment mapping is a technique for making an object's surface look reflective and chrome-like, as in Figure 5, by simulating the reflections of the objects surrounding it.

All recent GPUs support a type of texture called a *cube map*, which consists of six square texture images representing the six faces of a cube. This type of texture is accessed by three values, representing a vector emanating from the center of the cube, and the color value

returned is the intersection of this vector with one of the six textures.

By replacing an object with a camera and taking six snapshots, one in each direction, a complete picture of the scene, as seen from the object, is generated and can be encoded in a cube map, also called an environment map. When the object is then rendered, the reflected view direction ray of each fragment on the object's surface is used to perform a lookup in the map; this is similar to *ray tracing*, where a number of light rays are shot into the scene and traced as they are reflected from object to object. In this case there is only a single reflection, however.

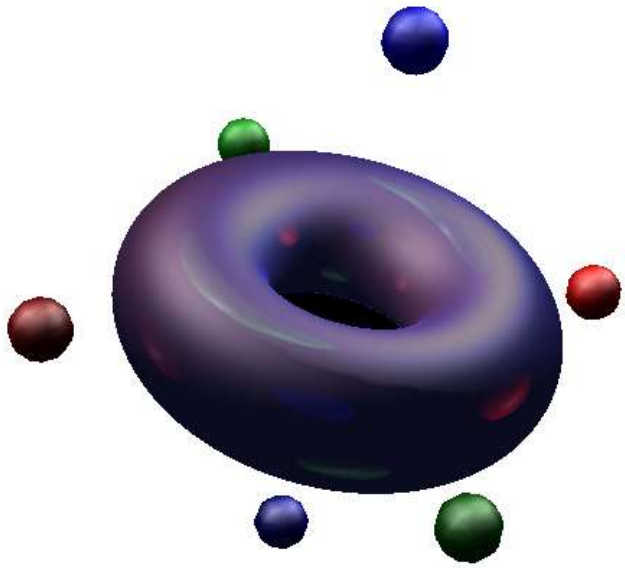

Figure 5: A rendering of a torus surrounded by six spheres. Environment mapping is used to make reflections of the spheres visible on the torus surface, and the scene is also Phong shaded.

What we now see is not only the object's surface, but also how the object reflects its environment. This is similar to a mirror, but as only the direction, and not the position, of reflected rays matter, this technique works best for curved surfaces where distortions mask the error this introduces, making the scene appear believable although it is not physically accurate.

When implementing this it is necessary to calculate the reflection vector from the view and the normal vector. While this is easy to do manually, it is better to use Cg's built-in function for this as it will execute faster in most cases. The code can then be written as:

```
I = positionW – eyePosition;
R = reflect(I, N);
```

Here *I* is the incident vector, *positionW* is the position, in world space coordinates, of the fragment on the object's surface, *N* is the normal vector and *R* the reflected vector.

Chapter 5

Near neighbor approach to GPU-based volume rendering

Near-Neighbor Approach to GPU-based Volume Rendering

Abstract

Volume rendering via 3D textures has proven to be an efficient approach to interactively visualizing and exploring volumetric data sets. By taking advantage of the Graphics Processing Unit (GPU) on modern graphics hardware, well-known acceleration techniques, such as early ray termination and empty-space skipping, can be integrated into the renderer.

This paper introduces the integration of the near-neighbor acceleration technique in ray casting into a GPU-based volume renderer, and how this affects the fragment culling efficiency. Here, the distance to the nearest interesting voxel is encoded and allows for skipping over uninteresting sections of the volume. We demonstrate a 15-30% reduction in fragments processed in the main pass, and almost an order of magnitude reduction in the intermediate pass.

The proposed technique has been implemented on the nVidia GeForce 6800 graphics card, which supports the `GL_EXT_depth_bounds_test` *OpenGL extension required for fine-grained control of fragment culling.*

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – color, shading and texture — ; I.3.8 [Computer Graphics]: Applications —

Keywords: Volume Rendering, Programmable Graphics Hardware, Ray-Casting

1 Introduction

Volumetric rendering typically involves large data sets whose visualization even strain current state-of-the-art computer systems. General-purpose CPUs have proved inadequate for interactive rendering of such large volume data sets, and significant effort is directed towards finding methods of utilizing specialized graphics hardware to improve rendering performance.

The most common way of exploiting graphics hardware for speeding up volume rendering involves taking advantage of the texture mapping capabilities of commodity 3D graphics cards. The volume data is re-sampled onto so-called "proxy geometry", either represented as stacks of axis-aligned planes with 2D textures mapped onto them, or planes aligned orthogonal to the view vector in the case of using 3D textures. The volume is rendered back-to-front and the rendering of each plane, or slice, is blended into the frame buffer. An important reason for performing the volume rendering on the graphics card is to take advantage of their dedicated hardware for bilinear (trilinear for 3D textures) interpolation, a costly operation to perform on the CPU.

More recently, the broad availability of programmable graphics processors on commodity graphics cards has enabled the implementation of a more traditional approach to volume rendering, ray-casting, using the card's Graphics Processing Unit (GPU). The volume is rendered front-to-back, and individual control of the processing of each ray makes it possible to do important optimizations such as early ray termination and empty-space skipping. Many volume data sets contain large amounts of empty space, and a significant number of voxels do not contribute to the final image; without such optimizations much unnecessary work is done. For such volumes, it is desirable to avoid processing these voxels, and, as the voxel sampling is done in a fragment program, it is necessary to prevent its execution if the result is known to not be of use.

The early z-test is the normal way of avoiding the execution of fragment programs. Here the incoming fragment's depth value is compared to the value in the depth buffer. If the depth test fails, the fragment is rejected without running the fragment program. However, as fragment programs can have side effects, it is necessary for all other per-fragment tests to be disabled and for the fragment program to not write a depth value if the early z-test is to take place. A fragment program writing depth would, however, be able to control which fragments are rejected in the next rendering pass. A significant speed-up can be realized by rejecting fragments in this manner. The rate at which different GPUs can reject fragments varies, but this is of less concern since fragment programs for ray casting are typically long.

Very recent GPUs have another way of rejecting fragments, by way of an OpenGL extension, the `GL_EXT_depth_bounds_test`, which is not dependent on the incoming fragment's depth value. It allows the user program to specify a range in the interval [*0, 1*], and only fragments whose value in the depth buffer fall inside this range are processed. This makes it safe for the GPU to reject fragments even if the fragment program writes a depth value, and this enables a finer control of the rejection of fragments beyond merely turning the processing in the next pass on or off. This paper explores how this feature can be used to reduce the number of fragments processed by the GPU.

The remainder of this paper is organized as follows. Chapter 2 presents background and related work; 3D texture-based volume rendering on proxy geometry; GPU-based ray casting; and using the early z test to perform early ray termination and empty-space skipping. Chapter 3 presents the near-neighbor approach and how it can be used to reduce the number of processed fragments. Results and discussion are presented in Chapters 4 and 5, and,

finally, conclusions and future challenges are given in Chapters 6 and 7, respectively.

2 Background and related work

Volume rendering on modern 3D hardware is usually performed using 3D textures rendered on proxy geometry, most commonly a stack of planes oriented perpendicular to the viewing direction, see Figure 1.

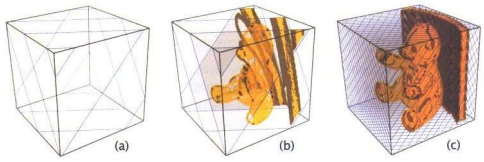

Figure 1: *Volume rendering using 3D textures on proxy geometry. Illustration taken from The Cg Tutorial.*

Each plane is clipped against the volume bounding box and the 3D texture applied to its surface using hardware-accelerated trilinear interpolation. The resulting image is blended into the color buffer. Normally the volume is rendered back-to-front as this does not require an additional α-buffer. The blending equation is:

$$C_{dst} = (1 - \alpha_{src}) C_{dst} + \alpha_{src} C_{src}$$

C_{dst}, α_{dst}, C_{src} and α_{src} are the color and opacity of the color buffer and of the incoming fragment, respectively.

A disadvantage of using planar proxy geometry is that with a perspective projection the sampling rate varies from pixel to pixel. It is in principle also possible to use other types of geometry than a plane, however. Using spherical shells is the best approximation to volume ray casting, but this is not used in practice due to the large number of triangles needed to approximate this shape.

The biggest disadvantage to back-to-front rendering on proxy geometry is that normal volume acceleration techniques, such as early ray termination and empty-space skipping, cannot be used. In a typical volume rendering a large number of voxels do not contribute to the final image, as pixels are saturated before the end of the volume is reached, or the voxels are completely transparent. This is especially true as volume visualizations often emphasize boundary regions and make structures of less interest transparent, and much time is essentially wasted processing invisible or occluded voxels. This waste is increased further when lighting calculations are taken into account; in the evaluation of the lighting equation normally a gradient texture is sampled for each voxel and most of the texture fetches and calculations will have no effect on the final image.

2.1 Ray casting on the GPU

The state-of-the-art of graphics hardware has advanced enormously in recent years, and while early hardware only supported a fixed-function pipeline, contemporary graphics cards feature a fully programmable pipeline enabling the implementation of arbitrary functionality. In fact, a stream model for performing ray-tracing on GPU hardware was proposed in [Pur04], and more recently in [BFH*04] a more general framework for virtualizing the GPU to facilitate general-purpose programming.

Processing of fragments is an inherently parallel process and GPU performance is currently advancing at a much higher rate than general-purpose CPUs. Taking advantage of this added processing power is therefore a very worthwhile cause and has the potential of significantly speeding up interactive renderings. In each rendering pass the same fragment program is executed for each incoming fragment generated by the rasterizer in a SIMD manner; textures are used for providing input data.

2.2 Ray casting with early z-testing

This algorithm was first described in [KW03]. A summary is included here for completeness and for introducing the method for calculating the ray direction vector passing through each voxel.

The algorithm is a multi-pass approach in which a fixed number of samplings along each ray is performed in each pass, and a fixed number of passes is done depending on the desired sampling rate. Ray traversal is aborted when the ray exits the volume bounding box, the opacity exceeds a given threshold or a selected iso-value is reached.

Before ray traversal starts the ray direction vector passing through each voxel is calculated and stored in a 2D texture along with the length of the ray. This texture is used in subsequent rendering passes for calculating the sampling points along the ray. In each rendering pass the samplings done are blended together and finally blended with the value from the previous rendering pass and written to a 2D texture.

Between ray traversal passes an intermediate pass is performed to determine which fragments should be rejected due to early ray termination. If a ray is to be terminated, because of the opacity exceeding a threshold or an iso-value is hit, the depth value is set to zero which occludes any geometry and prevents the execution of the fragment program for that fragment. Otherwise, it is set to one and the fragment is processed in the next rendering pass. Calculating the 2D RGBA texture (*DIR*) storing the ray direction vectors and vector lengths is done in two steps.

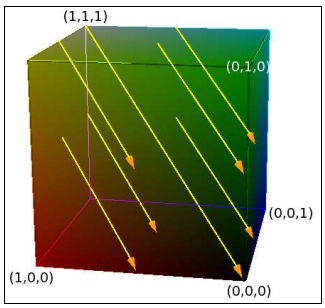

Figure 2: *Ray direction and first ray intersection is calculated by rendering the front and back faces of the volume bounding box*

First the front faces of the volume bounding box are rendered to a 2D texture (*TMP*), where the color for each vertex of the box is set equal to its coordinates, such that the interpolated color for any point on the cube equals the coordinates for that point.

In the next step the back faces of the same bounding box are rendered and a fragment shader is issued which calculates, for each fragment *xy*, the normalized ray

direction vector as *normalize*($COL_{xy} - TMP_{xy}$) *(see Figure 2)*, where *COL* is the fragment color. The vectors are stored in the RGB components of the target texture *(DIR),* while the alpha component is used for storing the length of each vector before normalization.

The actual rendering now commences, and is performed in a number of passes where each pass consists of a main and intermediate pass.

- **Main render pass** (*ray traversal*): In this pass *M* samplings are performed along each ray, and the output of the rendering is copied to a 2D texture *RES* which can be accessed in consecutive passes.

  First the front faces of the bounding box are rendered and a fragment shader for ray traversal is issued. In addition to the *DIR* texture with the ray directions and the previously mentioned *RES* texture with the color and opacity value accumulated so far, the shader program takes three inputs: volume data from a 3D texture, the starting position *startPos*, which gives the current distance along the ray already traversed (in previous passes), and *stepSize*, which gives the distance between sampling points on the ray. The shader now performs *M* samplings along the ray, starting at *startPos* and then incrementing the position by *DIR[x][y] * stepSize*, where *x* and *y* are the current window coordinates, before each additional sampling. The sampled values are blended together and finally blended with the result from the previous pass read from *RES*. This value is then written as output from the shader.

  After the render pass is done the frame buffer contents is copied to *RES* in cases where direct Render-To-Texture (RTT) is not used.

- **Intermediate render pass** (*ray termination)*: This pass is performed after each main render pass (except for the last where it is not needed) and checks if the opacity value at the current ray position has exceeded a threshold *T*. If this condition is true the z value is set to zero and the fragment will not be processed in consecutive passes; otherwise it is set to one:

```
if (RESx][y]  > T)
        z = 0;
else
        z = 1;
```

  Note that if the ray has exited the bounding box the opacity will always be larger than *T* and the ray terminated. Writing to the color buffer is disabled while this shader is running and only the depth value is affected.

2.3 Empty space skipping

Many typical volumes consist of large areas of "empty space", that is, voxels deemed uninteresting and therefore completely transparent by the current transfer function. By empty-space skipping we seek to speed up rendering by avoiding sampling these empty regions.

To be able to skip over empty space additional data structures encoding information about which regions are safe to skip over are needed. There are several viable choices: an axis aligned octree whose inner nodes encodes information about the leaf nodes in its sub-tree can be used; the block skipping method described in the next section uses a data structure corresponding to a single level of such an octree encoding min/max bounds for each block. A different approach, the one presented in this paper, encodes the distance to the nearest interesting voxel, enabling the ability to skip larger distances without further checking.

2.4 Block skipping

This is the original empty-space skipping approach described in [KW03]. Disjoint blocks of size 8^3 (other values can be used, however, this was the maximum value supported by the hardware used in the original paper, and informal testing with higher values does not give significantly different results with typical volume data sets) within the original volume data set are encoded as another 3D texture, where each block encodes the minimum and maximum voxel value within that block stored in the R- and G-components of the texture map. A 2D texture encodes, for each min/max pair, corresponding to the *x* and *y* values, if there is at least one interesting voxel within the given range. This texture is generated on the CPU and updated whenever there is a change to the transfer function.

Empty space is detected by extending the intermediate pass to perform an extra check: if empty spaces is detected at the current position the z value is set to 0. The shader now essentially performs the following:

```
if (RES[x][y]  > T) OR (EmptySpace == TRUE)
    z = 0;
else
    z = 1;
```

Note that since this check is performed immediately before each main shader pass, sampling will always be re-enabled in the case empty space is not detected.

3 Near neighbor skipping

This approach tries to skip over larger regions of empty space by storing in an additional data structure the distance to the nearest interesting voxel. This is a well-known technique in software-based volume rendering, however, the goal in GPU based volume rendering is to be able to skip the intermediate rendering pass in many cases. This requires the ability to encode more information in the z buffer than simply "on" or "off"; essentially, we need to be able to specify that the processing of a given fragment should be turned off until pass *K* .

For this purpose the normal depth test is insufficient; the reason is that the fragment program cannot be skipped since it modifies the depth value. What is needed is a depth test independent of the depth value of the incoming fragment, which is exactly what the OpenGL extension `GL_EXT_depth_bounds_test` provides. It is controlled by the function call:

```
DepthBoundsEXT(zmin, zmax);
```

After enabling this test, the value from the depth buffer corresponding to an incoming fragment is compared to the range [*zmin*, *zmax*], and if it falls outside, the fragment is rejected. The depth value of the incoming fragment is ignored and it is for this reason safe to reject fragments even in the case where the fragment program alters the depth value.

While the depth bounds test does not interact directly with the normal depth test they still share the depth buffer and care must be taken to ensure that the depth values used do not occlude the drawing of the volume bounding box. For this reason the depth buffer range is split in two: the lower range is divided into *N* sections, one corresponding to each main shader pass, while the volume bounding box is drawn with depth values in the higher range. The normal depth test function is set to GREATER.

The distance to the nearest interesting voxel for any given voxel is encoded in a 3D texture map. Since this distance may depend on the ray direction there are several ways to calculate and use this data structure; Section 3.1 explores this topic in more detail.

The main shader pass remains unchanged from the description in Section 2.2, however, the intermediate pass is now performed *before* each main shader pass and reads the distance to the nearest interesting voxel. It then calculates in which shader pass this voxel will be rendered and sets the z value to within the range for this pass. The opacity test is still performed: if the opacity has exceeded the threshold *T* the z value is set to one, essentially turning off rendering of this ray for the remainder of the rendering passes.

Before each rendering pass the depth bounds range is adjusted to go from *0.0* to the end of the range for the current render pass. This way, only fragments whose depth value lies within this range will be processed, which in effect means that no processing will be performed for a given fragment before the distance along the ray set by the intermediate fragment program has been traversed.

The reason for still performing the rendering in two passes even though the `GL_EXT_depth_bounds_test` allows for skipping fragment programs which writes a depth value is due to a performance consideration of current GPUs. Although the nVidia GeForce 6800 supports conditional execution, its fragment engine is still SIMD (Single-Instruction, Multiple-Data) as opposed to the vertex engine which is MIMD [Har04]. The effect of this is that if there is a high number of divergent fragments (i.e. nearby fragments which take different branches) the fragments taking short branches may be limited by the execution time of nearby fragments taking long branches. A short shader program is therefore preferable.

3.1 Methods for calculating the near-neighbor textures

The near-neighbor data structure encodes the distance to the nearest interesting voxel from each voxel within the volume. Several methods can be used for calculating this data structure.

Omnidirectional: The simplest approach encodes a single value, the distance to the nearest interesting voxel in any direction, that is, the radius of the largest possible sphere centered on the voxel in question not containing any other interesting voxels.

This brute-force approach, calculating the distance to every other voxel from every voxel, runs in time $O(N^6)$, which is far too slow for any reasonable-sized volumes. The approach used for the purpose of this paper is a k-d tree [AMN*98], providing $O(N \log N)$ lookup time, which is reasonably quick even for volumes of size 256^3.

This data structure is independent of the ray direction, however, which has the disadvantage of considering neighbors interesting even if they do not lie on the current ray, in fact, even voxels directly behind the current position will be considered interesting. This significantly reduces the distance that can be skipped.

Optimal: The other extreme is a data structure encoding the distance to the nearest interesting voxel along a specific direction vector. This ensures the maximum possible distance can be skipped, however, the data structure will have to be re-calculated every time the viewing direction changes. It is not necessary to re-calculate when zooming in or out of the volume, however.

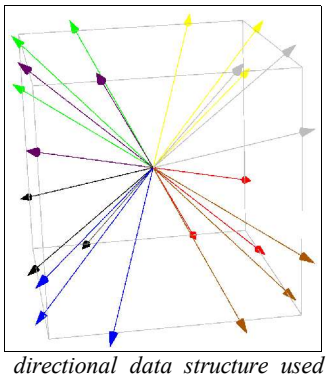

Figure 3: *A directional data structure used depends on which of the 8 quadrants the viewing vector is pointing into*

Directional: A middle ground is to divide the volume into 8, with one data structure corresponding to each of the 8 quadrants, see . Before the fragment shader is issued, the correct near-neighbor 3D texture for the current viewing direction is bound and used in the intermediate rendering pass. While less optimal than encoding the distance only along the viewing direction it is a marked improvement from the omnidirectional near-neighbor.

The 8 data structures can be calculated as follows: For each of the 8 corners of the volume bounding box, traverse backwards along the vector from the corner to the opposite corner. In each step, add all interesting voxels within the sub-cube from the corner to the current position on the vector to a k-d tree. Then, for each voxel on the inner faces of the sub-cube (laying within the volume), set the distance to the nearest interesting voxel by looking this up in the k-d tree.

3.2 Iso-surface rendering

Rendering a single iso-surface can be done very effectively with GPU-based ray casting, by only performing the actual illumination calculations in one final pass after first determining the intersection between the iso-surface within the volume and each ray.

The rendering setup is the same as before, but the main pass is simplified. The current ray segment is traversed back-to-front and for each sampled value we check if it

falls within a given range; if it does the position of the value within the volume is written to the output with an alpha value of 1 in order to terminate the ray. After all rays have terminated we have stored the intersection point for each ray, and this is used as input to a final render pass where the value and gradient at the given point is sampled and lighting calculations are performed; the output of this pass is the final color value.

The reason for only performing the lighting calculations in a separate shader program is the limitation of the conditional execution of current GPUs, as previously mentioned. As performance is limited by the longest branch a short shader program is preferable.

3.3 Using an occlusion query to avoid updating the target texture

An occlusion query, like the OpenGL `GL_ARB_occlusion_query` extension, can be used to obtain the number of fragments actually processed. If this number is zero for a given pass the copy from the frame buffer to the target 2D texture used for accumulating the result of the rendering can be skipped. For platforms which do not support a direct Render-To-Texture mechanism this can result in a significant speed-up.

3.4 Platform requirements for near-neighbor skipping

While the required features for performing ray casting and exploiting the early z-test for rejecting fragments are present in all third-generation GPUs, like the ATI R300 and nVidia NV30 architectures, the `GL_EXT_depth_bounds_test` extension is only available in the third and forth generation GPUs from nVidia. The following features are essential for the approach proposed in this paper.

- Per-fragment texture fetching: The number of textures available for use in a fragment program varies with the architecture, however, this limit is generally high enough for this not to be an issue. Dependent texture lookups can cause pipeline stalls and slow down fragment processing, however.
- Render to texture: While it under some systems is possible to render directly to a 2D texture map instead of the frame buffer, depending on available hardware and graphics driver, on other systems without this capability it is necessary to copy the frame buffer contents to a texture after rendering. In both cases the texture is available to the fragment program in the next rendering pass and enables the passing of data between rendering passes. Values are range-compressed to fit in the [0, 1] range.
- Per-fragment arithmetic operations: The most common, as well as several more complex, arithmetic operations are available for both scalar as well as vector values. This enables manipulation of the input data provided in texture maps as well as conditional writing of output values.
- Depth write: The fragment program can write an arbitrary value to the depth buffer.
- `GL_EXT_depth_bounds_test` extension, as previously mentioned.

An nVidia GeForce 6800 graphics card was used for implementing the method proposed in this paper.

4 Results

To evaluate the effectiveness of the near-neighbor approach it is insightful to look at the number of fragments actually processed in the main and intermediate passes. For the purpose of obtaining these numbers, the occlusion query extension was used, which counts the number of non-culled fragments. It should be noted that for the intermediate pass, the empty block skipping will always process all fragments; this is because the fragment program writes a depth value and thus cannot be skipped. However, even if this restriction was lifted it would still be necessary to run the fragment program in order to check for empty space.

Using the `GL_EXT_depth_bounds_test` extension allows the fragments in the intermediate pass to be skipped for passes with no interesting voxels along the fragment's ray, and this significantly reduces the total number of fragments processed.

Fragment counts for four volume data sets are presented here: `MRI-Head`, an MRI scan of a human head; `Foot`, a CT scan of a human foot; `Engine`, a CT scan of an engine block; and `Fuel`, a simulation of fuel injection into a combustion chamber. The first three are of size 256^3, while Fuel is of size 64^3. Rendering was directed to a 500^2 viewport. The renderings in Figure 4 are courtesy of the V^3 renderer [Roe04]; for the fragment counts presented here, a simple linear transfer function was used.

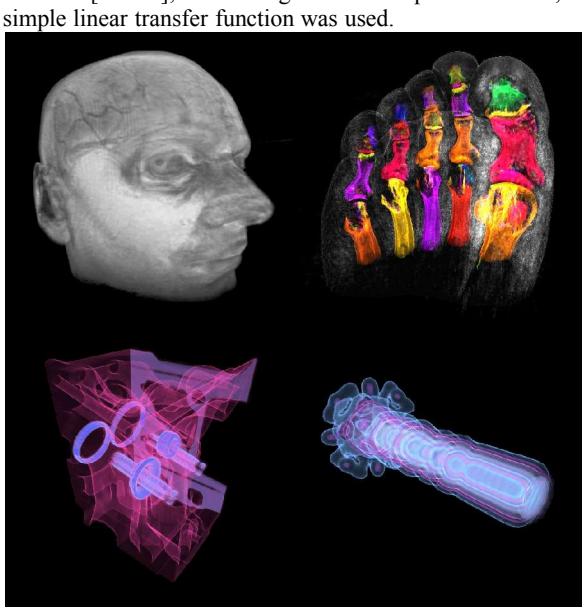

Figure 4: *Volume data sets used for rendering. From top, left to right: MRI-Head, Foot, Engine and Fuel.*

In Table 1 and Table 2, the number of fragments processed in the intermediate and main passes, respectively, is given for the three different data structures used to determine the nearest neighbor; the percentage in parenthesis is the the fraction of fragments

processed compared to empty block skipping.

	MRI-Head	***Foot***	***Engine***	***Fuel***
Empty block	9,891,936	9,891,936	9,891,936	2,381,392
Optimal	534,272	427,510	274,393	244,118
Omni-directional	1,218,485 (12%)	2,134,119 (22%)	1,552,627 (16%)	1,106,584 (46%)
Directional	1,458,291	1,856,402	1,367,713	916,156

Table 1: *Fragments processed in the intermediate pass*

	MRI-Head	***Foot***	***Engine***	***Fuel***
Empty block	869,293	538,173	309,097	299,356
Optimal	181,139	159,380	65,927	31,220
Omni-directional	744,754 (86%)	455,547 (85%)	186,059 (60%)	155,011 (52%)
Directional	725,099	410,729	178,331	131,495

Table 2: *Fragments processed in the main pass*

5 Discussion

When using an optimal near-neighbor data structure, more than an order of magnitude improvement to the number of fragments processed can be achieved in the intermediate pass, a five-fold improvement in the main pass. It should be noted, however, that the near-neighbor data structure is the same size as the volume and thus significantly higher resolution than the encoding of the empty blocks.

As noted earlier using an optimal near-neighbor data structure has the disadvantage of needing to re-calculate the data structure whenever the viewing direction changes. A more fair comparison is the use of either an omnidirectional or directional data structure. The improvements over the empty block approach are not as dramatic for the main pass, however, the gain in the intermediate pass is still almost an order of magnitude for the larger volumes. The smaller 64^3 volume does not show as much improvement, as the first intermediate pass, where also the near-neighbor approach must process all fragments in order to set the initial distance to skip, becomes more dominating.

The advantage of using a directional near-neighbor data structure, compared to an omnidirectional, is perhaps not as significant as could be expected; this is because the case where this data structure would be of the most benefit, a transparent surface with empty space behind it, is not present to any great degree in these volumes.

While the reduction in processed fragments is impressive, we have not been able to demonstrate a corresponding increase in frame rate, however. While reliable information is scarce, some sources indicate that the `GL_EXT_depth_bounds_test` cannot, in the current revision of nVidia graphics hardware and drivers, be used for avoiding the execution of fragment programs. This extension is part of nVidia's "UltraShadow" technology, however, and is supported in the popular 3D game "Doom III". Since the 3D graphics industry is currently being driven in large part by the needs of the game industry, there is a good probability that this limitation will be removed in the future. We thus do not consider this a limitation of the algorithm. GPU-based ray casting should in fact scale very well on future hardware able to execute a higher number of fragment programs in parallel.

Compared to slice-based volume rendering, ray casting also has another benefit, which is a correct perspective view, as the ray directions are individually calculated based on the current view perspective.

The choice of M, the number of samples to take along the ray in each shader pass, has a significant impact on performance. Since it is not possible to abort a fragment program before its completion, M should be small so as to not continue working for too long after the ray has been terminated. Conversely, M should be as large as possible to reduce the number of rendering passes necessary, as there is some overhead associated with each such pass, including drawing the front faces of the bounding box, the binding of textures and in the case Render-To-Texture is not available, copying the frame buffer to a 2D texture. A large M will, however, limit the effectiveness of empty-space skipping, since larger blocks or longer ray segments within each pass will increase the probability of at least one interesting voxel being present. The optimal value for M depends on several factors, such as the hardware used and the size of the volume.

For shading of opaque iso-surfaces, all illumination calculations are done only once for each ray, in a final pass. Although both intermediate and main passes are still required due to the limited conditional execution of current GPUs, as mentioned in Chapter 3, the rendering is significantly more efficient as compared to slice-based rendering, where lighting calculations have to be performed for each sampled voxel.

Finally, it should be clear that for highly transparent and dense volumes, where early ray termination and empty-space skipping cannot be applied, no gain in speed can be expected. In fact, some overhead is introduced and for such volumes the renderer should be switched to traditional slice-based rendering.

6 Conclusions

In this paper, we show a novel approach for integrating near-neighbor skipping in a GPU-based volume renderer, on graphics hardware supporting the `GL_EXT_depth_bounds_test` extension. Near neighbor skipping, when used in conjunction with ray casting implemented using the GPU of modern graphics hardware, can significantly reduce the number of fragments that needs to be processed.

The proposed approach has been implemented on the nVidia GeForce 6800 graphics card using OpenGL 1.4. While the NV35 and NV40 architectures from nVidia are currently the only lines of graphics cards supporting the `GL_EXT_depth_bounds_test`, it is expected that support will improve in the next generation of graphics hardware, especially as this extension also benefit the games industry.

While the number of processed fragments is reduced using the proposed approach there is unfortunately not a corresponding increase in performance as the `GL_EXT_depth_bounds_test` cannot be used for avoiding fragment program execution on current hardware and graphics drivers. Improved performance can be realized once this restriciton is lifted, as it is expected to be in future hardware.

7 Future work

While the optimal near-neighbor data structure does provide for a large reduction in the number of processed fragments, it needs to be re-calculated whenever the viewing direction changes, and this would introduce unacceptable delays for most interactive rendering.

One interesting possibility is calculating the distance to the nearest neighbor falling within a cone around the viewing direction instead of only along the ray itself. This would allow for some change of the viewing direction without the need for re-calculation of the data structure. When the viewing direction changes, a background task could be started to calculate a new data structure as seen from the location and viewing direction the user is expected to be at in the near future, if the viewing direction changes at a constant rate. Since the CPU is often idle while the GPU is performing rendering tasks, this would mean better utilization of available resources.

If the user changes camera position or viewing direction too quickly, there would still be a delay while the re-calculation is being performed, however. In this case, the renderer could switch to an omnidirectional data structure while waiting for the calculation to finish.

One other possible optimization when using the optimal near-neighbor data structure is combining the intermediate and main shader passes into one, except for the first pass, which is still the same as the intermediate pass. This should result in even faster rendering since, as the rendering always jumps directly to the next voxel to be sampled, when rendering resumes there will always be samples to take for that ray segment and the intermediate pass does not result in the subsequent main pass being skipped for any fragments.

Finally, it should be noted that at least early ray termination is possible to integrate into a slice-based volume renderer. The render direction must then be done front-to-back, which does incur an overhead to maintain the accumulated transparency value. There is, however, no intermediate pass, and thus the check for saturated opacity is done for every fragment. The efficiency of this method depends on the volume: a mostly opaque volume should give a significant speed-up.

For integrating empty block skipping, an intermediate pass processed only every M render passes could check for empty space, as well as saturated opacity, and disable rendering in the normal pass. It is uncertain whether this will give any speed-up over ray casting, however.

References

[AMN*98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman and A. Y. Wu. 1998. "An optimal algorithm for approximate nearest neighbor searching". In *Journal of the ACM,* Volume 45 (1998), p. 891-923.

[BFH*04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston. "Brook for GPUs: Stream Computing on Graphics Hardware". In *Proceedings of the 2004 SIGGRAPH Conference*, p. 777-786.

[Fer04] nVidia Corporation; edited by R. Fernando. "GPU Gems". 2004. ISBN 0-321-22832-4.

[FK03] R. Fernando and M. J. Kilgard. "The Cg Tutorial". 2003. ISBN 0-321-19496-9.

[Har04]]M. Harris. Comment on dynamic branching on the nVidia GeForce 6 series. http://www.gpgpu.org/forums/viewtopic.php ?p=1694#1694

[KW03] J. Krüger and R. Westermann. "Acceleration Techniques for GPU-based Volume Rendering". In *Proceedings of the 14th IEEE Visualization Conference*, p. 38, 2003.

[LCN98] B. Lichtenbelt, R. Crane and S. Naqvi. "Introduction To Volume rendering". 1998. ISBN 0-13-861683-3.

[Pur04] T. J. Purcell. "Ray Tracing on a Stream Processor". Ph.D. dissertation, Stanford University, March 2004.

[Roe04] S. Roettger. V^3: The Versatile Volume Viewer. http://www9.cs.fau.de/Persons/Roettger/ #Volume

[SWND04] D. Shreiner, M. Woo, J. Neider and T. Davis. "OpenGL Programming Guide, Fourth Edition". 2004. ISBN 0-321-17348-1.

Chapter 6

Transparent Polygonal Geometry in Volume Rendering

GPU-based Transparent Polygonal Geometry in Volume Rendering

Abstract

Utilizing the Graphics Processing Unit (GPU) of modern graphics cards has proved an efficient way to interactively visualizing volumetric data sets, enabling acceleration techniques such as early ray termination and empty-space skipping. However, integrating support for transparent polygonal geometry embedded in the volume presents some additional challenges.

This paper presents a novel approach which combines GPU-based ray casting with the "depth peeling" technique. This makes it possible to correctly render transparent polygonal geometry embedded in a volume data set in an order-independent manner. The proposed approach takes advantage of depth culling, and, compared to normal volume rendering, does not require additional sampling of the volume data. We also show how the technique can be used to simulate arbitrary clip geometry. This is especially important for ray casting, since the clipping planes provided by OpenGL cannot then be used directly.

The proposed technique has been implemented on the nVidia GeForce 6800 graphics card, which supports the `GL_EXT_depth_bounds_test` *OpenGL extension required for fine-grained control of fragment culling.*

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – color, shading and texture — ; I.3.8 [Computer Graphics]: Applications —

Keywords: Volume Rendering, Programmable Graphics Hard-ware, Ray-Casting, Transparent Geometry, Mixing polygons and volumes

1 Introduction

Volume rendering is the process of projecting 3D volumetric data onto a 2D image plane. Many fields require the visualization of such data, perhaps most notably in the medical imaging industry as well as seismologic surveys used in the search for pockets of hydrocarbons (oil & gas reserves). Volume rendering has been extensively described in the literature, and several techniques exist for their visualization [LCN98]. This paper is concerned with how transparent polygonal geometry can be integrated in a volume renderer, both traditional slice-based rendering and GPU-based ray casting [KW03].

Often volumes are visualized without any other geometry being present in the scene, and for many applications this is acceptable. However, some applications can benefit from being able to embed polygonal geometric structures inside the volume, and the traditional slice-based approach does allow for this as long as these structures are opaque.

Even for traditional polygon-based rendering, the integration of transparency is not straight-forward, since several color values must be blended together in the correct order. Rendering the scene back-to-front by sorting the geometry is the simplest approach. However, with this technique the sorting process can be costly if the number of geometric primitives is large, and can even be incorrect in the case of intersecting primitives. Geometry sorting can be combined with slice-based volume rendering [KK99], however, significant pre-processing is required in addition to the sorting itself.

Another technique, *depth peeling* [Eve01] has only recently been possible to implement with hardware acceleration on consumer-level 3D graphics cards. It which renders the individual "layers" of a scene to textures which are then blended together to form the final image. Since rendering a high-quality representation of a volume of size N^3 can require as much as $2N$ slices, or layers, it is not efficient to apply depth peeling directly. A better way is to first render the scene without the volume, and then blending in the resulting textures while rendering the volume in a final step. This is the approach which is explored in this paper, as well as how the technique can be used to simulate arbitrary clip geometry for volume rendering.

The rest of this paper is organized into the following sections: Section 1.1 introduces volume rendering, while in Section 1.2 and 1.3 we briefly describe slice-based volume rendering and GPU-based ray casting, respectively. Section 1.4 describes challenges faced when combining volume rendering with polygonal rendering, and section 1.5 introduces the depth peeling technique. In Chapter 2 the depth peeling technique and its integration with volume rendering approaches is presented in detail. Chapter 3 describes how the same technique can be used to simulate arbitrary clip geometry, Chapter 4 presents the results, Chapter 5 contains discussion, and conclusions and future work are given in chapters 6 and 7, respectively.

1.1 What is volume rendering

Unlike polygonal geometry, which only requires the rendering of surfaces, volume data consists of samples distributed along all three dimensions; each such sample is called a *voxel* and, in addition to the voxel value itself, has a 3D position associated with it. Some of these voxels can be fully or partially transparent, in which case the voxels located behind it, as seen from the camera, must also be considered. The voxels are combined with one of several possible operators; the most common are back-to-front and front-to-back where the values are blended together in the

specified order.

As consumer-level 3D accelerator cards have become common, several techniques have been devised for using the additional computational resources they provide towards speeding up volume rendering. These techniques typically make use of either 2D, or more recently 3D, textures and the hardware's ability to perform interpolation. The following sections explain the traditional slice-based approach as well as ray casting, which has only become possible on very recent 3D cards including a *Graphics Processing Unit* (GPU).

1.2 Slice based volume rendering

The volume is rendered by mapping a 3D texture of the volume data inside the volume bounding box, and then drawing slices of planes called "proxy geometry", which is clip against the bounding box and oriented perpendicular to the viewing direction. The 3D hardware samples the texture using tri-linear interpolation and draws the texture values on top of the plane and the result is blended into the frame buffer. The planes are most commonly drawn in back-to-front order, and, if a high enough number of planes is used, the resulting image is of high quality.

1.3 Ray casting on the GPU

By utilizing the GPU found on modern 3D cards, ray casting, the most common approach and which usually is found in software renderers, can be implemented with high enough performance for interactive rendering. A 2D texture, of the same size as the viewport, contains the ray direction for each pixel of the front faces of the volume bounding box. A fragment program is then issued which, for each such pixel, takes samples along the vector read from the 2D texture and blends these samples together. The ray is divided into segments, and the rendering is performed in multiple passes; in each pass the value of the partial ray is blended into the frame buffer. Rendering is performed in front-to-back order, which enabled such techniques as early ray termination and empty-space skipping, significantly enhancing rendering performance for volumes highly opaque or consisting of large regions of uninteresting voxels. An added benefit over slice-based rendering is a correct perspective view.

1.4 Challenges in combining volume data with polygons

Volumetric data and polygons, although both are visualized as shapes in 3D space after rendering, use very different data structures internally. Volume data sets consist of a regular grid of sampling points in 3D space, while polygons are described by the 3D coordinates of their vertices.

Combining volume rendering with polygon rendering presents some challenges as most graphics hardware does not directly support volume rendering and clever tricks have to be employed to perform such rendering.

1.4.1 Rendering with 3D textures

When rendering the volume on "proxy geometry", normally planes aligned perpendicular to the viewing direction, this geometry is rendered in its correct location in the scene, and the normal depth buffer is used and OpenGL will handle interaction with any other geometry present. If some of the geometry intersects with the volume, however, it is necessary for the volume to be rendered after all such intersecting geometry, and the geometry must also be opaque. If it is desirable to render transparent geometry intersecting with the volume a technique knows as "depth peeling" can be used as we shall see in section 1.5.

1.4.2 Rendering with ray casting

When performing volume rendering with GPU-accelerated ray casting, the normal OpenGL depth test cannot be used for detecting intersecting geometry within the volume, as only the front faces of the bounding box are actually rendered. However, the depth test can instead be performed manually inside the fragment program as the ray is traversed.

The scene is first rendered normally without the volume, and the depth buffer is then copied to a "depth texture", also known as a shadow map. The fragment program takes an additional input, the depth value at the current location along the ray, and before each sampling it is compared to the value in the depth texture; if it is greater than the depth texture value the ray is terminated. After all rays have terminated, the volume is blended with the contents of the frame buffer containing the scene rendered without the volume.

As with slice-based rendering, any geometry intersecting with the volume must be opaque. By using "depth peeling", however, this restriction can be lifted.

1.5 Depth peeling

Depth peeling is a method for enabling order independent transparency. Basically, it involves rendering the scene multiple times, one for each layer, and after each pass the contents of the depth buffer is copied to a depth texture. This texture is accessed in the subsequent render pass and is used to "peel away" the current surface. What results is essentially a stack of depth textures giving, for each fragment in the viewport, the intersections with scene geometry along the vector in the viewing direction, as well as the color value of the geometry at the intersection points. The color values can then be composited together in

Figure 1: *Correctly rendered objects with multiple transparent layers using depth peeling*

back-to-front order using the following blending function:

(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA)

The resulting image will have correct transparency; Figure 1 shows an image rendered using the depth peeling technique.

2 Transparent geometry intersecting the volume

As previously mentioned, by conventional means it is only possible to render opaque geometry intersecting the volume. The reason for this is that it is impractical to guarantee that fragments for a given pixel will arrive in back-to-front order. This is especially true for slice-based volume rendering where the volume data is rendered on a stack of planes with minimal spacing between them. For correct transparency it would be necessary to separately render only the geometry falling in between two such planes, a difficult task in the general case [KK99]. By integrating depth peeling into the volume renderer this can be avoided.

2.1 Depth peeling and slice-based volume rendering

Since slice-based volume rendering is performed by rendering the volume data on "proxy geometry", normally a stack of planes, it contains a high number of "layers" and depth peeling all these layers would be very inefficient. A better approach is to perform the following steps:

1. Apply the depth peeling technique to the scene without rendering the volume
2. Set the depth function to GREATER
3. Starting with the second last (furthest away from the camera) depth texture, copy it into the depth buffer.
4. Render the volume slices normally in back-to-front order
5. Starting with the last frame buffer texture, blend it into the frame buffer.
6. Repeat from step 3 with the next texture pair if there are more left.

After step 1 is complete we have a stack of texture pairs, the contents of the depth buffer and frame buffer at each "layer" in the scene, see Figure 2. The contents of the depth buffer and frame buffer are that of the last layer, and the depth comparison function is set to GREATER, meaning that any subsequent volume rendering pass will only include geometry found behind the layer present in the depth buffer. The second last depth texture is copied into the depth buffer, meaning only the proxy geometry between between the last and second last layer layer will be rendered. Note that it is important for the last layer to be located behind the volume bounding box as seen from the camera; if this is not the case the color buffer must be cleared and the last depth texture used instead of the second last. After the volume rendering pass the color values from the next layer are blended into the frame buffer, the depth buffer moved to the next layer and the next section of the volume is rendered. This process is repeated until all sections of the volume have been rendered, and the resulting image will be correct with regards to transparency; examples are given in Figure 3.

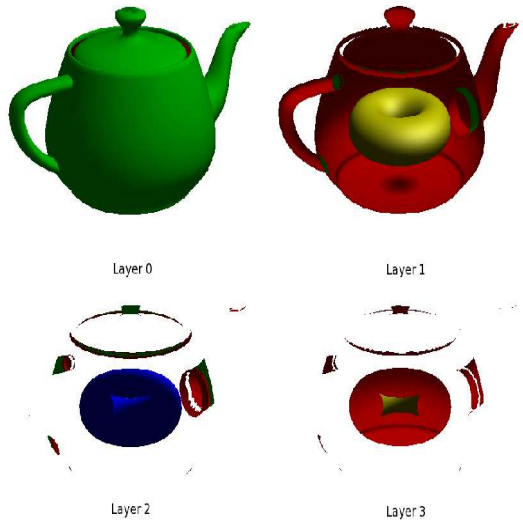

Figure 2: *The first four layers of a scene. The teapot is green outside and red inside, while the torus is yellow outside and blue inside. Note that the red color in the torus shape in layer 3 is from the teapot, not the torus.*

2.2 Depth peeling and GPU-based ray casting

The GPU-based ray casting algorithm differs from the slice-based volume rendering in that the depth buffer is only used for controlling which fragments should not be rendered due to early ray termination or skipping of empty space. The "proxy geometry" in this case, the front faces of the volume bounding box, is always rendered in the same location in the scene, and the depth test in OpenGL cannot be used for only rendering the parts of the volume behind a given layer. While it is possible to move the bounding box in the scene (and enlarging it so it still occupies the same fragments of the viewport) the OpenGL depth test still cannot be used as multiple samples along each ray are taken in each rendering pass. The depth test can instead be done inside the fragment program, however; the depth texture is given as input to the fragment program instead of being copied into the depth buffer, and, before each sampling, the depth value of the current position along the ray is compared to the value in the depth texture. The ray is terminated when the boundary into the next layer is crossed. There are some other considerations, however:

- Ray casting is done in front-to-back order, and the layers must be processed in this order as well. The first (closest to the camera) frame buffer texture is copied into the frame buffer before rendering, and the second closest depth texture is bound. The fragment program then checks if the current depth value is less than the depth texture, and terminates rendering when this condition is false.
- The frame buffer texture is blended in by a separate fragment program after each rendering of the volume.
- If a layer intersects a ray in the middle of a ray segment the rendering must be restarted at the correct location within the ray segment. For this reason the previous depth texture must also be

available and sampling only be performed if the current position is greater than the value in the previous depth texture.

Two extra texture reads for each render pass, as well as two comparisons for each sampling along the ray, have to be performed inside the fragment program with this approach, and the volume still has to be rendered once for every layer in the scene. However, due to the early z rejection, the number of fragments processed in the main shader passes is not higher than when rendering the volume only once. In fact, if parts of the volume is obscured by geometry, the number of fragments will be smaller. In the intermediate pass the number of fragments increases linearly with the number of passes, however.

2.3 Depth peeling and near neighbor ray casting

The near neighbor approach to GPU-based ray casting is very well suited to combining with depth peeling for rendering embedded geometry. Since this method allows for also skipping fragments in the intermediate pass, the efficiency in terms of processed fragments can be expected to be almost as high as for normal ray casting where the volume is rendered only once.

The basic approach to integrating depth peeling is nearly identical to the early z method described in the previous section. The difference lies in the fragment program for the intermediate pass; instead of merely disabling rendering of the next main pass if the current position is less than the previous depth texture, rendering is completely disabled until the pass where rendering is to begin. Since rendering is also disabled after the depth texture boundary has been reached, the number of processed fragments does not increase significantly even when the volume is rendered many times.

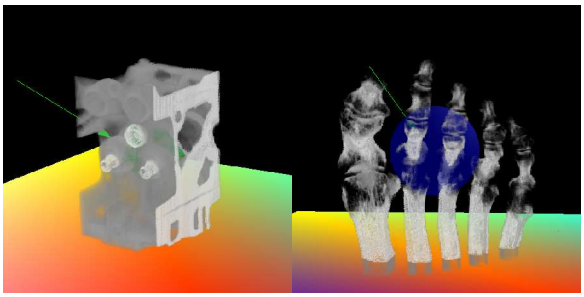

Figure 3: *Left (`Engine`): engine block with embedded geometry (in green) and an arrow pointing inside the volume. Right (`Foot`): CT scan of a human foot with a transparent blue sphere embedded in the volume and a green arrow behind the sphere. Both are of size* 256^3

Since the depth buffer is split in two, with one half representing linear distance within the volume bounding box, it is necessary to convert the depth values in the depth texture to a position within the bounding box which then determines in which pass rendering will resume. For this purpose a one-dimensional texture can be used, which maps a depth value to a position within the volume.

3 Simulating arbitrary clip geometry with ray casting

The normal OpenGL clipping planes cannot be used with the ray casting technique, as the proxy geometry, the volume bounding box, is rendered in the same location for all passes. Clipping planes, or in fact, arbitrary shapes, can easily be simulated in the fragment program, however.

The clip geometry is rendered in a separate pass with depth writes enabled and frame buffer writes disabled. After rendering, the depth buffer is copied to a depth texture which is then made available in the fragment program. This depth texture is sampled during rendering of the volume, and if the current position is less than the clip value from the depth texture the volume sample at that position is not blended in. Rendering only starts once the clip geometry has been reached and the effect archived is that of the volume being "cut" at the surface of the clip geometry.

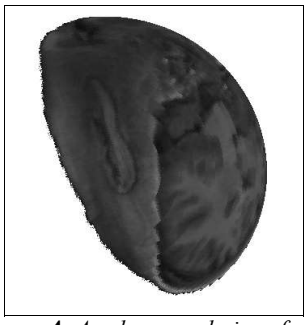

Figure 4: *A volume rendering of a scan of a human head, clipped against a sphere*

If this approach to clip geometry is used in conjunction with embedded geometry no extra test is needed inside the fragment program; the depth values of the clip geometry can instead be rendered into the depth buffer after the other geometry in the scene has been rendered. The depth test is reversed when the clip geometry is being drawn so that other geometry is effectively masked.

When using this technique it is important to ensure that the clip geometry actually covers the volume bounding box, otherwise the parts of the volume not covered will not be correctly clipped.

4 Results

Depth peeling in conjunction with GPU-based ray casting has been implemented. The `Foot` volume seen to the right of Figure 3 has been used to obtain the fragment counts; three depth layers were used, and tables 1 and 2 give the counts for the main and intermediate passes, respectively. The rendering was directed to a 500^2 viewport.

	Layer 0	***Layer 1***	***Layer 2***	***Sum***
Early z				394,987
with geometry	363,046	22,960	536	386,542
Near neighbor				336,413

	Layer 0	*Layer 1*	*Layer 2*	*Sum*
with geometry	281,973	27,569	16	309,558

Table 1: *Fragment counts for the main pass*

	Layer 0	*Layer 1*	*Layer 2*	*Sum*
Early z				9,891,936
with geometry	9,891,936	9,891,936	9,891,936	29,675,808
Near neighbor				2,088,150
with geometry	1,830,186	251,942	255,178	2,337,306

Table 2: *Fragment counts for the intermediate pass*

The reason for the slightly lower fragment count in the main pass is that some rays are terminated earlier due to the opacity falling below the threshold value. In fact, the number of fragments will always be equal or lower to the fragment count without embedded geometry.

In the intermediate pass the near neighbor method shows its superiority clearly, as fragments can be skipped for this pass as well, unlike the early z approach. The reason for the slightly higher fragment count with embedded geometry is that all fragments must be processed at least once per layer, to set the initial value to skip.

5 Discussion

Depth peeling is an efficient way to render transparent polygonal geometry, and, as has been demonstrated by this paper, it can be successfully integrated with volume rendering. With the early z approach, the number of processed fragments in the intermediate pass increases linearly with the number of depth layers in the scene; the near neighbor approach does not have this drawback. The early z method can still be useful, however, as, especially when taking into account complex calculations, such as lighting and shadowing, the main pass is by far the most expensive part of the volume rendering pipeline.

While the number of layers in a scene is often not easy to determine, the effects of adding more layers quickly diminish as the opacity of pixels is saturated. For this reason, using only 3 or 4 layers is usually sufficient to obtain good results.

Also, in most cases, it is not necessary to render the entire scene in the depth peeling step; only transparent geometry needs to be included. After the depth peeling technique has been applied to all transparent objects, all opaque geometry in the scene is rendered normally and the textures from depth peeling, along with any volume, are blended in afterward.

Among the possible uses of this technique, medical imaging is especially worth mentioning. Objects, such as a virtual scalpel of a surgeon practicing for an operation on an MRI scan of a patient, or annotations describing specific features of a similar scan, are often rendered inside of a volume. If these objects are opaque they might obscure important features of the volume and it is preferable to render them partly transparent to prevent this from happening.

Another use is for virtual reality, for example used by the games industry, where the geometry can be an avatar or a head-mounted display.

6 Conclusions

In this paper, we presented a novel method for enabling the rendering of transparent polygonal geometry embedded in a volume data set. The method combines "depth peeling", a known technique for order-independent transparency in polygonal rendering, with GPU-based volume rendering. By the use of early-z culling or the near-neighbor approach, the number of fragments processed in the main pass, where the actual sampling of the 3D texture is performed, is at most equal to rendering without embedded geometry. In fact, it is often less if the geometry obscures part of the volume. Additionally, for the near neighbor approach, the number of fragments in the intermediate pass does not increase significantly either.

In addition, we show how arbitrary clip geometry can be simulated with an approach very similar to depth peeling. In fact, if combined with rendering of embedded geometry, the processing of the clip geometry can be performed with no extra overhead.

While the renderer has been implemented using the nVidia GeForce 6800 graphics card, the early z method is applicable also to the previous generation of graphics cards (e.g. the ATI 9700 or nVidia GeForce FX series). However, the near neighbor approach requires the `GL_EXT_depth_bounds_test` OpenGL extension, which is currently only available on the third and forth generation graphics accelerators from nVidia.

7 Future work

In the current implementation, both the fragment program for the main and intermediate pass compares the value from the depth texture to the current depth; the reason for also performing this comparison in the main pass is to stop sampling at the correct point along the ray in the case the layer intersects the ray segment. Since this comparison entails some overhead it would be preferable if it could be avoided.

One method of achieving this is to terminate a ray when only a partial ray segment would be traversed in the next main pass, but write a special value in the depth buffer, slightly less than the value otherwise written for terminating a ray. After all passes have been rendered the depth test is reversed as to only include the fragments with the special value. The main pass is then performed to render the final partial segment of each ray, and, since Cg provides instructions for extracting the integer and fractional parts of a number, the starting position can be calculated from the depth texture. Since an additional pass is required it is unclear if this would provide any significant speed-up, however.

References

[Eve01] C. Everitt. 2001. Interactive "Order-Independent Transparency". White paper, NVidia, 2001.

[Fer04] nVidia Corporation; edited by R. Fernando. "GPU Gems". 2004. ISBN 0-321-22832-4.

[FK03] R. Fernando and M. J. Kilgard. "The Cg

Tutorial". 2003. ISBN 0-321-19496-9.

[KK99] KEVIN KREEGER AND ARIE KAUFMAN. "Mixing Translucent Polygons with Volumes". In *IEEE Visualization 99*.

[KW03] J. KRÜGER AND R. WESTERMANN. "Acceleration Techniques for GPU-based Volume Rendering". In *Proceedings of the 14th IEEE Visualization Conference*, p. 38, 2003.

[LCN98] B. LICHTENBELT, R. CRANE AND S. NAQVI. "Introduction To Volume rendering". 1998. ISBN 0-13-861683-3.

[SWND04] D. SHREINER, M. WOO, J. NEIDER AND T. DAVIS. "OPENGL PROGRAMMING GUIDE, FOURTH EDITION". 2004. ISBN 0-321-17348-1.

Chapter 7

Results

At the start of this project I was not familiar with the details of either volume rendering or GPU programming, however, now, at its conclusion, I have gained significant insight into the details of both.

In software-based volume rendering the most natural approach is that of ray casting, which can finally be implemented on ubiquitous 3D accelerator cards available in recent years. A first attempt at such an implementation was already publicized before the start of this project [1], thus I used this as a starting point and looked for ways of improving it.

Two major results have emerged from this project: the integration of the near neighbor skipping optimization technique for GPU-based ray casting, as well as a method for using the "depth peeling" technique to correctly render transparent geometry embedded in the volume.

In addition I created a volume renderer which does the lighting and gradient calculations on the GPU, however, the relatively limited time available for a project of this type did not allow for a more thorough investigation into these topics.

7.1 The near neighbor skipping optimization technique

This is the topic of Chapter 5; I here summarize the results.

Several optimization techniques are commonly used in software-based volume rendering, among them early ray termination, empty-space skipping, and the more advanced near neighbor skipping, all of which I have implemented on the GPU. As many typical volume data sets contain large regions of empty space, or voxels rendered completely transparent by the transfer function in use, significant savings can be realized by utilizing methods for detecting and effectively skipping over these regions. The near neighbor approach stores, for each voxel in the volume, the distance to the nearest visible voxel. Thus, after sampling a voxel, the distance to the nearest voxel is known and it is safe to skip ahead this distance along the ray before sampling the next voxel.

While it is straight-forward to implement this technique in software once all the distances are computed, it is significantly more involved to achieve any speed-up when implementing it on a GPU due to its parallel architecture. The most common way of avoiding the execution of a fragment program for a given fragment is by making it fail the depth

test. However, fragments programs which alter the depth value cannot normally be skipped, but by making use of an OpenGL extension, the `GL_EXT_depth_bounds_test`, even fragment programs writing a depth value can be safely skipped, as this extension is not concerned with the depth value of incoming fragments. By making use of this extension, it is possible to control exactly which fragments are processed at a given time, which is exactly what is needed for the implementation of the near neighbor optimization.

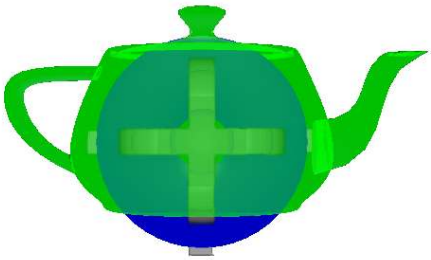

Figure 6: The scene used to obtain the fragment counts

By making use of an optimal near neighbor data structure, in which the distance to the nearest voxel is given only in one specific direction, the viewing direction, an order of magnitude reduction in the amount of processed fragments can be realized. Table 1 lists the number of processed fragments for the optimal near neighbor data structure as well as an omnidirectional and a directional for each quadrant. The numbers for the early z method are included for comparison, and the number of frames per second is given as well.

	Main pass	***Intermediate pass***	***FPS***
Early z	89,603	489,268	138
Optimal near neighbor	6,691	51,992	266
Omnidirectional near neighbor	69,330	315,061	132
Directional near neighbor	60,162	296,217	133

Table 1: Total number of fragments processed in the main and intermediate passes for three types of near neighbor data structures, as well as the early z method for comparison. The number of frames per second is given in the last column.

As we can see, especially the optimal near neighbor approach allows us

to avoid the processing of a high number of fragments in the main pass, which, when gradient and lighting calculations are taking into account, is much more expensive than in the intermediate pass.

The reason for the low frame rate achieved by the near neighbor approach is a lower fragment rejection rate when using the `GL_EXT_depth_bounds_test`, compared to the early z test. It should be noted that with a more complex fragment program in the main pass the near neighbor approach can be expected to overtake the early z method.

7.2 Embedded transparent geometry

This is the topic of Chapter 6; I here summarize the results.

Even with only polygonal geometry, transparency it problematic to render correctly. In principal it is easy: simply render all the geometry in back-to-front order and blend the color values into the frame buffer. Geometry is most often rendered object-by-object, however, with little consideration for depth order, and the geometry thus has to be sorted before rendering, which adds significant overhead. If geometric primitives overlap, as the two triangles in Figure 7 do, it is even necessary to split the triangles to achieve correct transparency with the sorting method. Clearly this can be inefficient for larger numbers of triangles, and this is especially true for volume rendering, as many triangles will intersect the proxy geometry used.

A better approach, depth peeling, is described in [6]. Now the scene is rendered multiple times, each time removing all visible pixels in the

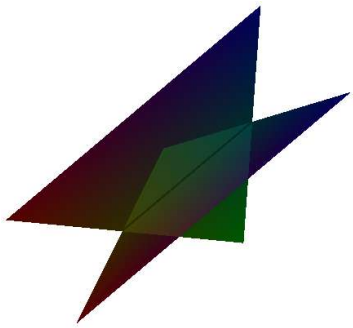

Figure 7: Two intersecting triangles rendered with correct transparency using depth peeling.

previous pass, and the result of each pass stored in a texture. The stack

of textures now depicts each layer of the scene, and when these textures are blended together in back-to-front order, the geometry will have correct transparency.

To integrate this with volume rendering the depth peeling is first applied to the scene excluding the volume, which is then rendered at the end. The volume has to be rendered the same number of times as there are layers in the scene, and in each pass only the section of the volume falling between layers is actually rendered. By taking advantage of the early z test this can be done efficiently, as each visible voxel will still only be processed once in the main pass.

	Main pass	***Intermediate pass***
Early z	89,603	489,268
Early z w/embedded geometry	79,665	1,957,072
Optimal near neighbor	6,691	51,992
Optimal near neighbor w/geom	5,125	170,587

Table 2: Number of processed fragments for the main and intermediate passes for early z and optimal near neighbor with and without embedded geometry

When rendering with embedded geometry the fragment counts for the main pass are actually slightly less than without; this is due to some fragments of the volume being obscured by geometry.

7.3 Illumination in volume rendering

While time did not allow for a closer investigation into this area of volume rendering, I did, as part of familiarizing myself with GPU programming and volume rendering, implemented a volume renderer which applies Phong shading to the rendered voxels.

As voxels do not have normals a vector quantity called the *gradient* is used instead. The gradient is interpolated from a voxel's neighbors and represents change in voxel value. It points in the direction of the greatest change and its length is determined by how quickly the value changes.

While the gradient can be calculated as part of rendering this is an expensive operation, especially if more than a simple linear interpolation is required, and since it is a static value it is usually pre-calculated and

supplied as a texture in addition to the volume data itself. I implemented a renderer capable of both modes of operation.

Figure 8: Volume rendered with Phong shading. A red light, with a strong specular component, is shining from the left, while a blue light is positioned to the right.

Figure 8 shows a volume rendering with Phong shading applied to each voxel. A red light is positioned up to the left and has a strong specular component, as can clearly be seen, while a blue light is located down to the right.

Chapter 8

Discussion

While volume rendering does not have the mass-marked appeal of for example computer games, it is an indispensable tool for the industries which make use of it. It does not mean, however, that the requirements to rendering performance and accuracy are any less than other fields in 3D graphics, in fact, the nature of volume rendering places very high demands on the rendering system just to handle the large volume of data. Any means of increasing rendering efficiency is therefore greatly welcomed.

GPU programming, on the other hand, is very much used in the mainstream and especially in games programming. In fact, the evolution of GPUs and commodity graphics cards is driven in large part by the game marked, and the volume it generates is what allows companies to develop and sell such powerful 3D hardware at low prices.

Taking advantage of this development in 3D hardware, also for volume rendering, is a natural step to take, and, as this thesis has shown, the use of a GPU has many benefits.

While everything the GPU can do can also be done in software, in most cases it is only at a severe disadvantage in speed. The GPU is especially tailored to providing interactive renderings, and this can mean the difference between a useful technology and one which cannot be used effectively.

While the methods proposed here do require a fairly recent graphics card, this should not be a deterrent to their use, as even modern 3D cards have a very low cost. This is especially true when compared to specialized volume rendering hardware. With the current pace of development in GPU performance, it can also be expected that methods which make more use of the GPU will also increase in performance faster then other methods, and will thus only increase their attractiveness.

Chapter 9

Conclusions

The research performed during this project has resulted in two major new techniques implemented on the GPU: ray casting with near neighbor skipping and correct rendering of transparent geometry embedded in a volume.

Both techniques have immediate usefulness in volume rendering and can readily be implemented in existing volume rendering systems. Systems in Motion, for whom this project has been realized, ships a volume rendering system called Voleon as part of their product portfolio. Both proposed techniques are scheduled for integration with this system, and will then be of benefit to their clients.

Chapter 10

Future work

For both techniques presented in this thesis possible future work is described in their respective chapters, however, perhaps the most important is the integration of these techniques in a full-featured volume rendering system, such as Voleon, and certifying that they interact with other features of the system in a nice way.

Another consideration in a real volume rendering system is that ray casting does not provide a speed-up for all volumes; in fact, for dense and highly transparent volumes the extra overhead may result in lower performance. A method for automatically detecting if this is the case and switching to a traditional slice-based renderer would free the user from having to make this determination on a per-volume basis.

Finally, there are several other fields in volume rendering, such as transfer function evaluation, lighting and shadow calculations and handling volumes larger than the available texture memory on the graphics card, which could be advanced by investigating which possibilities exist for taking advantage of the GPU.

Chapter 11

Bibliography

[1] J. Krüger and R. Westermann. 2003. Acceleration Techniques for GPU-based Volume Rendering.

[2] T. J. Purcell. 2004. Ray Tracing on a Stream Processor.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. SIGGRAPH 2004.

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman and A. Y. Wu. 1998. An optimal algorithm for approximate nearest neighbor searching. Journal of the ACM, 45 (1998), 891-923.

[5] Kevin Kreeger and Arie Kaufman. 1999. Mixing Translucent Polygons with Volumes. In *IEEE Visualization 99*.

[6] C. Everitt. 2001. Interactive Order-Independent Transparency.

[7] M. Harris. Dynamic branching on the nVidia GeForce 6 series. *http://www.gpgpu.org/forums/viewtopic.php?p=1694#1694*

[8] B. Lichtenbelt, R. Crane and S. Naqvi. 1998. Introduction To Volume rendering. ISBN 0-13-861683-3.

[9] R. Fernando and M. J. Kilgard. 2003. The Cg Tutorial. ISBN 0-321-19496-9.

[10] nVidia Corporation; edited by R. Fernando. 2004. GPU Gems. ISBN 0-321-22832-4.

[11] D. Shreiner, M. Woo, J. Neider and T. Davis. 2004. OpenGL Programming Guide, Fourth Edition. ISBN 0-321-17348-1.

[12] Bui-Tuong Phong. 1975. Phong. Illumination for Computer Generated Pictures. *Communications of the ACM 18(6) June 1975, p. 311-317.)*

[13] T. Porter and T. Duff. 1984. Compositing Digital Images. *Computer Graphics Volume 18, Number 3 July 1984 pp 253-259*