ACCELERATION OF DIRECT VOLUME RENDERING WITH TEXTURE SLABS ON PROGRAMMABLE GRAPHICS HARDWARE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HACER YALIM

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2005

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Veysi İşler
Co-Supervisor

Assoc. Prof. Dr. Ahmet Coşar
Supervisor

Examining Committee Members

Prof. Dr. Bülent Özgüç (BİLKENT UNV.,CENG) _____

Assoc. Prof. Dr. Ahmet Coşar (METU, CENG) _____

Prof. Dr. Adnan Yazıcı (METU, CENG) _____

Assoc. Prof. Dr. Ferda Nur Alparslan (METU, CENG) _____

Assoc. Prof. Dr. İsmail Hakkı Toroslu (METU, CENG) _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name :

Signature :

ABSTRACT

ACCELERATION OF DIRECT VOLUME RENDERING WITH TEXTURE SLABS ON PROGRAMMABLE GRAPHICS HARDWARE

Yalım, Hacer
MSc., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Ahmet Coşar
Co-Supervisor: Assoc. Prof. Dr. Veysi İşler

June 2005, 69 pages

This thesis proposes an efficient method to accelerate ray based volume rendering with texture slabs using programmable graphics hardware. In this method, empty space skipping and early ray termination are utilized without performing any preprocessing on CPU side. The acceleration structure is created on the fly by making use of depth buffer efficiently on Graphics Processing Unit (GPU) side. In the proposed method, texture slices are grouped together to form a texture slab. Rendering all the slabs from front to back viewing order in multiple rendering passes generates the resulting volume image. Slab silhouette maps (SSM) are created to identify and skip empty spaces along the ray direction at pixel level. These maps are created from the alpha component of the slab and stored in the depth buffer. In addition to the empty region information, SSM also contains information

about the terminated rays. The method relies on hardware z-occlusion culling that is realized by means of SSMs to accelerate ray traversals. The cost of generating this acceleration data structure is very small compared to the total rendering time.

Keywords: Direct volume rendering, graphics hardware, empty space skipping

ÖZ

HACİM DATA GRAFİK SUNUMUNU DOKU DİLİMLERİ KULLANARAK PROGRAMLANABİLİR GRAFİK İŞLEMCİDE HIZLANDIRMAK

Yalım, Hacer
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ahmet Coşar
Ortak Tez Yöneticisi: Doç. Dr. Veysi İşler

Haziran 2005, 69 sayfa

Bu tez çalışması ışın tabanlı hacim grafik sunum metodlarını doku dilimleri kullanarak programlanabilir grafik işlemcilerde hızlandırmaya yönelik yeni bir yaklaşım önermektedir. Yöntem, boş alanları atlama ve erken ışın sonlandırma tekniklerini ana işlemci tarafında ön işlemeye gerek duymadan uygular. Hızlandırma yapıları grafik işlemci tarafında derinlik belleğini verimli bir şekilde kullanarak oluşturulurlar. Önerilen çalışmada, ince doku dilimleri bir araya getirilerek kalın doku dilimleri oluşturulur. Bu kalın dilimlerin önden arkaya çizdirilmesi ile hacim görüntüsü elde edilir. Dilimlerden silüet haritaları oluşturulur ve bu haritalar piksel düzeyinde boş alan atlama ve erken ışın sonlardırma işleminde kullanılırlar. Dilim silüet haritaları dilimlerin geçirgenlik özelliğinden faydalanılarak oluşturulurlar ve derinlik ara belleğinde tutulurlar. Silüet haritaları yalnızca hacimdeki boş alanları

değil, aynı zamanda erken biten ışınların bilgisini saklamakta da kullanılırlar. Yöntem, ışın izleme sürelerini hızlandırmak için donanım destekli z-gizleme metodunu silüet haritalarından faydalanarak kullanır. Hızlandırma yapılarını oluşturma maliyeti toplam görüntü oluşturma zamanına oranla oldukça düşüktür.

Anahtar Kelimeler: Hacim görüntüleme, grafik işlemci programlama, boşluk atlama yöntemi ile hızlandırma

To My Family

ACKNOWLEDGMENTS

I wish to express my deepest gratitude to my supervisor co-supervisor Assoc. Prof. Dr. Veysi İşler and Assoc. Prof. Dr. Ahmet Coşar for their guidance, advice, criticism, encouragements and insight throughout the research.

I would also like to thank Şükrü Alphan Es for his suggestions and valuable comments during this thesis study.

This study was partially supported by the Tübitak-Bilten. I would like to thank to Dr. Uğur Murat Leloğlu and Işıl Gürdamar for their support. Moreover, I would like to express my deep appreciation to Devrim Tipi Urhan for her support during the thesis study.

Finally, I would like to thank to my husband for everything.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ vi
ACKNOWLEDGMENTS ix
TABLE OF CONTENTS x
LIST OF TABLES xii
LIST OF FIGURES xiii
CHAPTER 1
1 INTRODUCTION 1
1.1 Transfer functions 2
1.1.1 Iso-value Contour Surfaces 3
1.1.2 Region Boundary Surfaces 4
1.2 Physical Background 5
1.2.1 Emission-absorption model 5
1.2.1.1 Absorption only 6
1.2.1.2 Emission only 6
1.3 Direct Volume Rendering Algorithms 8
1.4 Literature Survey 10
1.4 Objective 13
1.5 Scope 13
1.6 Outline 14
2 GRAPHICS HARDWARE 15
2.1 Evolution of Graphics Hardware 15
2.2 Programmable Vertex and Fragment Processors 23
2.2.1 Programmable Vertex Processor 23
2.2.2 Programmable Fragment Processor 25

2.3 Programming Interfaces 27
2.3.1 Shading Languages 27
3 TEXTURE BASED VOLUME RENDERING 30
3.1 2D Texture Based VR 30
3.1.1 Algorithm 31
3.1.2 Advantages and Disadvantages 34
3.2 3D Texture Based VR 34
3.2.1 Algorithm 35
3.2.2 Texture Coordinate Generation 36
3.2.3 Advantages and Disadvantages 37
4 ACCELERATED DIRECT VOLUME RENDERING WITH TEXTURE SLABS 40
4.1 Algorithm 40
4.1.1 Empty Space Skipping (ESS) 42
4.1.2 Early Ray Termination (ERT) 43
4.2 Implementation 44
4.3 Kernel Operations 48
4.3.1 Slab Silhouette Map (SSM) Kernel 48
4.3.2 Ray Traverser Kernel 52
4.3.3 ERT Kernel 55
5 DISCUSSION AND RESULTS 57
6 CONCLUSION AND FUTURE WORKS 63
6.1 Future Works 65
REFERENCES 66

LIST OF TABLES

Table 1: Performance Results of the Experiments 59
Table 2: Total Kernel Execution Times (in ms) 62

LIST OF FIGURES

Figure 2.1: Graphics Hardware Pipeline..... 17
Figure 2.2: First Generation GPUs (1995) 18
Figure 2.3: Raster Operations Unit and per-fragment tests 19
Figure 2.4: Fixed Function Pipeline (T&L Unit on GPU side) 20
Figure 2.5: T&L Unit on GPU side 21
Figure 2.6: Third Generation Programmable Vertex Processor 22
Figure 2.7: Fourth Generation Programmable Vertex Processor 23
Figure 2.8: Vertex Processor Flow Chart 24
Figure 2.9: Fragment Processor Flow Chart..... 26
Figure 2.10: Software Architecture with Shading Languages..... 28
Figure 3.1: Object-Space Axis-Aligned Data Sampling..... 31
Figure 3.2: Texture Set for Three Object Space Axis..... 32
Figure 3.3: Sample Points for Different Slice Sets..... 33
Figure 3.4: Image-Space Axis Aligned Texture Slices..... 35
Figure 3.5: Bounding Cube..... 37
Figure 3.6: Rearrangement of Texture Slices According to View Direction 38
Figure 4.1: Viewport Aligned Texture Slices and Texture Slabs..... 41
Figure 4.2: Summary of the Algorithm 44
Figure 4.3: Flow of Kernels and Their Effect to Pbuffer. 47
Figure 4.4: Pseudo-code of the Algorithm 47
Figure 4.5: Depth Generation Function..... 49
Figure 4.6: Early Depth Test Initialization..... 50
Figure 4.7: Cg Source Code of CSSM Kernel..... 51
Figure 4.8: Cg Source Code of Ray Traverser Kernel..... 53

Figure 4.9: Traversal Through the Full Regions.. 54
Figure 4.10: Initialization of OpenGL States for Ray Traverser Kernel 55
Figure 4.11: Cg Source Code of ERT Kernel 56
Figure 5.1: Rendering Results of the Datasets..... 57
Figure 5.2: First 10 SSMs of the Engine Model 60
Figure 5.3: First 10 SSMs of the Teapot Model 61
Figure 5.4: First 10 SSMs of the Skull Model 61

CHAPTER 1

INTRODUCTION

Volume rendering is used, in this thesis, as a basis for developing techniques that allow the visualization of three-dimensional scalar data. Volume rendering is utilized in many application areas such as computational fluid dynamics, scientific modeling and visualization, and medical imaging.

Continuous three-dimensional volume data is discretized with different sampling grid structures. In medical imaging, these data sets are generated with tomographic measurements available from Computer Tomography (CT) scanners, Positron Emission Tomography (PET) scanners and Magnetic Resonance Images (MRI) in the form of uniform *rectilinear* grids. On the other hand, there are also unstructured grid structures based on tetrahedron, triangular prism, or square pyramid. These structures are mostly used in finite element simulations. In the scope of this thesis, volume data sets arising from CT and MRI measurements are considered and hence we restrict this work to uniform rectilinear grids.

Each volume element in the uniform grid is called a *voxel* and every voxel holds a density (scalar) value. The aim of a volume rendering algorithm is to determine the visibility of each voxel and visualize it by considering its density.

There are two main approaches to volume rendering [23]. The first approach is based on extracting conventional computer graphics primitives from the volume grid. The primitives can be surfaces, curves and points, which can be displayed using polygon based graphics hardware [1]. These methods are called in general as *Indirect Volume Rendering (IVR)* or *Surface Rendering* methods. Different approaches in IVR use different primitives with different scales. The assumption

here is that, a set of iso-surfaces exists in the volumetric data and that extracted polygon mesh can model every surface, including very small surfaces, as the true object structures with acceptable quality. However, IVR methods do not provide a general solution. Visualization of fuzzy or cloud-like semi-transparent data is not appropriate with this method.

Second group of volume rendering approaches render volume elements directly. That is, no intermediate conversion is required to extract surface information from the volume data. These methods are called *Direct Volume Rendering (DVR)* methods. In these methods, all volumetric voxels contribute to the final image. In contrast to IVR methods, DVR methods are appropriate for displaying weak or fuzzy surfaces as well as iso-surfaces.

In the following sections, the basics of volume rendering methods are described briefly. Next, an overview of direct volume rendering algorithms is given and a summary of the literature on the acceleration techniques in the context of volume rendering is presented.

1.1 Transfer functions

A volume can be modeled by a data grid, where each grid vertex contains a particle with certain density. Optical properties of these particles, like color and opacity, are required to be able to render the content of the volume. For this purpose, transfer functions are defined to map scalar density values to the optical parameters. For instance, mapping different tissue types in medical images to different color and alpha values is critical for true perception of volume content. This is called *data classification*. Data classification enables viewer to focus on the areas where valuable information is located. For example, mapping certain scalar values to high alpha (opacity) values and mapping the rest to lower values enables the visualization of certain *iso-surfaces*. The interior parts, as well as iso-surfaces, can also be visualized as clouds with varying density and color mapping.

In our study, two different transfer functions are implemented and utilized; iso-value contour surfaces and region boundary surfaces. These are the two different classification methods introduced by Marc Levoy [5]. Since main goal of this thesis is achieving real-time rendering, complex transfer functions are out of the scope of this thesis.

1.1.1 Iso-value Contour Surfaces

The basic principle in determining the iso-value surface is assigning an opacity value to all the voxels having the same density value. However, with this simple model, the generated image cannot contain multiple concentric semi-transparent objects. Hence, the assignment of opacity values is made utilizing the approximate surface gradients. The algorithm first assigns α_v value to voxels with selected density f_v. In addition to them, voxels with density values close to α_v are assigned opacities close to f_v. The transfer function is stated in (1).

$$\alpha(x_i) = \alpha_v \begin{cases} 1 & \text{if } |\nabla f(x_i)| = 0 \text{ and } f(x_i) = f_v \\ 1 - \frac{1}{r}\left|\frac{f_v - f(x_i)}{\nabla f(x_i)}\right| & \text{if } |\nabla f(x_i)| > 0 \text{ and } f(x_i) - r|\nabla f(x_i)| \le f_v \le f(x_i) + r|\nabla f(x_i)| \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

According to this function, in the neighborhood of selected density value f_v, opacity is decreased inversely proportional to the magnitude of local gradient vector. The neighborhood is represented with r, which is the pre-specified thickness value of the transition region. This thickness is kept constant throughout the volume.

The approximate surface gradient for voxel v in grid position (x_i,y_j,z_k) is calculated using the operator as shown in (2).

$$\nabla f(v) = \Big[\ \frac{1}{2}\big[f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)\big], \\ \frac{1}{2}\big[f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k)\big], \\ \frac{1}{2}\big[f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})\big] \ \Big] \tag{2}$$

For the classification of more than one iso-surface in a single image, the classification of each iso-surface is performed separately and then combined with the formulation given in (3).

$$\alpha_{tot}(x_i) = 1 - \prod_{n=1}^{N} (1 - \alpha_n(x_i)) \tag{3}$$

Where, N different density values are combined, each with different opacity values and different/same transition regions. α_{tot} is the resultant opacity value of the current voxel.

1.1.2 Region Boundary Surfaces

Volume data obtained from CT scan of human body contains predictable density values for different biological tissue types. Region boundary surface detection is based primarily on the assumption that density values of a certain tissue type falls into a small neighborhood of a certain density value. With this assumption, different tissue types are assigned to different opacity values and all can be visualized in one volume image. It is well suited to use region boundary surface classification for medical volume data with different tissue types rather than iso-surface contour classification.

The method proposed in Levoy's work has a constraint that one tissue type can touch at most two other tissue types. If this criterion is violated, the method can not classify some of the voxels unambiguously. In this respect this method is too restricted. However, as stated earlier, classification is not the main part of this thesis

study and we implemented this basic method considering its restrictions. The transfer function is given below.

$$\alpha(x_i) = |\nabla f(x_i)| \begin{cases} \alpha_{v_{n+1}} \left[\frac{f(x_i) - f(v_n)}{f(v_{n+1}) - f(v_n)} \right] + \alpha_{v_n} \left[\frac{f(v_{n+1}) - f(x_i)}{f(v_{n+1}) - f(v_n)} \right] & \text{if } f(v_n) \le f(x_i) \le f(v_{n+1}) \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

For $n = 1,2,..N$, and selected density values have the property that $f(v_n) < f(v_{n+1})$ and tissue of v_n touches only to the tissues v_{n-1} and v_{n+1}. Moreover, the surface gradient calculation is performed using equation (2).

1.2 Physical Background

Light transport theory forms the basis for all of the physically based rendering methods [6, 17]. Supposing that a volume is composed of small particles with certain densities, the light passing through a volume grid is affected by optical properties of these particles with absorption, scattering or emission. Scattering of light is a complex procedure that is usually neglected by volume rendering approaches. Hence, in this study, an *emission-absorption* model is selected to model the behavior of light. Light passing through a participating medium is affected from optical properties of individual particles and the total affect of each particle is modeled with a differential equation to express the light flow in the medium. For a continuous medium, absorption, emission and scattering occur at every infinitesimally small segment of the ray.

1.2.1 Emission-absorption Model

Emission absorption model could be understood when emission and absorption are stated individually first.

1.2.1.1 Absorption Model

The particles in the participating medium absorb the light that they intercept. This is modeled as shown in equation (5).

$$I(\mathrm{s},\mathrm{y}) = I_0 * \exp\left(-\int_0^s \tau(t)dt \right), \tag{5}$$

where $I(\mathrm{s},\mathrm{y})$ is the intensity of light at distance s that is received at location y on the image plane, I_0 is the initial intensity of the light. $\tau(t)$ is the extinction coefficient which defines the rate the light is occluded (the opacity value of the particles). In the formula, second term (the exponential function) gives the transparency of the medium between 0 and s.

1.2.1.2 Emission Model

In contrast to absorption of light, a medium adds light to the ray by reflection of external illumination. This is modeled as (6).

$$I(s,y) = I_0 + \int_0^s g(t)dt\,, \tag{6}$$

where, $g(t)$ is emission term.

The emission absorption model is defined combining to 5 and 6 as below:

$$I(S,y) = I0 * \exp\left(-\int_0^S \tau(t)dt \right) + \int_0^S \left(g(t) * \exp\left(-\int_t^S \tau(x)dx \right) \right) dt\,, \tag{7}$$

where, S is the length of the ray segment and the second term is the integral that calculates the contribution of the source term $g(t)$ at each position t by multiplying it with the transparency of the ray between t and eye (S). This model is referred to as

Volume Rendering Integral (VRI) that computes the amount of light that is received at location y on the image plane.

To solve this equation numerically, a discretization is performed along the ray to approximate the analytical integration. Integration range is partitioned into n equal intervals (hence each segment element has a length of S/n), the formula becomes as in (8).

$$\begin{aligned} I(S,y) &\approx I_0 \prod_{i=1}^{n} t_i + \sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} t_j \\ &= g_n + t_n(g_{n-1} + t_{n-1}(g_{n-2} + \ldots(g_1 + t_1 I_0)\ldots))) \end{aligned} \tag{8}$$

This iterative solution to emission-absorption model is the fundamental to almost all of the direct volume rendering methods. This expression is referred to as discretized VRI (DVRI).

A particle can absorb, reflect or emit the incoming light according to its specular, diffuse and emission material properties. The model expressed in equation (8) can be expressed for each light component with wavelength λ, as the amount of light coming from direction *r* that is received at location *y* on the image plane as:

$$I_\lambda(y,r) = \sum_{i=0}^{n} C_\lambda(s_i)\alpha(s_i) \cdot \prod_{j=0}^{i-1}(1-\alpha(s_j)) \tag{9}$$

where, opacity α = 1-transparency. $C_\lambda(s)$ is the light of wavelength λ reflected or emitted at location s in the direction r. VR algorithms calculate color and opacity values at discretized sample points s_i and composite them from front to back order according to (9).

VR algorithms can be distinguished by how they obtain color and opacities. In this respect, algorithms are classified as pre-shaded VR and post-shaded VR. The main point is the order of classification and shading when interpolating ray samples.

It is called *post-shaded*, if density value is interpolated first and then color and opacity components are determined. On the other hand, if color and opacity values are initially calculated for all the vertices and these values are interpolated at the sample points before classification, it is called *pre-shaded* algorithm. Post-shading gives more accurate results than pre-shading which tends to give blurry images. For pre-shaded algorithms, equation (9) is valid. To express post-shaded algorithms, interpolation function is included as:

$$I_\lambda(y,r) = \sum_{i=0}^{n} C_\lambda(f(s_i))\alpha(f(s_i)) \cdot \prod_{j=0}^{i-1}(1-\alpha(f(s_j))) \tag{10}$$

where, $f(x)$ gives interpolated density value at point x, $C(x)$ and $\alpha(x)$ transfer the density values to color and alpha components respectively. The images contain fine detail when the density values are interpolated first and then classified. In this thesis, post-shading is utilized in the calculation of DVRI.

1.3 Direct Volume Rendering (DVR) Algorithms

In contrast to Indirect Volume Rendering (IVR) techniques, which display only the surface primitives, Direct Volume Rendering (DVR) techniques display the contents of all voxels utilizing the model stated with equation (8).

In general, DVR algorithms can be classified in three main groups as image-order methods, object-order methods and fourier-space methods. *Image-order* methods calculate the final color for each pixel of the resulting image. Hence, starting points in these methods are pixels on the image plane. Ray casting [5] and share-warp [15] methods are in this category. On the other hand, *object order methods* calculate the contribution of each voxel to the resultant image. Starting points in this case are voxels (object cells). Splatting [9, 3], cell projection [8] and 3D texture based methods [16] are grouped in this category. *Fourier-space methods* generate the volume image working in frequency domain. In a preprocessing step

the volume data (3d) is transformed to frequency-domain. Then the projection image is created by extracting a slice image. Finally, this 2d projection is transformed back to spatial domain with inverse fourier transform [11].

In *Volume Ray Casting*, rays are cast from the observers' eye point through the volume data. For each ray a vector of sample colors and opacities are obtained by resampling the voxels at evenly spaced locations. The obtained values are composited using DVRI from front-to-back or back-to-front order to yield a single color and opacity value for that ray. Finally, the resultant color is projected on the viewing plane. This process is done for each image pixel.

In *Shear-Warp Factorization* methods an appropriate shear transform is used to efficiently access volume data along a slice. With an appropriate shear-transform volume data is transformed to object space. By this way, sampled slices are mapped to actual planes in volume data, which enables doing sampling more efficiently for any viewing direction. Then, the obtained image is warped to transform back to original viewing direction.

Splatting methods do rendering by first sorting the voxels from back to front order and then composite the projections of each cell into a resultant image. These projections are called *footprints (splats)* of cells. Different splatting algorithms use different representations of volumetric data with different splat sizes.

Texture Mapping Based methods, take advantage of hardware assisted 2D and 3D texture mapping utilities of graphics hardware. These methods represent the volume data as a stack of 2D textures or as a 3D texture. Rendering is performed using the hardware support of graphics units. These methods are fairly faster than the methods described above, however they have high memory requirements.

In the following section, works in the literature about acceleration methods for DVR is listed and hardware accelerated texture based algorithms are summarized.

1.4 Literature Survey

One of the challenges of DVR methods is their rendering speeds. Displaying the contents of every cell through a viewing direction is a costly process. Therefore, these methods suffer from the long-display times. Starting from the earlier days of volume rendering methods, many researchers have devoted their times on refining these methods. Many acceleration techniques have been developed to have real-time control over volume data.

The acceleration methods differ significantly in terms of the principal methodology they use and the kind of data structures they can display. All of these techniques depend on the classification of features in the data in a pre-processing step. Early acceleration methods used hierarchical data structures such as K-d trees [6] and octrees [5] to skip empty regions of volume data to reduce the number of samples needed to construct the final image. Afterwards, several works focused on hierarchical data structures [9], [10]. These data structures provide acceleration of rendering homogenous regions as well as empty regions. However, usage of complex data structures (such as octrees) has extra memory requirements. Considering these disadvantages, later techniques explored other forms of encoding schemes such as look-aside buffers, proximity clouds [17], and shell encoding [12] to skip empty spaces. These methods are more successful than hierarchical techniques. This is because of the encoding scheme they use. Information about empty regions is indexed with the same indices used for volume data.

Recently the number of volume rendering techniques that make use of hardware assisted texture mapping has increased. The idea of using 3D textures for rendering volumetric images of substantial resolution is first mentioned in *SGI Reality Engine* [13]. Cullip and Neumann [14] and Cabral's [16] work are among the first papers using 3D texture mapping hardware to accelerate volume rendering. These early approaches loaded the pre-calculated shading results into a 3D texture and used texture-mapping hardware for visualization. However, shading calculations have to be redone whenever the viewing parameters change. Van Gelder et al. [19]

proposed Voltx that utilizes hardware assisted 3D texture mapping with a light model. In their approach the 3D volume texture is reloaded whenever viewing direction changes. Later, Westerman and Ertl [20] introduced new approaches about accelerating volume rendering with advanced graphics hardware implementations through standard APIs like OpenGL. They mentioned using color matrix for shading and shaded volume on the fly. In their approach, there is no need to reload volume texture when viewing parameters change. Meibner et al. [22] extended their approaches giving support for semi-transparent volume rendering. They introduced multiple classification spaces using graphics hardware. Volume texture is stored only once and the rest of the calculations are performed on GPU part. Rezk-Salama [24] et al. used multi-texturing and multistage rasterization utilities of Nvidia GeForce graphics cards to improve both the performance and image quality of 2D texture based approaches. This work aimed to introduce a method to visualize volume data interactively on low cost consumer graphics cards which have no hardware support for trilinear interpolation. Engel et al. [25] proposed a method to decrease the number of texture slices without loosing rendering quality using multi textures and advanced per-pixel operations on programmable graphics hardware. They explained pre-integrated classification to sample continues scalar field without the requirement for increasing sample rate, hence improved the rendering performance. Pre-integration is completed in a preprocessing step and dependent textures are utilized to efficiently render the volume data.

In addition to these advances in volume visualization using 3D texture mapping hardware, new acceleration techniques directed their ways to apply classical acceleration techniques like empty space skipping and early-ray termination to 3D texture based methods exploiting the new generation programmable graphics hardware chips. New approaches designed their algorithms and data structures in order to take advantage of internal parallelism and efficient programmability of the dedicated graphics hardware utilities [27, 28, and 29]. The study of this thesis belongs to this category. Empty regions in volume data are the parts, which have zero opacity or an opacity value that are unimportant for

visualizing. Skipping those regions has no effect on the final image. In [29], a volume ray casting method is offered, which uses an octree hierarchy to encode empty regions. The information in this data structure is calculated in a pre-processing step on CPU and loaded into a 3D texture. GPU utilizes this information to skip empty spaces. Furthermore, early ray termination is performed on GPU by checking accumulated opacity value against a predetermined threshold in an intermediate pass and exploiting early z-test utility of ATI 9700 graphics cards. On the other hand, in [28], volume is partitioned into sub-volumes containing similar density properties using growing boxes [27]. The sub-volumes are rendered in visibility order and they are reorganized whenever the viewing direction changes. For this purpose an orthogonal BSP tree structure is constructed. Empty sub-volumes are skipped utilizing this structure. In addition to skipping empty regions, an orthogonal opacity map is created to skip occluded pixels on GPU. Occluded pixels are determined in sub-volume level by checking the projections of sub-volumes to the occlusion map.

In this thesis, a new acceleration method for volumetric ray casting algorithms on *Graphics Processing Units (GPU)* is explained. This algorithm creates and uses a special representation of volume regions for skipping empty spaces efficiently. To do this, the method exploits the programmability of the new generation graphics chips. The creation of this information is done in real-time and its burden on display times is very small compared to volume rendering times. Hardware assisted 2D and 3D textures are used extensively to transfer data between CPU and GPU. Using this method, rendering is performed at least two-times faster than the original volume ray casting method. Both [29] and [28] utilize particular data structures, octrees and BSP trees respectively, created on the CPU side to store empty space information for acceleration. The approach in this study differs from [28] and [29] in that; no explicit data structure on the CPU side is created to encode volume space information. Instead, the information is created on the fly on GPU side without doing any pre-processing.

1.4 Objective of the Thesis

The objective of the thesis is to achieve volume visualization using programmable graphics hardware and to accelerate this visualization again by using advanced features of GPU. Hence, the objective can be divided into two main parts.

The first issue is to work on the volume rendering methods and to select a method that generates good quality volume images. After determining this method, the next step is to implement it using classical software based methods.

The second issue is to analyze and design a new algorithm using advanced features of the GPU. Therefore, initial objective here is studying GPU programming and accumulating knowledge in this subject. The next objective is adapting the selected method to work efficiently on programmable GPU. Final objective is to create an acceleration structure on GPU and accomplish high quality visualization of volume data in real-time.

1.5 Scope of the Thesis

There are different approaches to volume visualization problem. The study has to consider both image quality and rendering time while selecting the method. In this respect, the scope of this thesis is limited to ray casting based direct volume rendering algorithms using texture mapping hardware, as ray casting methods provide high quality images.

Using programmable graphics units efficiently in volume visualization is among the main research topics of this study. For improving rendering times, the development of new acceleration structures on a programmable GPU is added to the scope. In addition to them, the study covers the data classification part of volume rendering. However, advanced classification techniques are beyond the scope of this study.

1.6 Outline

The outline of the thesis is as follows. In the second chapter, an overview of graphics hardware is provided. In the third chapter, the details of the texture based volume rendering techniques are given. In Chapter 4, the proposed acceleration method is explained in detail. In the subsequent chapter, a discussion about the features of proposed method takes place providing the sample test results. Finally, a conclusion is made and future works are stated.

CHAPTER 2

GRAPHICS HARDWARE

The basic concepts of the graphics hardware are outlined in the following sections. Since the proposed acceleration method relies on the advanced features of the programmable graphics units, this information serves as a reference for the GPU concepts mentioned in this thesis. Nowadays, programmability of graphics hardware exists in many general purpose consumer PCs.

The outline of this chapter is as follows. Firstly, evolution of graphics hardware is briefly explained. Following that, programmable vertex and fragment shaders are introduced. Finally, programming interfaces are mentioned and popular shading languages are stated.

2.1 Evolution of Graphics Hardware

The GPU on commodity graphics cards are evolving at incredible rates not only in the processing power but also in the flexibility and programmability since 2001. With the recent advances, GPU become a very fast general purpose stream processing hardware. Their performance is increasing faster than the ratio stated in Moore's law, especially in arithmetic power. It is because of the specialized nature of GPUs that makes it easier to utilize additional transistors for computation. There are many forces driving to this speedy improvement. The constant redoubling of the computer power with semiconductor industry is one of the fundamental forces. Another one is the tendency of people on simulating the 3d world in computer environment. Moreover, the incredible grow rate of the game and entertainment market results in more demand on faster GPUs.

According to the advances in graphics hardware, the evolution of GPUs is divided into four generations by industry observers [34]. From one generation to the next, the performance and the programmability of GPUs have increased.

Before the production of GPUs, companies like Silicon Graphics (SGI) and Evans & Sutherland had designed their special purpose graphics hardware which was very expensive. Many of today's important concepts have been introduced with these graphics chips. Hence, these works are considered to be the starting points of the evolution of new generation GPUs. In the following paragraphs, evolution of GPUs and the graphics pipeline is briefly summarized.

Hardware graphics pipeline is composed of the two fundamental stages at the top layer: *Geometry stage* and *Rasterization stage* (Figure 2.1). Each of these stages has a pipeline structure inside. In the application side, the scene is represented with many 3D triangles. The triangles that are sent to the graphics unit for visualization are first entered to the geometry stage of the pipeline. For each vertex of triangles, model-view projection transformation is performed to find the vertex's 2D screen positions. In addition to the position information, some of the vertex related attributes are calculated in this stage. This stage generates 2D triangles as the outputs. These triangles are sent to the rasterization stage.

In the rasterization stage, view frustum culling and clipping is performed and the visible parts of the triangles are rasterized. *Rasterization* is the task of determining the pixels covered by a geometric primitive. The result of this operation is a set of fragments and a set of pixel positions. *Fragment* is defined as a state that is required to update a particular pixel in the frame buffer. Vertex attributes like color, texture coordinates and normals are interpolated and assigned as fragments' attributes. Then shading is performed according to these parameters. Finally, a sequence of visibility tests are applied to the fragments and the frame buffer pixels are modified according to the results of these tests.

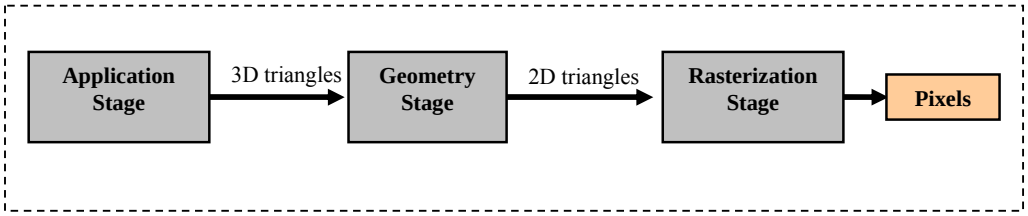

Figure 2.1: *Graphics Hardware Pipeline*

After this overall explanation of graphics pipeline, the evolution of the GPUs and corresponding pipeline stages can be understood easily.

Deering et al. designed GPU architecture with a pipeline of triangle processors and a pipeline of shader processors utilizing an inexpensive VLSI solution in 1988 [2]. Following that, GPUs are designed to include several triangle processors to use triangles in the geometry rasterization. Those early GPUs can perform the pre-transformed rasterization of triangles and have the ability to map one or two textures onto the geometries. The GPUs that are produced until 1998 are grouped as *the first generation GPUs* (see Figure 2.2). In this generation, pixel updates are started to be achieved on GPU side. On the other hand, vertex transformations are still performed by the application, which means that the load of the geometry stage is on CPU side. Also, the set of mathematical operations on GPU side is very limited. Examples of GPUs in this generation are NVidia TNT2 and ATI Rage. In 1998, the texture unit in Figure 2.2 was replaced with *multi-texture unit.*

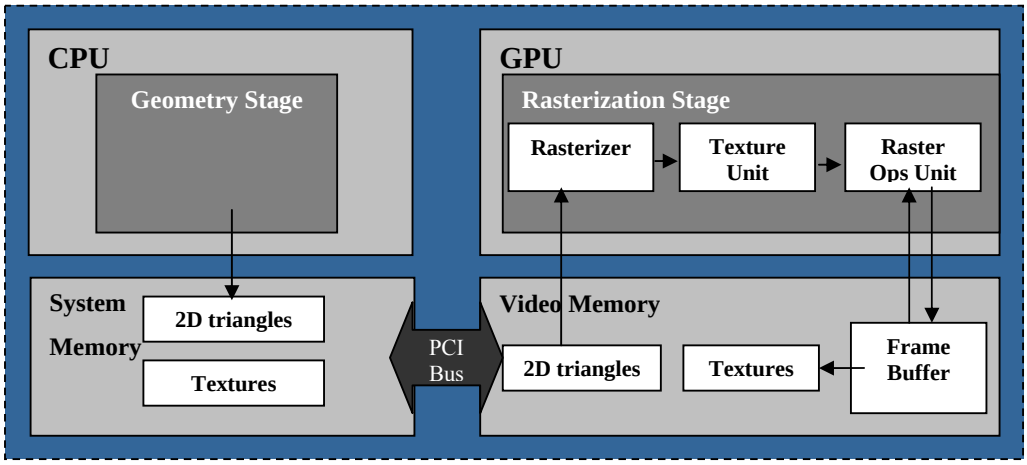

Figure 2.2: *First Generation GPUs (1995)*

As it is shown in Figure 2.2, rasterization stage contains three main units; rasterizer, texture unit and raster operations unit. Raster operations unit (ROU) performs many per-fragment tests before modifying the frame buffer. These tests are shown in Figure 2.3. As stated earlier each fragment has interpolated attributes for position, color, alpha and depth values. In ROU, as a first step, screen position (2d) parameters are tested against scissor rectangle in *scissor test*. Next, fragment's alpha value is tested against a reference value in the *alpha test*. Then, for the *stencil test*, stencil buffer value at corresponding screen position is tested against a stencil reference value. Following stencil test, depth value of the fragment is tested against the z-value at corresponding screen position. Finally, alpha blending is performed with the incoming fragment's color values and corresponding color value of the color buffer. The order of the tests is exactly as stated here.

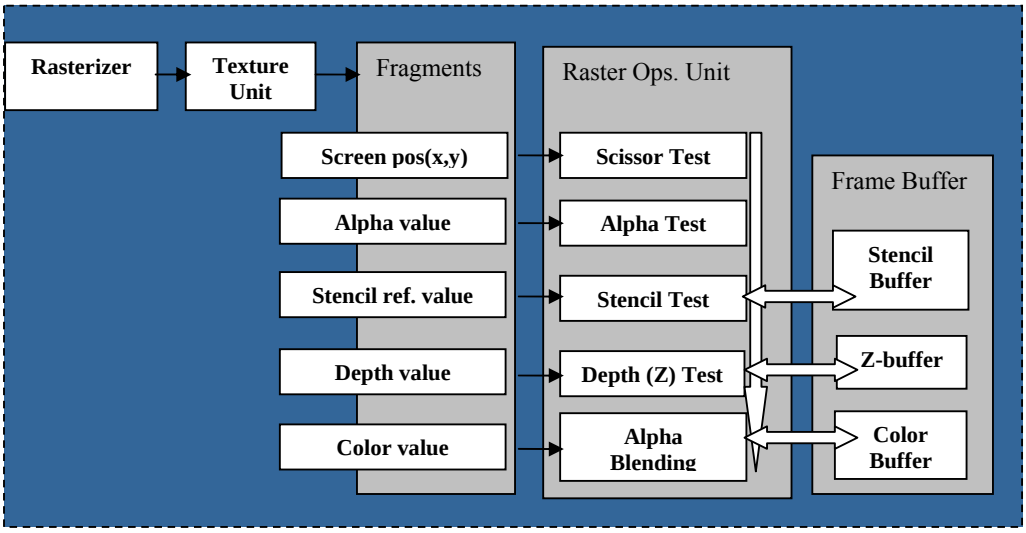

Figure 2.3: *Raster Operations Unit and per-fragment tests*

The second generation GPUs were produced in 1999 and 2000. In this generation, vertex transformation and lighting (T&L) has started to be computed on GPU side rather than CPU (see Figure 2.4 and Figure 2.5). Moreover, the set of mathematical operations of GPU to combine textures and coloring is expanded to include signed mathematical operations and cube map textures. However, GPUs are still in the fixed function pipeline mode and have no programmability features in this generation. Examples of second generation GPUs are NVidia GeForce 256 and ATI Radeon 7500.

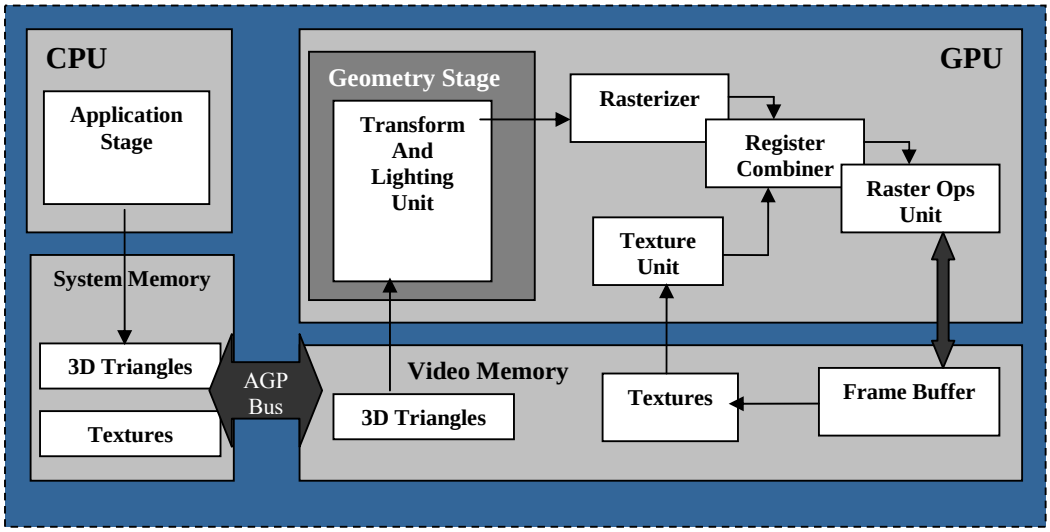

Figure 2.4: *Fixed Function Pipeline (T&L Unit on GPU side)*

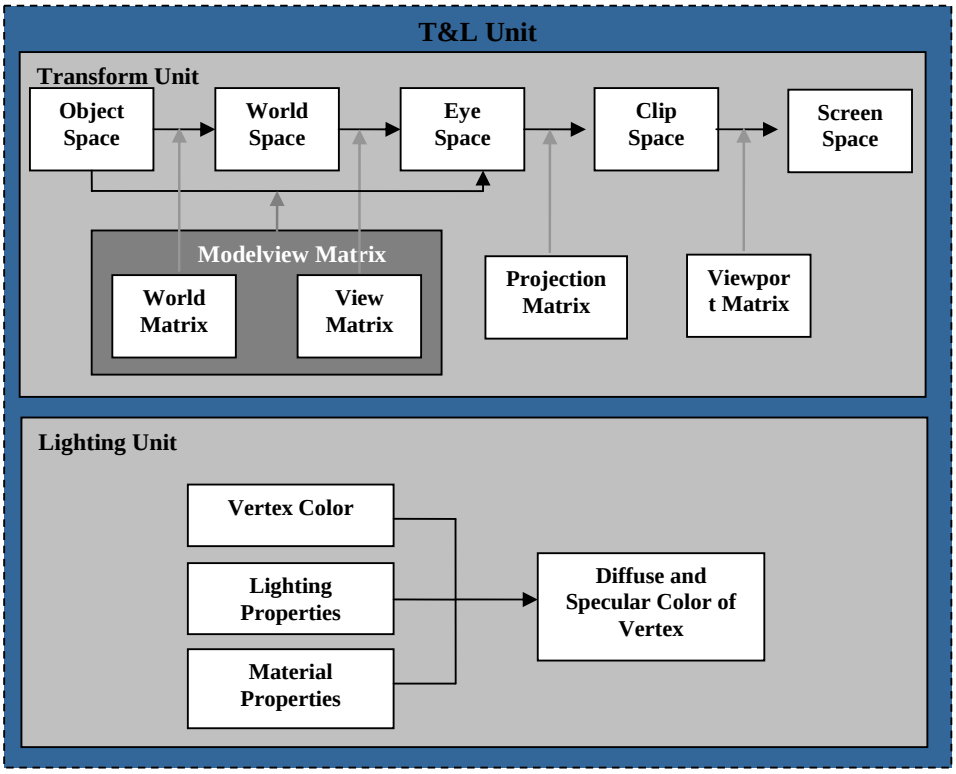

Figure 2.5: *T&L Unit on GPU side*

The third generation GPUs were introduced in 2001 (see Figure 2.6). The GPUs in this generation provide vertex programmability. By this way, the application can specify the sequence of instructions for vertex processing instead of the fixed function T&L modes specified by graphics APIs. Therefore, these GPUs provide more pixel-level configuration variety. However, they are still not truly programmable; there is no flow control support in vertex shaders.

Rasterizer, on the other hand, predicts the fragments that will fail the z-test and discards them. This is called early-z-culling. With this efficient test, unnecessary processing for invisible fragments is avoided. In this generation, texture

shader provides more addressing and texture operations. Examples of third generation GPUs are NVidia GeForce3-4 and ATI Radeon 8500.

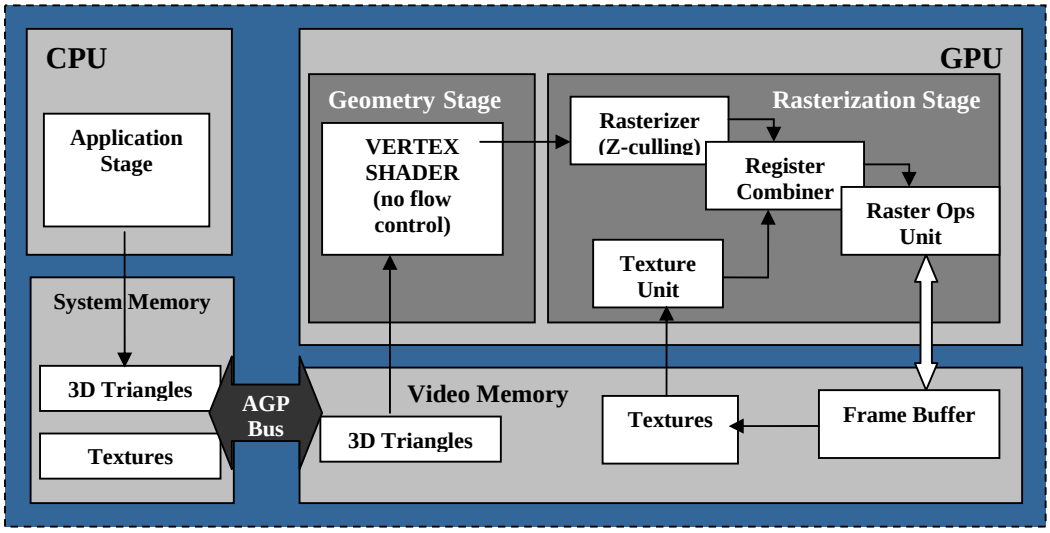

Figure 2.6: *Third Generation Programmable Vertex Processor*

The fourth and the last generation GPUs are the ones produced since 2002 (see Figure 2.7). Both vertex programming and pixel programming are supported by the GPUs of this generation. By this way, complex vertex transformations and pixel shading operations can be performed with the vertex and fragment programs loaded on GPU. In the first GPUs of this generation, vertex shaders support static and dynamic flow control, while fragment shaders support only static flow control. However, later GPUs which were produced in 2004, support both dynamic and static flow control for vertex and fragment shaders. In *static flow control*, a conditional expression in the shader program varies per batch of triangle basis,

while in *dynamic flow control* the condition varies per vertex/pixel basis. Examples are NVidia GeForce FX family GPUs and ATI Radeon 9700.

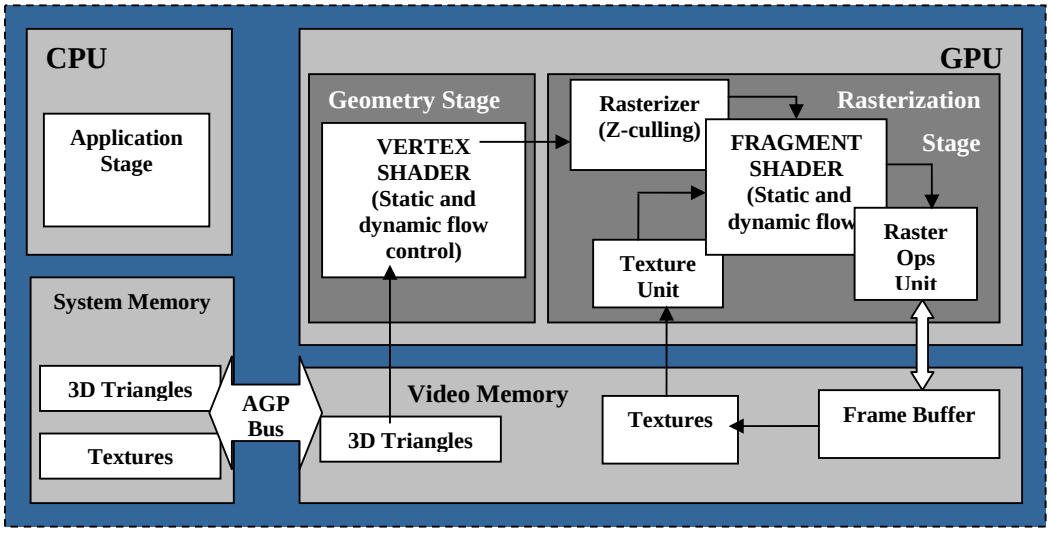

Figure 2.7: *Fourth Generation Programmable Vertex Processor*

2.2 Programmable Vertex and Fragment Processors

Programmable vertex and fragment processing units are the hardware units that run the loaded vertex and fragment programs. With this structure vertex and fragment units have programmability in addition to their configurability. In this section, the basics of the fragment and vertex processors are briefly introduced.

2.2.1 Programmable Vertex Processor

The flow chart of a typical vertex processor is shown in Figure 2.8.

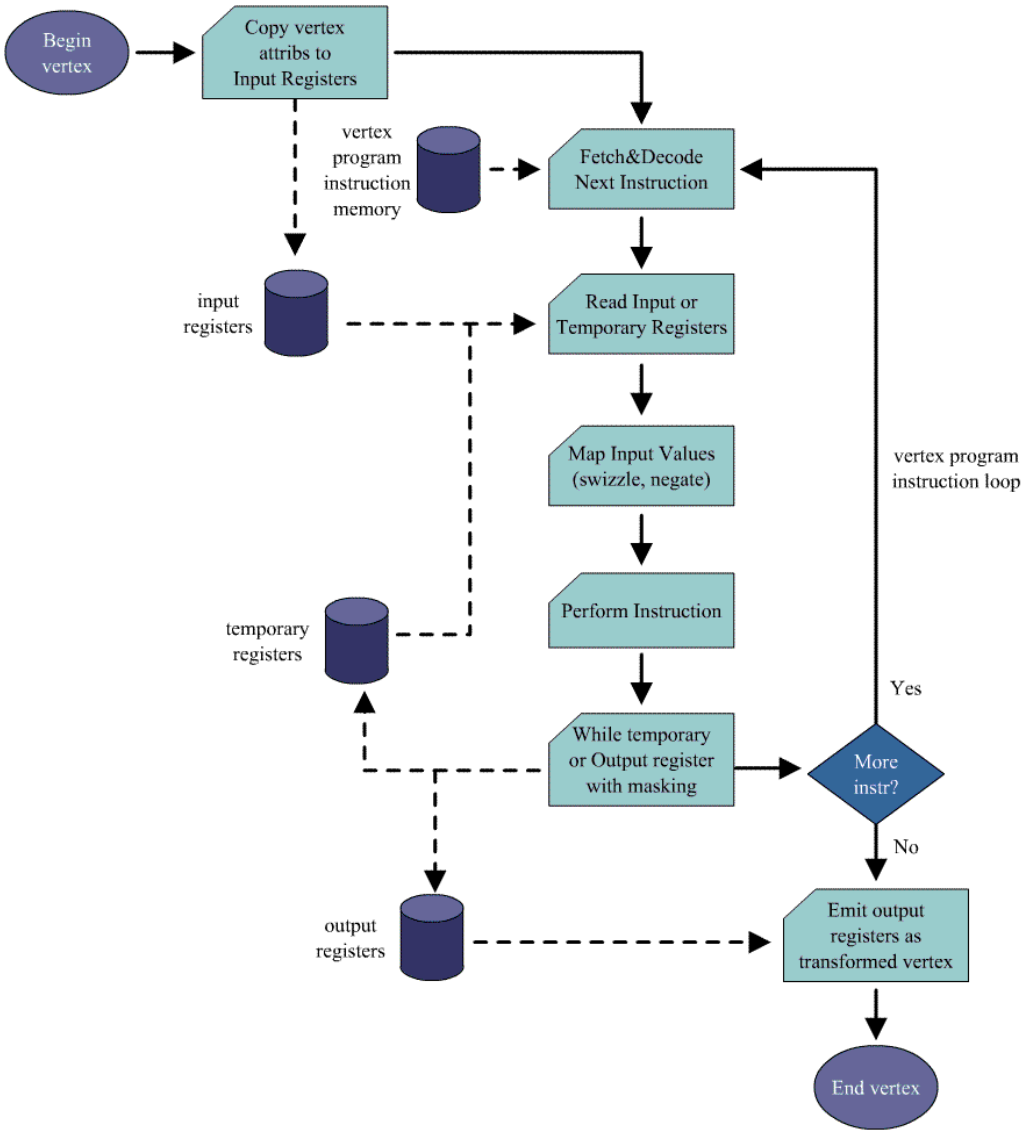

Figure 2.8: *Vertex Processor Flow Chart [34]*

In the flow chart, initially each vertex attributes such as position, color, texture coordinates are loaded into the vertex processor. Until the termination, the vertex processor fetches the next instruction and executes it. There are register banks that contain vector values such as position, normal and color. These registers are accessible from the instructions. There are three different types of registers:

Input registers: These are the read-only registers that contain the attributes particular to a vertex that are specified by the application.

Temporary registers: These registers can either be read or written and they are used for computing intermediate results.

Output registers: These registers are used only for writing. The results of vertex programs are written to these registers. When the vertex program terminates, the output registers contain the final transformed vertex information.

2.2.2 Programmable Fragment Processor

Fragment processors support texture operations in addition to the mathematical operations. The texture samples are fetched according to the given texture coordinates. The flow chart of a programmable fragment processor is displayed in Figure 2.9. Just as programmable vertex processors, programmable fragment processors contain different types of registers as input registers, temporary registers and output registers.

Input registers, different from the ones in vertex processor; contain the interpolated per-fragment attributes obtained from the per-vertex parameters.

Temporary registers contain intermediate results as in the vertex processor.

Output registers contain the color value of a fragment.

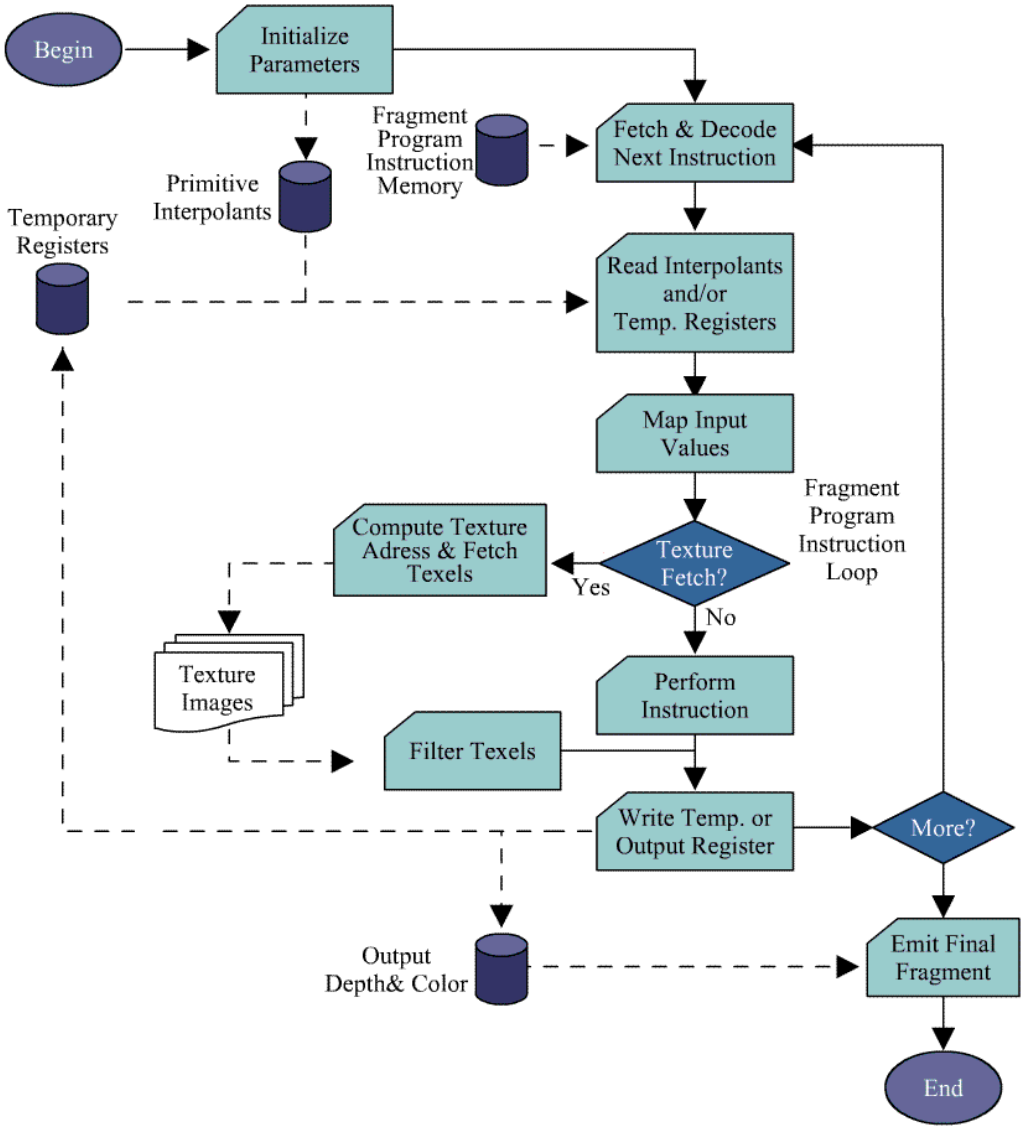

Figure 2.9: *Fragment Processor Flow Chart [34]*

2.3 Programming Interfaces

Graphics applications are developed using Graphics Application Programming Interface (API). An API is the software layer between the graphics hardware device driver and high-level languages. These interfaces prevent the user from learning device specific low-level coding. The quality and efficiency of graphics applications depend on these interface implementations. In an ideal world, an API should add no additional overhead on the applications and it should be platform independent. Moreover, it should provide support for the advances in graphics hardware. The most well known graphics APIs are OpenGL [32] and Direct3D [30]. OpenGL is widely used in industrial and scientific applications, while Direct3D is usually preferred in game programming and entertainment applications. OpenGL is an open standard while Direct3D belongs to Microsoft. Direct3D API is based on Component Object Based (COM), hence its usage is restricted to Windows platforms. In this thesis, OpenGL API is chosen for implementation with C++ language.

2.3.1 Shading Languages

There is an increasing rate in the power of graphics processors. With programmable GPUs, real-time shading capabilities are expanded from one-pass simple shading and simple texturing to multi-pass rendering and to the texture combiners. However, as GPUs become more powerful, programming them becomes more complicated and difficult without existence of high-level shading languages. Producing complicated affects with assembly languages is actually very difficult. Especially, starting from the fourth generation of GPUs, the assembly codes' length exceeds thousands of lines. Hence, recently, the need to high-level shading languages increased dramatically. Graphics developers want easier programming, and code reusing features when programming graphics units.

Considering these situations, new researches are directed to the design and implementation of high-level shading languages. Renderman is the first shader

language that is developed at Pixar in 1988 [4]. It is still a good choice by graphics developers when high quality rendering is required. But it does not work in real-time, generally used for offline rendering. From then, many researches are directed to the development of real-time high-level shading languages. In 1998, PixelFlow Shading System (with shader language and its compiler) is proposed at University of North Carolina as the first real-time shading system [21]. In 2001, Real-time Shading Language is proposed at Stanford University [26]. In this work, the abstraction level of the shading language is increased to a level that causes no performance penalties. Then, in 2002, Microsoft provided a high-level shading language called *HLSL* [33]. Same year Nvidia introduced *Cg* [34]. Next year, in 2003, Architecture Review Board (ARB) provided OpenGL Shading Language as *GLSL* [35]. HLSL, Cg and GLSL languages, different from the early shading languages, give support for many of the previous languages. That is, they work with many general purpose languages like C, C++ and Java; many APIs like OpenGL, Direct3D and with previous shading languages like PixelFlow, RenderMan, Real-Time Shading Language. These high-level shading languages take place between API layer and GPU layer in software architecture as shown in Figure 2.10.

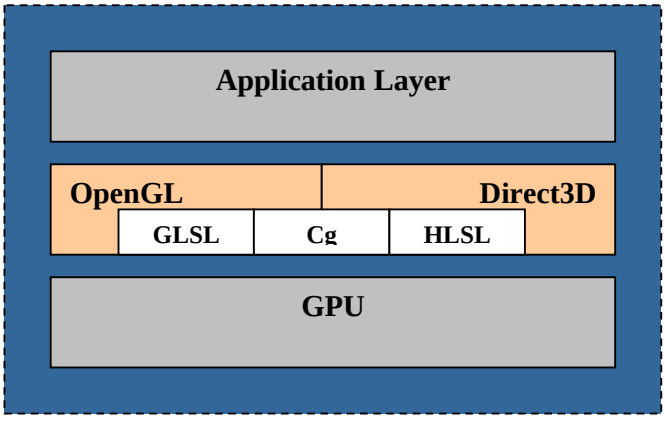

Figure 2.10: *Software Architecture with Shading Languages*

HLSL is developed by Microsoft and it works with Direct3D API. Similarly, GLSL is developed by ARB specific to OpenGL API; it requires OpenGL 2.0. On the other hand, Cg is designed as a platform independent and architecture neutral shading language. In this respect, it is one of the first GPGPU languages that is widely used in many platforms with different languages. In this thesis all the vertex and fragment shaders are developed using Cg language. Refer to the Cg Reference Manuals for the specifications of the language in detail [34]. Details of Cg language and programming are kept beyond the scope of this thesis report.

In general, a graphics program that utilizes shaders initially specifies the vertex and/or fragment shaders using graphics API calls. Then, the specified shaders are enabled. In the program, texture loading and geometry specifications take place as its usual way, using API calls. For each vertex of the geometries, loaded vertex program is executed on vertex processor of the GPU. Similarly, loaded fragment shader on fragment processor is executed for each fragment. Collection of records requiring similar computation like vertex positions, voxels, etc. is referred to as *stream*. Functions specified in the vertex and fragment shaders are applied to each element in the stream. These functions are referred to as *kernels*. Kernels usually have high arithmetic intensity and the dependencies between stream elements in kernels are very few.

After the brief introduction about GPUs, programmable vertex and fragment processors, it is time to make an introduction to texture based volume rendering algorithms. There exists a brief discussion about 2d and 3d texture based volume rendering methods in the next chapter.

CHAPTER 3

TEXTURE BASED VOLUME RENDERING

In this thesis, we studied on texture based volume rendering algorithms and accelerated 3d texture based volume rendering defining a new acceleration structure. Before describing the details of our acceleration method, it is wise to explain the basic principles of texture based volume rendering algorithms. This section is a brief introduction to texture based VR methods.

Sampling of volume data is one of the major components of volume rendering algorithms. It requires interpolation for each sample point throughout the ray. Therefore, its additional cost to the total rendering time is very high. Utilizing the graphics chip's hardware support for interpolation, which exists in the texturing subsystem, hence considerably reduces the load of CPU. Texture based methods utilize the hardware support of texture units for interpolation in sampling calculations. Therefore, these techniques are fairly faster than the software based VR methods.

Texture based methods are classified as 2d texture based methods and 3d texture based methods. The details of these methods are clarified in the following subsections.

3.1 2D Texture Based VR

Graphics pipeline does not support volumetric objects as rendering primitives. Rasterizer supports only the polygonal rendering primitives. Therefore, volume data content should be decomposed into planar polygons for direct volume

rendering. These polygons are referred to as *proxy geometries*. There are different ways of doing the decomposition.

Today, many graphics boards support 2d texture mapping hardware. Hence, utilizing the texture hardware for sampling (interpolation) is advantageous. 2d texture mapping hardware provides bilinear interpolation. In this respect, this method gives similar results with the software implementation of shear-warp method explained in section 1.3. In the next section the basic principles of the algorithm is explained. Following that, the advantages and disadvantages of this method are discussed.

3.1.1 Algorithm

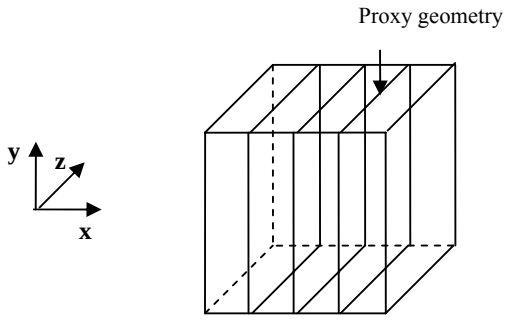

Figure 3.1: *Object-Space Axis-Aligned Data Sampling*

The main part of the algorithm is the definition of proxy geometry. The volume is decomposed into a stack of *object-axis aligned* polygon slices according to the current viewing direction. In Figure 3.1, slicing planes are defined parallel to YZ plane and every slice is rendered as a texture mapped polygon. When texture parameters are assigned truly, the texturing hardware maps the true sampling parameters onto that proxy geometry. These polygons are blended from back to front viewing order to obtain the resultant volume image. While blending,

composition of sample colors and alpha values are performed utilizing DVRI (see section 1.2).

True arrangement of volume slices is the most important part of this algorithm. The slicing is performed on the object space with respect to three major axes. When viewing direction changes, the slicing polygons should be reorganized according to the new view direction to give the true volume image. For example, considering Figure 3.1, if initial view direction is through *x* axis, slices are located parallel to the view plane and we can obtain a proper volume image. Assume that with the same slices, the view direction is set through *z* axis, then the slices become orthogonal to the view plane and we obtain vertical lines instead of volume content as the resultant image. Hence, for each view direction change, reorganization of volume slices is required. However, the reorganization of texture slices on the fly is very costly. For this reason, as it is done in the shear-warp method, for three main object axis, x, y and z, texture sets are prepared in a preprocessing step (see Figure 3.2) and stored in the memory.

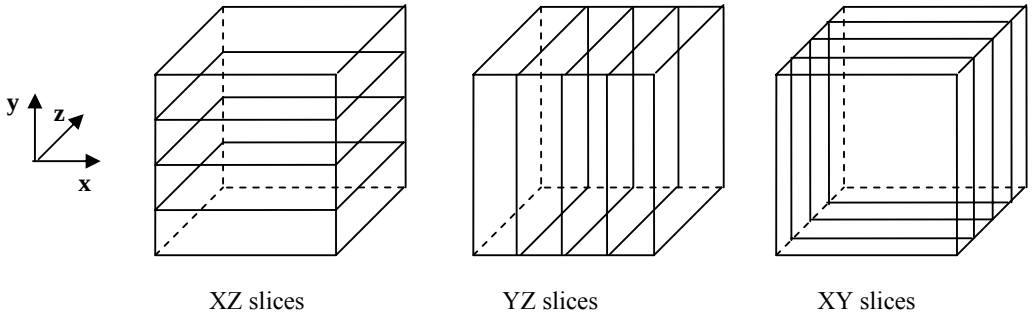

Figure 3.2: *Texture Set for Three Object Space Axis*

The true slicing set is chosen according to the minimal angle between the current view direction and the slice normal. However, in some situations, from one major view direction to the next, the change in the image intensity can be fairly visible. This artifact is called *popping effect*. The cause of popping effect is the abrupt changes in the locations of the sample points depending on the sudden change between two slice sets. The reason of this artifact is depicted in Figure 3.3. In Figure 3.3-*c,* the displacement from one slice set to the other can easily be visualized.

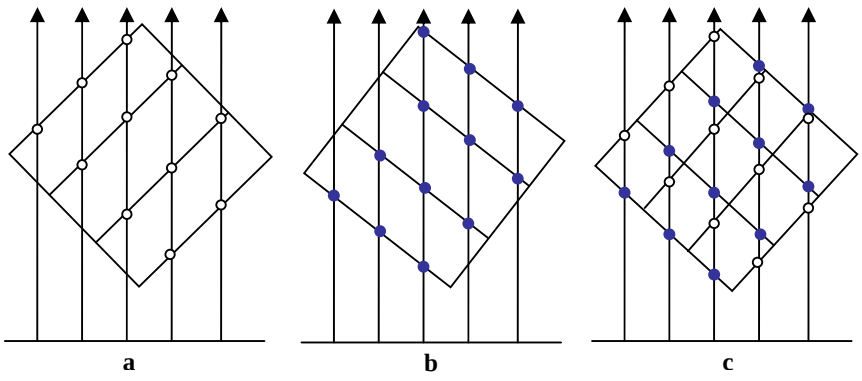

Figure 3.3: *Sample Points for Different Slice Sets (a and b).* c shows the superposition of a and b.

There are some solutions to decrease the popping effect. One of them is decreasing the length between two consecutive sample points. This is accomplished by inserting intermediate slices between two slices. By this way, sampling distances are decreased and the abrupt intensity changes between two different viewing direction is reduced. Another solution is defining image-space axis-aligned slicing

planes, which implies making slices parallel to the view plane all the time. However, it requires defining slices in arbitrary orientations in the object space. Preparing slices in arbitrary orientations on the fly is very time consuming and not feasible with 2d texture mapping methods. This approach is used in 3d texture based methods and will be clarified in the next section.

3.1.2 Advantages and Disadvantages

2d texture based methods have some advantages and disadvantages. Rendering times and high availability are the advantages of this method. On the other hand bilinear sampling, inconsistent sampling rates, visual artifacts and high memory requirements are among the disadvantages of this method.

First, using preprocessed three object axis aligned slices enables the visualization with very high performance. Texture unit accomplishes the interpolation. There remains only the blending operation from back to front viewing order. Hence, rendering can be realized with high performance. Another advantage of this method is its availability. Considering that nowadays all the graphics chips support 2d texture mapping, this method works in almost all the graphics cards.

Contrary to these advantages, there are some disadvantages. Obtaining high quality images with this method is difficult because of the bilinear interpolation done during sampling. Moreover, when slice sets are changed from one major axis to the next, sampling distances change abruptly. This causes inconsistent sampling rates and popping effects. As a final point, the method requires the preparation of the slice sets for each major axis before rendering. These stacks of slices are stored in the memory, which requires high storage capacity.

3.2 3D Texture Based VR

As proposed in the previous section, visual artifacts can be reduced by defining image-space axis-aligned slices rather than object-space axis alignment. The reason of utilizing 2d object-space axis aligned polygon slices are due to the earlier graphics chips' bilinear interpolation support. Preparation of slices in

arbitrary orientations on the fly is very costly with these graphics units. However, new generation graphics cards give support for the trilinear interpolation in their texturing subsystems. This capability enables changing the orientations of the proxy geometries dynamically according to the new view direction.

3.2.1 Algorithm

3d texture based volume rendering algorithms are built on the trilinear interpolation support of texture units in the new generation graphics hardware. All the texture slices are arranged parallel to the view plane in 3d texture based VR methods. This is as shown in Figure 3.4.

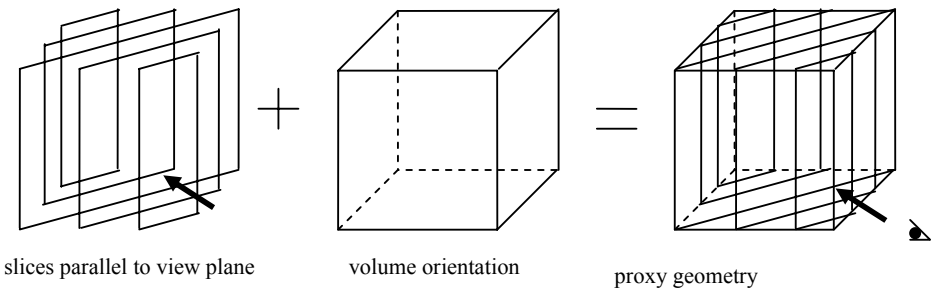

Figure 3.4: *Image-Space Axis Aligned Texture Slices.*

Independent from the orientation of the volume object, the viewer's line of sight is orthogonal to the texture slices. This is accomplished in OpenGL using 3d texture mapping API calls. In 3d texture mapping, each proxy polygon vertex is assigned a point in the texture space (see section 3.2.2). The graphics cards' texture unit maps texture content to the vertices and carries out the required interpolation

between these vertices. It is like Gouraud shading, except that; here the interpolation parameters are textures rather than colors.

3.2.2 Texture Coordinate Generation

In 3d texture mapping algorithms each quad vertex is assigned a point in texture space [19]. The graphics unit provides proper texture coordinate values for the whole polygon surface by interpolating the coordinates at the vertices. Then texture mapping is completed according to the assigned texture coordinates. Interpolation of texture coordinates is performed even for the outside of the range [0, 1]. However, the color values fetched outside this range is clamped to [0, 1]. In the method, the corner vertices of the quads are assigned with the texture coordinates out of the range [0, 1]. Inner parts of the quads will have in-range values by means of the interpolation. The slicing planes are always kept parallel to the view plane in screen space while the volume texture can be oriented in texture space.

Assume that, we defined a coordinate system (x,y,z) that originates from the center of the volume. As known, the texture space counterparts of these coordinates are (s,t,r). A bounding cube is created such that it is centered at the origin and it contains the whole volume inside its body for all different orientations of the volume. This is accomplished by equating one side of the bounding cube to the length of the diagonal of the volume. The bounding cube can be called as proxy volume. The view plane aligned quad slices are actually the slices of this bounding cube parallel to xy plane in this bounding cube's object space. The intension here is to visualize the volume from different directions, with different rotation angles in three coordinate axes. Screen size bounding cube provides this. The idea is displayed in Figure 3.5, below.

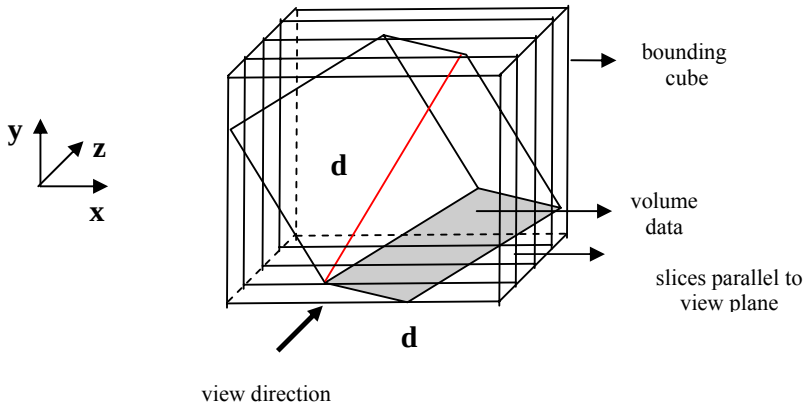

Figure 3.5: *Bounding Cube.* One side of the bounding cube (d) is equal to the diagonal of the original volume.

Main idea assigning proper texture coordinates for the vertices of the bounding cube. The formulization of the texture coordinates are explained in [19] in detail. According to this work, the following formulas generate the proper texture coordinates.

$$s(x) = (x + \frac{1}{2} n_x \Delta x) / (N_x \Delta x)$$

$$t(y) = (y + \frac{1}{2} n_y \Delta y) / (N_y \Delta y)$$

$$r(z) = (z + \frac{1}{2} n_z \Delta z) / (N_z \Delta z)$$

where, the volume has resolution (n_x, n_y, n_z) and spacing in the world coordinates are $(\Delta x, \Delta y, \Delta z)$. Moreover, since the texture map resolutions are represented by powers of two; N_x, N_y and N_z are the least powers of two that are greater than n_x, n_y and n_z.

3.2.3 Advantages and Disadvantages

3d texture based VR methods have both advantages and disadvantages. First advantage of this algorithm is that it generates high quality images. In contrast to 2d

texture based algorithm, 3d texture based algorithm utilizes trilinear interpolation for sampling volume data. The resultant images have same quality with the ray casting methods, if sampling distances are prepared equally in both methods. The second advantage is that, orientation of the texture slices prevents the occurrence of visual artifacts, like popping effect. Hence, high quality interactive visualization without popping effect is possible with this method.

On the other hand, for any change in the viewing direction, reorganization of the texture slices is strictly necessary. All the time, slices should be oriented parallel to the view plane (see Figure 3.6). However, this can be avoided by setting the parallel planes as explained in the section 3.2.2 and modifying only the texture transformation matrix. By this way, the view direction is kept constant, but only the orientation of the volume content in texture space is changed. This has the same effect with the view direction change in world space.

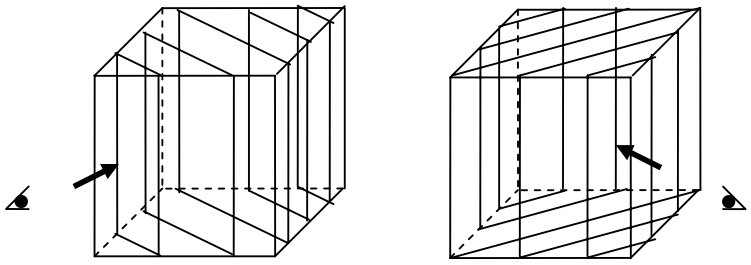

Figure 3.6: *Rearrangement of Texture Slices According to View Direction.*

Moreover, the method works only with the graphics chips that give support for the 3d texture mapping. Hence, this method works with the new generation graphics cards.

CHAPTER 4

ACCELERATED DIRECT VOLUME RENDERING WITH TEXTURE SLABS

This chapter is about the accelerated DVR technique based on the texture slabs that we worked on during the thesis study. During the thesis study different components of volume rendering algorithms are studied such as volume data classification, volume rendering on both CPU and GPU, and acceleration. At first, ray casting based DVR methods and texture based DVR methods are studied and implemented on both CPU and GPU. After obtaining sufficient experience on GPU programming and volume rendering techniques, the study is directed to the design of a new acceleration structure that utilizes the advanced features of the programmable graphics hardware. As a result of this study, we obtained a promising acceleration with the proposed acceleration structure and rendering method. The method works very efficiently on general purpose graphics cards.

First, the fundamentals of the texture slab based DVR algorithm are explained in the following section. Next, the implementation on GPU is given. Subsequently, the main GPU kernels are explained in detail.

4.1 Algorithm

Rendering unit in our method is a texture slab, which is a group of consecutive rectangular texture slices that are parallel to the view-plane (see section 3.2 for details). Texture slices and texture slabs are depicted in Figure 4.1.

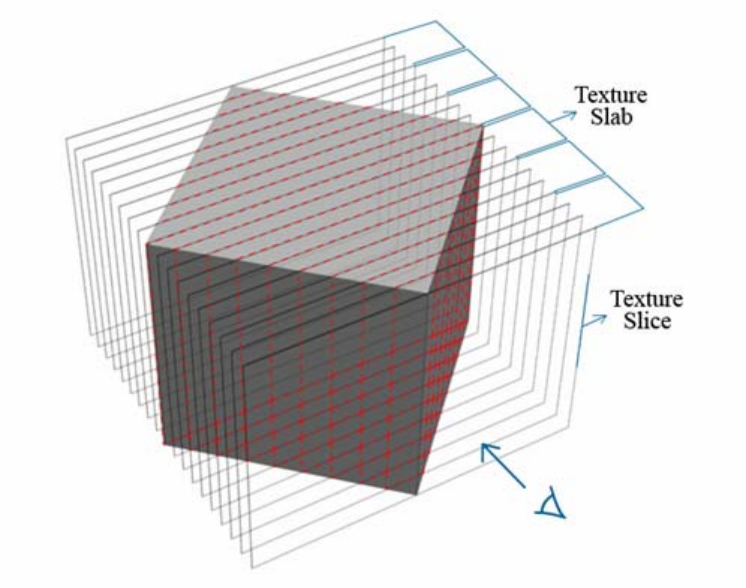

Figure 4.1: *Viewport Aligned Texture Slices and Texture Slabs.*

The proposed algorithm generates the volume image in multiple passes. It means that, the application sends the geometric primitives to the graphics pipeline several times during the generation of the resultant image. In each pass slices of a slab are rendered from front-to-back viewing order. To initiate rendering of a slab a screen sized quad is sent to the GPU. In this manner, screen pixels correspond to the rays.

Rendering a texture slab is performed by setting proper texture coordinates corresponding to the current slabs' starting texture slice. This can be thought as volume rays are cast through the entrance slice of a slab and traverses inside the slab until reaching the ending slice. In this respect, this method is a volume ray casting method that utilizes hardware texture support of GPUs. Rays that exit from a slab enters to the next slab in the subsequent passes. Along the ray path, volume data is sampled in fixed intervals. Therefore, each ray sample actually corresponds to a point on a texture slice.

After the data classification, voxels are assigned the opacity values according to the voxel's scalar density value. These opacity values tell much about the property of a voxel. A fully transparent voxel has an opacity value of 0. This type of voxels contribute nothing to the resultant image. Therefore, rendering these voxels is unnecessary and time-consuming especially when per fragment operations are highly loaded with lighting calculations and texture fetch operations. To avoid processing of those voxels some acceleration techniques are used such as empty space skipping and early ray termination.

At this point, it is wise to give some definitions, such as full region, empty region, empty space skipping and early ray termination that help understanding the concepts related to our acceleration technique. *Full region* is composed of the voxels with opacity values greater than zero or equal to some certain value that we want to visualize. *Empty region*, in contrary to full region, is composed of the voxels with zero opacity or some certain opacity values that we are not interested in visualizing. According to these definitions, it is easy to define Empty Space Skipping and Early Ray Termination concepts, which are the two very important concepts for the volume rendering acceleration techniques.

4.1.1 Empty Space Skipping (ESS)

During the ray traversal, a ray passes from many voxels with different properties throughout the ray direction. As it is stated before, rendering the voxels belonging to empty regions is unnecessary. Hence, many acceleration techniques aim to determine these empty regions before rendering. This information is used to skip the samples inside the voxels that are in one of the empty regions. Skipping the voxels in empty regions and rendering only the voxels in full regions is referred to as Empty Space Skipping. ESS accelerates the rendering time considerably, especially when volume content is sparse.

4.1.2 Early Ray Termination (ERT)

During the ray traversal, a ray continuously accumulates the color and opacity values of the samples through the ray path according to DVRI (see section 1.2). When opacity value of a ray comes very close to 1, the ray color does not change considerably any more. What we mean is that, the pixel corresponding to that ray becomes opaque and change in the pixel color is perceptually brought to a standstill. Therefore, processing the remaining samples through the ray direction is unnecessary. Stopping the traversal after that sample does not change the resultant image. This is called *early ray termination*. It is called early because traversal is ended before the ray actually leaves the entire volume. ERT, when used with ESS, significantly accelerates the rendering time, especially when the volume data contains opaque objects.

ESS and ERT methods are utilized by many acceleration algorithms as explained in section 1.4. All of these techniques create some data structures on the CPU side in a pre-processing step to encode the content of volume data. These data structures are then utilized in ESS to skip empty voxels. The difference of our algorithm is that; the data structure we used is created on the fly on GPU side. Hence, it does not require a pre-processing step to encode volume content. Generating the acceleration structure on the fly on GPU is the main contribution of this thesis study.

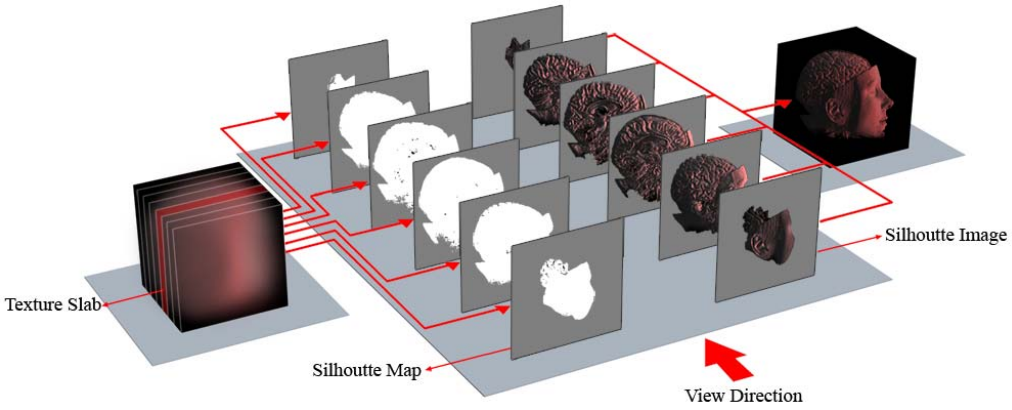

Figure 4.2: *Summary of the Algorithm.*

A brief summary of the algorithm is as the following (see Figure 4.2). Before rendering each slab, its silhouette map is created for the non-terminated regions using the advanced features of the GPU. The Slab Silhouette Maps (SSMs) are used to determine empty regions and early ray terminations. Two different fragment programs are loaded on GPU to modify the contents of SSMs. One of these programs reads the slab's alpha content and encodes the empty spaces according to this information into SSM. The second fragment program reads the accumulated density information obtained after ray traversal. This program determines the early terminated rays and encodes this information into SSM according to the accumulated density information. As a result, SSM contains information about the empty regions of the slab as well as the terminated rays. As the contents of the silhouette map are stored in the depth buffer, graphics hardware can utilize them in early depth tests to skip empty regions and to prevent processing of terminated rays.

4.2 Implementation

We used OpenGL and Cg with fp40 profile for the implementation. A 4-component half-float pixel buffer (PBuffer) with 2 color buffers and a depth buffer

is employed to perform operations. PBuffers are very useful for multi-pass algorithms. They enable reading back the contents of the frame buffer very efficiently. Utilizing PBuffers, a program can modify the contents of the frame buffer in one pass and then read the contents as textures in another pass. Since our algorithm is a multi-pass algorithm we utilized the PBuffers much. During rendering one of the color buffers of the PBuffer is set as the drawing target, while the other one is accessed as the texture source in an alternating fashion.

The algorithm relies on early z-occlusion culling for computation masking. For this purpose GL_ext_depth_bounds_test and OpenGL standard depth test functions are exploited. The details of SSM creation and its utilization are made clear in the following sections.

Our volume renderer calssifies volume data according to classification scheme proposed by Marc Levoy (see section 1.1 for details) and creates 3D volume texture before the rendering stage. As a result of the classification, RGB (red-green-blue) components of the volume texture are set with the approximate surface gradients and alpha component is set with the opacity values that are assigned to each voxel.

An orthographic projection is used for the projection of the volume into image plane. The view frustum is defined as a bounding cube of the original volume, such that for all different transformations the volume data is kept inside the frustum. This is achieved by defining the side length of the frustum equal to the diagonal of the volume. The texture space coordinate assignment to the slice vertices of the bounding cube are calculated exactly as in the formula stated in section 3.2.2. Keeping the texture coordinates constant at the vertices, the volume transformation is performed in texture space. Hence a texture transformation matrix is created that gives the same effect with the world space transformation of the volume. By this way, interpolated texture coordinates outside the range [0, 1.0] are clamped and nothing is visualized in these regions. The fragments that are mapped to the texture coordinates in [0, 1.0] range contain the volume image and generates the resultant image. The algorithm allows changing view position and direction

interactively. All the texture coordinates sent through the rendering pipeline, including the eye and the light position, are transformed into the volume texture space. Traversals and shading operations are performed directly in the texture space.

Considering the GPU as a stream processor, a series of vertex and fragment programs are utilized as for the kernels. The main kernels are depicted in Figure 4.3. In the figure, the blue arrows indicate the buffers bound as textures to a kernel. Red arrows show the rendering targets. The algorithm is constructed on three main kernels, which are responsible for creating the slab silhouette map (CSSM kernel), traversing rays (Ray Traverser kernel) and modifying depth buffer for early ray terminations (ERT kernel). *CSSM kernel* uses the slab textures and renders the processing results into depth buffer. After that, the content of the depth buffer is referred to as SSM, as mentioned earlier. *Ray Traverser kernel* traverses through the non-terminated rays in non-empty regions and makes shading calculations. To determine non-terminated rays and non-empty regions, kernel utilizes SSMs. The shading results are accumulated to the color buffer. Finally, *ERT kernel* reads from the recently modified sections of the accumulated color buffer and modifies the SSM. It is important to note that SSM is never cleared in the course of rendering so as to keep the terminated ray information. The pseudo code of the algorithm is given in Figure 4.4.

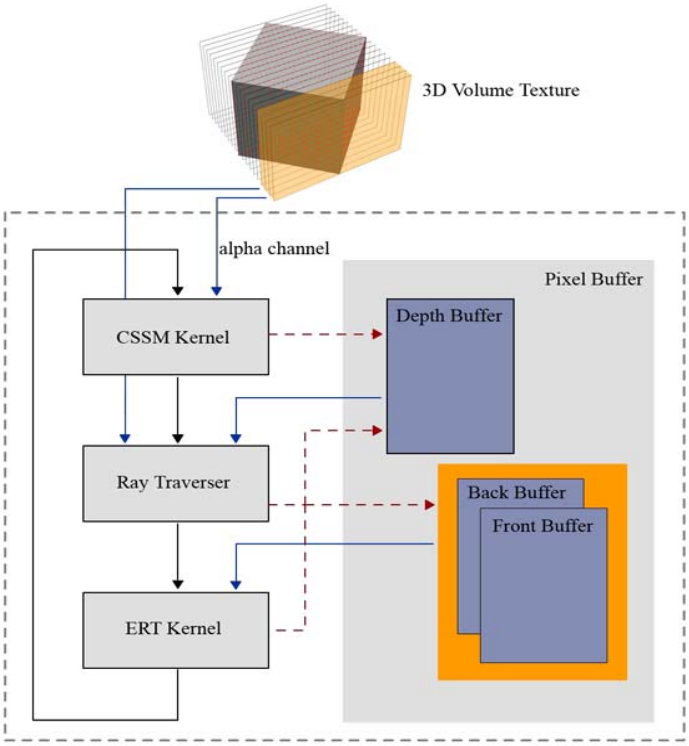

Figure 4.3: *Flow of Kernels and Their Effect to Pbuffer.*

```
      1- Make initializations
            -Initialize lighting parameters
            -Compose texture transform matrix
            -Reset front, back and depth components of PBuffer
      2- For each slab
      3-    Set current draw and accumulation texture buffer
      4-    Determine depth values for full and empty regions (see
Figure 4.5)
      5-    Create SSM (only for non-terminated rays)
      6-    Traverse Slab (only full regions)
      7-    Copy the modified parts of draw buffer to texture buffer
      8-    Check Early Ray terminations (only for the last modified
rays)
      9- End for
      10- Return resultant accumulation buffer
```

Figure 4.4: *Pseudo-code of the Algorithm .*

4.3 Kernel Operations

This section covers the detailed explanations for the kernel operations.

4.3.1 Slab Silhouette Map (SSM) Kernel

Assume that SSM_k represents the SSM of the k'th slab. Rays are cast from the front slice of the slab and traversed throughout the ray direction until the last slice of the slab is processed. SSM is created after this traversal. To clarify the SSM creation, some definitions are essential:

1- Parametric ray equation:

$$R_k(t) = O_k + D * t$$

where:

k: slab index,

t : slice index in a slab,

O_k : origin of the ray on the k'th slab,

D: normalized direction of the ray.

and inside each slab, $0 \leq t < N$ holds. Here N is the total number of slices in a slab.

2- *VAT[P]* : the opacity value of a volume alpha texture at P, where P is a point in texture space.

As it is stated earlier, the proposed algorithm is based on the early z-test feature of the new generation GPUs. We utilized this functionality by means of depth bounds testing. However, depth bounds testing on current graphics cards puts some restrictions to work properly. It works properly only if depth function is set to *GL_LESS* before any depth write operation. In this case, the depth buffer can only be modified if the current fragment's depth value is less than the value stored in the depth buffer. Because of these reasons, a special formulization is required for the selection of proper depth values. Actually, the algorithm contains many depth write sections during rendering. Hence, we developed a mechanism to determine the proper depth value. For each modifications of the SSM the method sets monotonically decreasing depth values to the depth buffer. According to the

definitions 1 and 2, above, it is helpful to define empty and full regions formally as below.

Empty regions in slab *k* corresponding to the pixels through the ray *R*, satisfy the following equation.

$$for\ \forall t,\quad 0 \le t < N,\ \ VAT\left[R_k(t)\right] = 0\,.$$

Similarly, we refer to the regions of slab k corresponding to the pixels through the ray R, as full regions if they satisfy the following equation.

$$\exists t,\quad 0 \le t < N,\quad which\ \ satisfies\ \ VAT\left[R_k(t)\right] > 0\,.$$

According to these definitions, fragment depth values for SSM_k are set along with the function below.

$$depth(k) = \begin{cases} 1 - 2k\tau & for\ \ full\ region \\ 1 - (2k+1)\tau & for\ empty\ region \end{cases}$$

Figure 4.5: *Depth Generation Function.*

Where, τ is the decrement factor, which is set to 0.001 in the application. It allows assigning an adequate amount of decreasing numbers for the empty regions in (0, 1] range during the rendering passes. The depth bounds test initialization parameters should be set properly before rendering in order to efficiently utilize early z-testing feature of the GPUs. The source code for the initialization is as shown in Figure 4.6.

```
glDepthMask(GL_TRUE);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glEnable(GL_DEPTH_BOUNDS_TEST_EXT);
glDepthBoundsEXT(depthEMPTYminus,1.0f);
glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);
```

Figure 4.6: *Early Depth Test Initialization.*

SSM creation is achieved as follows. Initially, the depth buffer is cleared to 1. Then, the vertex and texture coordinates of the current slab's starting slice are sent to the rendering pipeline along with some kernel parameters. The parameters are the sampling distance of each slice in texture space, number of sample points in a slab and ray direction. CSSM kernel sets SSM parameters for full and empty regions for a fragment according to the function defined in Figure 4.5. Color buffers are kept intact during rendering; only the depth buffer is modified. Cg source code of the CSSM kernel is shown in Figure 4.7.

```
fixed sum_res = 0;
half3 currTexCoord = texCoord;
for(int i=0;i<number_of_slices_in_a_slab;i++)
{
      sum_res += tex3D(texVolume,currTexCoord).a;
      currTexCoord += directedTextStep;
}
depth = (sum_res>0) ? fullDepthVal : emptyDepthVal;
```

Figure 4.7: *Cg Source Code of CSSM Kernel. Alpha component of the volume texture is sampled according to the current texture coordinate for all texture slices in the slab. Current texture coordinate is iterated according to the parametric ray equation in the ray direction in order to get the texture coordinates of the subsequent slices. if alpha component of any sample has a value greater than 0 towards the ray direction, depth value is set with fullDepthVal. Otherwise it is set by emptyDepthVal. fullDepthVal and emptyDepthVal parameters are uniform parameters that are set by the application according to the formula stated in Figure 4.5.*

When '*depth'* parameter is set to a value, the depth buffer is modified with that value, if it is an appropriate value as for the depth function. Assume that the depth buffer content for corresponding fragment is set to X and depth function is set to GL_LESS. When fragment program sends a value Y to the depth buffer, X is replaced with Y only if $Y < X$. Otherwise, Y is ignored. Therefore, the depth modification formulation is necessary to effectively use the depth range [0, 1]. This is why a monotonically decreasing order is chosen for modifying the depth buffer.

Another purpose of this acceleration technique is providing an efficient mechanism while creating the acceleration structure. Hence, the additional cost of creating silhouette map to the total rendering time is decreased to minimal by means of utilizing depth bounds test. The idea is based on the fact that traversing through the terminated regions is redundant. Hence, the depth bounds test is enabled before sending geometries to the CSSM kernel on the application side and the depth

bounds are set to $[1-(2k+1)\tau-\delta,1]$, where $\delta < \tau$. This range includes both empty and full regions but excludes terminated rays. Therefore, while creating the SSM, processing of the rays for terminated regions are efficiently avoided.

4.3.2 Ray Traverser Kernel

Ray traverser kernel is the main kernel that performs the traversal of the rays through the ray directions and determines the color of the corresponding pixels. The shading calculations according to the optical properties of the voxels are performed in this kernel. Moreover, the accumulated color and alpha values of the previous slabs are fetched and composited with the current slab's shading results by means of PBuffers in this kernel. The Cg code of the Ray Taverser kernel is shown in Figure 4.8.

```
half3 directedTextStep = directedTextStepParam;
for(int i=0;i<_slabSliceCount;i++)
{
      volColor = tex3D(texVolume,currTextCoord);
      Asrc = volColor.a;

      // lighting part
      L = normalize(objSpaceLightPos - currTextCoord);
      N     = volColor.xyz;
      half3 V = normalize(objSpaceEyePos - currTextCoord);
      half3 H = normalize(L+V);
      diffuseLight = dot(N,L);
      half specularLight = dot(N,H);

      half4 lighting = lit(diffuseLight, specularLight, shininess);

      half3 shade = Kd_lightColor * lighting.y + Ks_lightColor *
lighting.z;
      Csrc = globalAmbient + shade;

      Cdst = Cdst + (1-Adst)*Asrc*Csrc;
      Adst = Adst + (1-Adst)*Asrc;

      currTextCoord +=  directedTextStep;
}

color = half4(Cdst,Adst);
```

Figure 4.8: *Cg Source Code of Ray Traverser Kernel.*

Ray Traverser kernel fetches the surface gradients and opacity values from the volume texture and shades the volume slab according to phong shading. The resultant color and alpha samples are blended with the accumulated color and alpha values with the following formula, since the slicing is performed from front to back viewing order (See DVRI in section 1.2 for the details).

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

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

The content of this kernel is heavily loaded on behalf of arithmetic calculation. Therefore, traversing through each slice sample and accumulating the color and alpha values is very costly. Encoding the empty and terminated regions into SSM and discarding those fragments accelerates the rendering significantly. Hence, this kernel utilizes SSM_k to avoid processing of empty or terminated regions. The depth bounds test is enabled again and the depth test bounds are set to $[1-(2k+1)\tau+\delta,1]$ range. This range guarantees that, only the fragments in the full regions are processed by this kernel. The fragments out of this range have no valuable information for the visualization, hence skipped before entering to this kernel (Figure 4.9).

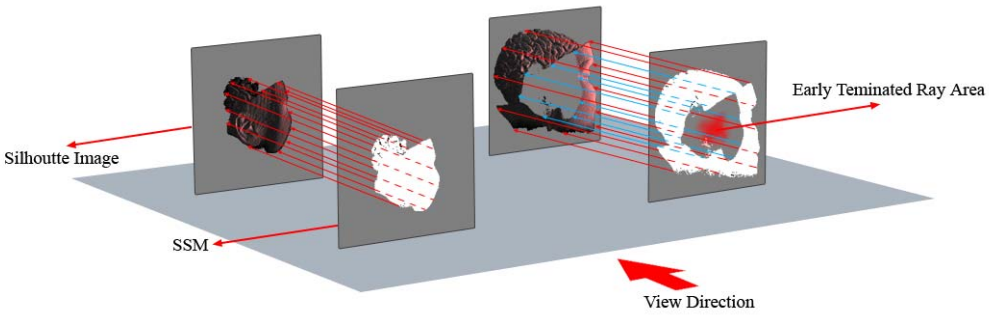

Figure 4.9: *Traversal Through the Full Regions. The two quads on the left side belong to the first slab of the volume; the ones on the right side belong to the second slab. Front quads hold the SSMs of the slabs. The ray traverser kernel only processes the pixels in the white part of the SSMs. Opacity of the first slab is set very high in the first pass and the regions covered by the first SSM in the second slab is marked as early terminated region. Hence, in the second SSM, the gray values inside the head are set as the early terminated regions. The gray values outside the head are set as the empty spaces. Hence processing in these parts is avoided.*

The initialization of the OpenGL states and the depth bounds are set as shown in Figure 4.10.

```
glDepthMask(GL_FALSE);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glEnable(GL_DEPTH_BOUNDS_TEST_EXT);
glDepthBoundsEXT(depthEMPTYplus,1);
glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
```

Figure 4.10: *Initialization of OpenGL States for Ray Traverser Kernel. Depth bounds are set to [1-(2k+1)τ+δ, 1] for the k'th slab.*

4.3.3 ERT Kernel

After the ray traverser kernel, the contents of the current color buffer are set with the accumulated color and alpha values. In the next rendering pass, ERT kernel is loaded on GPU and fragments are processed with this kernel.

When the accumulated opacity value of a fragment comes very close to 1, subsequent rendering passes do not add visible contribution to the final color value. Therefore, ERT modifies SSM_k such that fragments with high enough opacity values are no longer processed in the following rendering passes. In other words, early ray termination is performed.

ERT kernel uses accumulation color buffer as source texture and modifies the SSM considering the predetermined opacity threshold. The depth buffer values corresponding to the terminated rays are set to 0. Setting depth values to 0 blocks those areas such that neither CSSM kernel nor the Ray Traverser kernel can make further execution in these areas. This is because of the fact that, the depth bounds testing is utilized in these kernels with the lower depth bound for the computation

masking is set always greater than 0. SSM modification code with ERT kernel is depicted in Figure 4.11.

```
textureDensity = texRECT(accumTexture,texCoord).a;
clampToZeroOne = saturate(alphaThreshold-textureDensity);
depth = sign(clampToZeroOne)*fullDepthVal;
```

Figure 4.11: *Cg Source Code of ERT Kernel. The alpha component of the accumulated color buffer is fetched using texRECT routine. Then, this value is compared against the alphaThreshold. If accumulated opacity is greater than the threshold, depth is set to 0, representing the terminated region. Otherwise it is set to fullDepthValue, representing unterminated and non-empty region for the following rendering passes. alphaThreshold, fullDepthVal and accumTexture are aplication specified uniform parameters.*

The additional cost of execution time of the ERT kernel is minimized yet again with the depth bounds testing. Checking only the most recently modified regions in the SSM is sufficient to determine the latest terminated rays. Processing this kernel on the empty regions is pointless. In addition to that, processing ERT kernel for the previously terminated regions is also redundant. Because, only the most recently updated rays have the potential of accumulating high opacity values. For this purpose the SSM_k is used once more as a calculation mask. The depth bounds are set to the $[1-(2k+1)\tau+\delta,1]$ range. This range spans only the most recently modified regions by the ray traverser kernel. By this way, additional time caused by this kernel is reduced extensively.

CHAPTER 5

DISCUSSION AND RESULTS

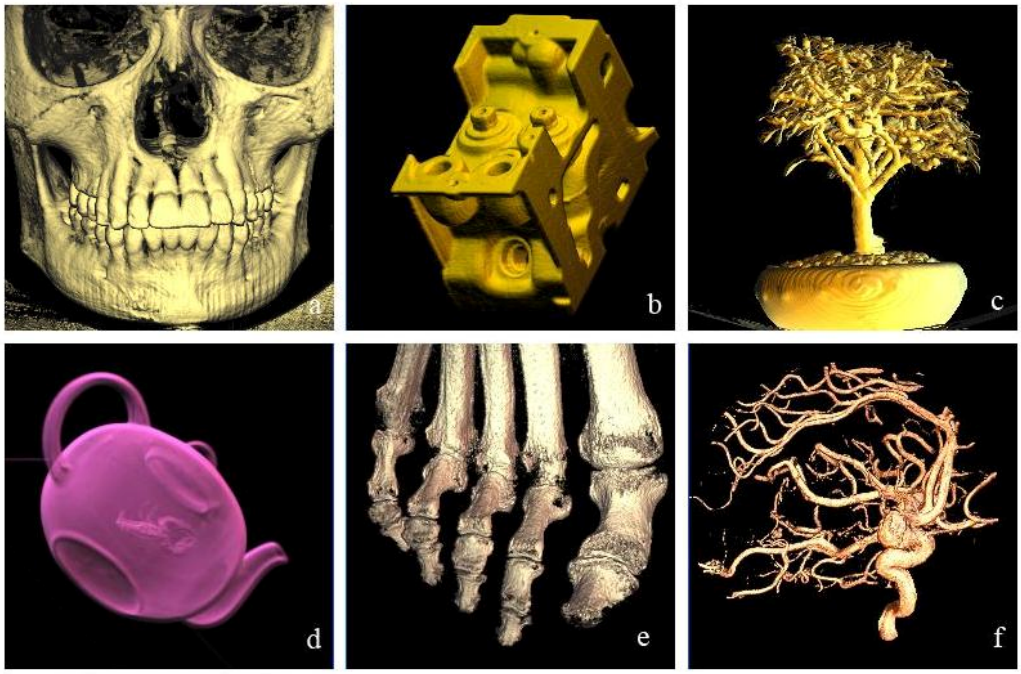

Figure 5.1: *Rendering Results of the Datasets. (a) skull (256x256x256), (b) engine (256x256x128), (c) bonsai (256x256x256), (d) teapot (256x256x178), (e) foot (256x256x256), (f) aneurism (256x256x256).*

The proposed stream programming algorithm for volume ray casting using texture slabs has many benefits over the traditional texture based volume rendering algorithms. In addition to providing equal sample rate through the ray direction, the algorithm reduces the amount of fragments to be processed. The reduction increases especially if the volume contains opaque structures or empty spaces.

We have performed our experiments on Pentium 4, 2.4GHz PC with 512MB RAM using GeForce 6800 Ultra graphics card. We used six datasets to assess the performance of our algorithm. Volume data sets are obtained from [31]. The rendering results are shown in Figure 5.1. Size of the viewport is set to 512^2. The data sets are rendered with several methods for comparison, including classic texture slice based rendering method *(SB)*, slab based rendering method with early ray terminations and empty space skipping enabled *(ASLAB)* and without early ray terminations and empty space skipping *(SLAB)*. In SB method, we render the texture slices in front to back viewing order. The rendering times and relative acceleration of our method is given in Table 1 and Table 2. The execution speed depends on many factors including the number of ray samples (total slice count), selected shading method, image size, volume size etc. In these experiments, the maximum number of volume samples through the ray directions is selected as 15×30 (15 slabs with 30 slices each and hence 450 sample points trough the ray directions). One point light source is used for shading. For consistency, we strictly conformed to the same settings while taking the experiment results for all the algorithms defined above.

Table 1: *Performance Results of the Experiments.*

	SB	SLAB	ASLAB	Processed Fragments	Speedup
aneurism	307	263	82	5.7%	374%
teapot	294	251	81	12.3%	362%
bonsai	296	251	86	9.9%	344%
engine	441	380	75	5.3%	588%
foot	307	262	91	10.29%	337%
skull	306	262	129	21.9%	237%

In Table 1, *SB* corresponds to slice based volume rendering time, *SLAB* corresponds to slab based rendering time without SSMs, *ASLAB* corresponds to accelerated slab based rendering time. Moreover, *Processed Fragments* shows the percentage of total processed fragments in ASLAB and *Speedup* shows the acceleration percentage of ASLAB against SB. Time results are all in milliseconds. As the results in Table 1 show, activating the early ray termination method and empty space skipping using SSMs provide huge performance gains. The percentages shown in the table validates the speedup, such that the rendering time decreases in accordance with the number of processed fragments. Among the dataset, *engine* largely benefits from early ray terminations as most of the rays are terminated in the outer shells of the model. It is displayed in Figure 5.2. According to the SSMs and Table 1, only 5.3% of the fragments are in the full regions and processed during rendering.

Figure 5.2: *First 10 SSMs of the Engine Model. Orange color (■) represents the processed regions. Black colors (■)for early terminated regions. Remaining gray colors are for empty regions. Empty regions and early terminated regions are discarded.*

Aneurism on the other hand, benefits from empty space skipping, as most of the slab regions are empty.

Similarly for *teapot*, the speedup is due to the empty space skipping; although almost no rays are terminated because of the transparent material properties. SSMs of *teapot* data are displayed in Figure 5.3. Only 12.3% of the total fragments are processed during rendering. Most of the fragments fall into the empty regions for this data.

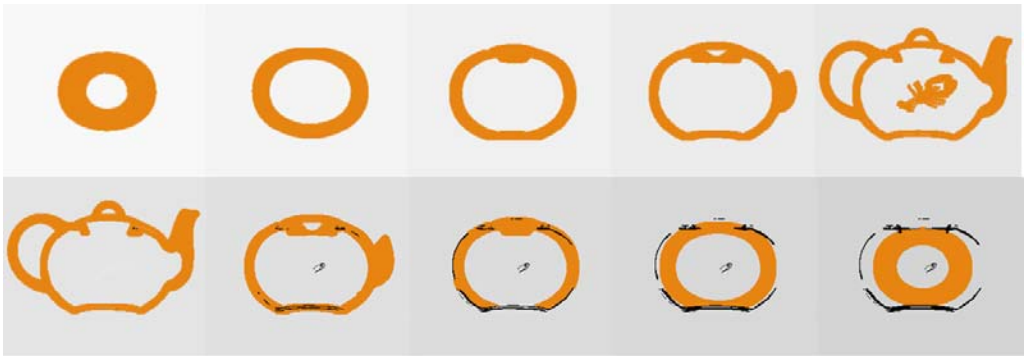

Figure 5.3: *First 10 SSMs of the Teapot Model. Empty space skipping is utilized much. Early ray terminations are rare.*

As for the skull model, there are a few empty regions. However, due to the opaque material properties, there are large numbers of early ray terminations. Hence, in total 21.9% of the fragments are processed. SSMs of the skull model are shown in Figure 5.4.

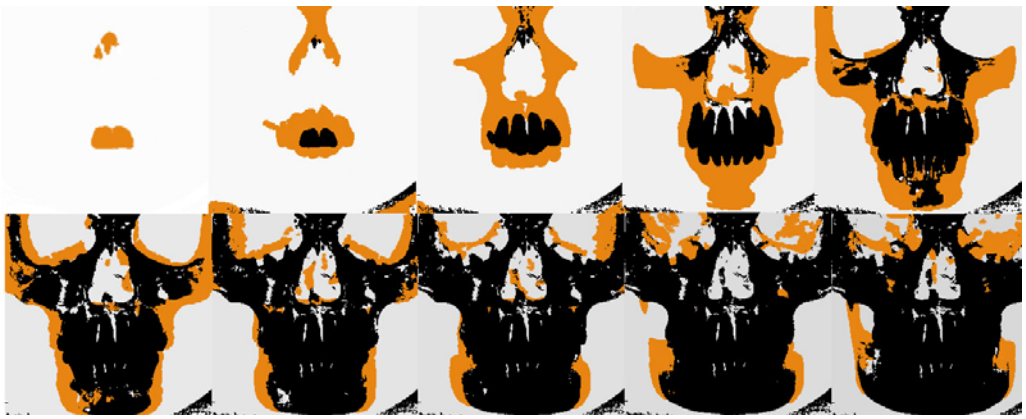

Figure 5.4: *First 10 SSMs of the Skull Model. Empty space skipping is little. However, there are many early ray terminations.*

Table 2: *Total Kernel Execution Times in Milliseconds.*

	CSSM	**Ray Traverser**	**ERT**	**Total**
aneurism	31	42.7	1.2	82
teapot	29	44.7	1	81
bonsai	26.7	53	1.1	86
engine	31	39	2	75
foot	28	57.5	1.2	91
skull	23	99	1.7	129

We have developed our method based on the observation that the silhouette map generation is significantly cheaper than complex ray traversals and shading. This situation can be clearly verified from Table 2. Although the ray traversal kernel ignores the empty and terminated regions, it is still slower than the silhouette map generation. Also note that, the additional cost of running ERT kernel is very small compared to the total rendering time. Therefore it does not cause a bottleneck in the algorithm. Our platform facilitates using long fragment programs, looping and data dependent branching. Therefore, it is possible to combine all of the main kernels into one large kernel. However, we decided to split the kernels as described in chapter 4, so that the algorithm can work with minor modifications in the GPUs which do not support the mentioned features.

The pleasing results of the tests performed on test data sets obviously depict the effectiveness of our acceleration algorithm.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

In this thesis study, we proposed a new acceleration technique for the 3d texture based volume rendering methods, which utilizes the programmable graphics hardware and occlusion z-culling. We applied empty space skipping and early ray termination acceleration techniques to the slice based volume rendering. By the proposed acceleration structure, volume datasets are visualized in high quality at interactive frame rates.

The study covers three major components. One of them includes studying and implementation of ray casting based and 3d texture based volume rendering methods utilizing standard graphics APIs. In this part, main components of a volume renderer are created such as determining transfer function, data classification and shading calculations. These fundamentals are utilized during the study.

The second component of the study includes gaining knowledge about stream programming. Programming graphics hardware units requires knowledge and experience about stream programming. In this part different rendering techniques are studied using advanced features of new generation GPUs. The basic principles, difficulties and bottlenecks of GPU programming are discovered. Programming is made using Cg language.

The third part includes adaptation of the standard volume rendering algorithms on GPU. The algorithms are redesigned such that they can work efficiently with the logic of stream programming. Both ray casting based and 3d texture based volume rendering methods are adapted to work on GPUs. Main kernel

designs and implementations are completed in this part. Utilization of both methods is realized by defining texture slabs as the rendering units. Texture slices are sent to GPU as in standard 3d texture based methods while fragment programs perform volume ray casting inside a slab. After adapting the basic part of the volume rendering, the focus of the study is directed to the acceleration method. Utilizing the experiences we obtained during the study about programmable graphics hardware, we created an acceleration structure on GPU side. This structure uses the occlusion z-culling feature of the graphics unit. The acceleration structure does not require any pre-processing on CPU part for volume region encoding. It is created on the fly on GPU part. This acceleration structure is utilized for empty space skipping and early ray terminations.

The results of our acceleration method showed that rendering is performed with huge performance gains with the proposed acceleration structure. For some volumetric data sets, the volume content is composed of empty regions and opaque objects. Processing only a tiny percentage of the whole volume is sufficient for generating the final volume image. With our acceleration method, empty regions are discarded efficiently in pixel basis and only the full regions are processed. Since the rendering time is proportional to the number of fragments processed, discarding fragments corresponding to empty regions accelerates the rendering time significantly. Moreover, the proposed algorithm is designed in a way that the time required to create of the acceleration structure is very small compared to the ray traversal time. This is the second advantage of the algorithm.

In summary, we worked on the acceleration of 3d texture based volume rendering method utilizing the advanced features of the programmable graphics hardware. The proposed algorithm runs on new generation general purpose GPUs very efficiently. Interactive visualization of the volume data is achieved with high image quality with the proposed acceleration method.

6.1 Future Works

We are currently working on accelerating the creation of the silhouette maps further. This is possible by sending the projected polyhedral faces defined by the intersection of the volume and slab regions to the rendering pipeline, instead of screen sized quads. This can be achieved automatically by setting appropriate OpenGL clipping planes. In addition to that, more acceleration can be achieved by using lower resolution volume alpha texture for SSM generation, as it decreases the required bandwidth.

Furthermore, in addition to the empty, full and terminated region information, an additional attribute may be added into SSMs for homogenous region encoding. This can be achieved on the fly by tracking equal opacity values during ray traversal. Homogenous region encoding accelerates the execution time of the ray traverser kernel, which is the real bottleneck of the direct volume rendering algorithms. Therefore, homogenous region encoding can provide additional performance gains, when used with empty space skipping and early ray terminations.

REFERENCES

[1] W. E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", ACM SIGGRAPH'87, pp.163-169, 1987.

[2] M. F. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt. "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", Proceedings of SIGGRAPH'88, vol. 22, pp. 21–30, August 1988.

[3] L. Westover, "Footprint Evaluation for Volume Rendering", SIGGRAPH'90, pp. 367-376, 1990.

[4] S. Upstill, "The RenderMan Companion", Addison-Wesley, 1990.

[5] M. Levoy, "Display of Surfaces from Volume Data", IEEE Comp. Graph. , vol. 9, no. 3, pp. 245-261, 1990.

[6] K. R. Subramanian, and D. S. Fussel, "Applying Space Subdivision Techniques to Volume Rendering", Proceedings of Visualization'90, pp. 150-158, 1990.

[7] W. Krueger, "The Application of Transport Theory to the Visualization of 3D Scalar Fields", IEEE Visualization Proceedings, pp. 273-280, 1990.

[8] J. Wilhelms and A. Van Gelder , "A Coherent Projection Approach for Direct Volume Rendering", Proceedings of SIGGRAPH, 1991, vol 25, no. 4, pp 275-284, 1991.

[9] D. Laurm P. Hanrahan "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering", Proceedings of SIGGRAPH'91, pp. 285-288, 1991.

[10] J. Danskin and P. Hanrahan, “Fast Algorithms for Volume Ray Tracing”, Workshop on Volume Visualization, pp. 91-98, 1992.

[11] T. Malzbender, “Fourier Volume Rendering”, ACM Transactions on Graphics, vol. 12, pp. 233–250, 1993.

[12] J. K. Udupa, and D. Odhner, “Shell Rendering”, IEEE Computer Graphics and Applications, pp. 58-67, 1993.

[13] K. Akeley, “Reality Engine Graphics”, Proceedings of SIGGRAPH’93, pp. 109–116, August 1993.

[14] T. J. Cullip, and U. Neumann, “Accelerating Volume Reconstruction with 3D Texture Mapping Hardware”, Technical Report TR93-027, Department of Computer Science, University of North Carolina, Chapel Hill, 1993.

[15] P. Lacroute and M.Levoy, “Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform”, Proceedings of SIGGRAPH’94, pp. 451-458, 1994.

[16] B. Cabral, N. Cam, and J. Foran, “Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware”, Symposium on Volume Visualization, pp. 91-98, 1994.

[17] D. Cohen, and Z. Shefer, “Proximity Clouds: An Acceleration Technique for 3D Grid Traversal”, The Visual Computer, vol. 10, no. 11, pp. 27-38, 1994.

[18] N. Max, “Optical Models for Direct Volume Rendering”, IEEE Transactions on Visualization and Computer Graphics,vol1, pp. 99-108, 1995.

[19] A. V. Gelder, K. Kim, “Direct Volume Rendering with Shading via Three-Dimensional Textures”, ACM Symposium on Volume Visualization’96, R. Crawfis and C. Hansen, Eds., pp. 23–30, 1996.

[20] R. Westermann and T. Ertl. “Efficiently Using Graphics Hardware in Volume Rendering Applications”, Proceedings of SIGGRAPH’98, pp. 169–178, 1998.

[21] M. Olano, and A. Lastra, “A Shading Language on Graphics Hardware: The PixelFlow Shading System”, Proceedings of SIGGRAPH’98, pp. 159-168, 1998.

[22] M. Meißner, U. Hoffmann, and W. Straßer, "Enabling Classification and Shading for 3D Texture Mapping Based Volume Rendering", Proceedings of IEEE Visualization'99, pp. 207–214, 1999.

[23] J.Huang, K. Mueller, R. Crawfis, D. Bartz, M. Meißner, "A Practical Evaluation of Popular Volume Rendering Algorithms", IEEE Symposium on Visualization, pp. 81-90, 2000.

[24] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization", Proceedings of Eurographics/SIGGRAPH Graphics Hardware Workshop 2000, pp. 109-118, 2000.

[25] K. Engel, M. Kraus, and T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading", Siggraph/Eurographics Workshop on Graphics Hardware 2001, pp. 9-16, 2001.

[26] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan, "A Real-Time Procedural Shading System for Programmable Graphics Hardware", Proceedings of SIGGRAPH 2001, pp. 159-170, 2001.

[27] W. Li, and A. Kaufman, "Texture Partitioning and Packing for Accelerated Texture-Based Volume Rendering", Graphics Interface 2003, pp. 81-88, 2003.

[28] W. Li, and K. Mueller, and A. Kaufman, "Empty Space Skipping and Occlusion Clipping for Texture-Based Volume Rendering", IEEE Visualization 2003, pp. 317-324, 2003.

[29] J. Krüger and R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering", IEEE Visualization 2003, pp. 287-292, 2003.

[30] N. Thompson, "3D Graphics Programming for Windows 95", Microsoft Press, 1996.

[31] Real World Medical Datasets Home Page, http://volren.org, last access date June 2005.

[32] Official OpenGL Website, http://www.opengl.org, last access date June 2005.

[33] Microsoft DirectX Home Page, http://www.microsoft.com/windows/directx, last access date June 2005.

[34] NVidia Developer Home Page, http://developer.nvidia.com, last access date June 2005.

[35] OpenGL Shading Language, http://www.opengl.org/documentation/oglsl.html, last access date June 2005.