ACCELERATED RAY TRACING USING PROGRAMMABLE GRAPHICS PIPELINES

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

BY

Ş. ALPHAN ES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

JANUARY 2008

Approval of the thesis

ACCELERATED RAY TRACING USING PROGRAMMABLE GRAPHICS PIPELINES

submitted by **Ş. Alphan Es** in partial fullfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering**, **Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, **Graduate School of Natural And Applied Sciences**

Prof. Dr. Volkan Atalay
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Veysi İşler
Supervisor, **Computer Engineering, METU**

Examining Committee Members:

Prof.Dr. İsmail Hakkı Toroslu
Computer Engineering, METU

Assoc. Prof. Dr. Veysi İşler
Computer Engineering, METU

Assoc. Prof. Dr. Uğur Güdükbay
Computer Engineering, Bilkent University

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering, METU

Assist. Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering, METU

Date:

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 : Ş. Alphan Es

Signature :

ABSTRACT

ACCELERATED RAY TRACING USING PROGRAMMABLE GRAPHICS PIPELINES

Es, Ş. Alphan
Ph.D., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Veysi İşler

January 2008, 127 pages

The graphics hardware have evolved from simple feed forward triangle rasterization devices to flexible, programmable, and powerful parallel processors. This evolution allows the researchers to use graphics processing units (GPU) for both general purpose computations and advanced graphics rendering. Sophisticated GPUs hold great opportunities for the acceleration of computationally expensive photorealistic rendering methods. Rendering of photorealistic images in real-time is a challenge. In this work, we investigate efficient ways to utilize GPUs for real-time photorealistic rendering. Specifically, we studied uniform grid based ray tracing acceleration methods and GPU friendly traversal algorithms. We show that our method is faster than or competitive to other GPU based ray tracing acceleration techniques. The proposed approach is also applicable to the fast rendering of volumetric data. Additionally, we devised GPU based solutions for real-time stereoscopic image generation which can be used in companion with GPU based ray tracers.

Keywords: GPU, ray tracing, volume visualization, stereoscopic rendering.

ÖZ

PROGRAMLANABİLİR GRAFİK İŞLEMCİLERİ İLE HIZLANDIRILMIŞ IŞIN İZLEME

Es, Ş. Alphan
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Veysi İşler

Ocak 2008, 127 sayfa

Grafik donanımları zamanla sadece yollanan üçgenleri çizen basit cihazlardan; çok yönlü, programlanabilir ve güçlü paralel işlemcilerine dönüştüler. Bu evrimleşme grafik hesaplama ünitelerinin (GPU) genel amaçlı hesaplamalarda olduğu kadar, ileri görüntü sentezleme hesaplamalarında kullanılmasına da olanak tanıdı. Sofistike GPUlar yüksek hesaplama gücü gerektiren foto gerçekçi görüntü üretiminin hızlandırılması konusunda pek çok olanak sunmaktadırlar. Foto gerçekçi görüntülerin gerçek zamanda hesaplanması henüz çözüme kavuşturulmamış bir alandır. Bu çalışmada, GPUların gerçek zamanlı foto gerçekçi görüntü üretiminde etkin kullanımına yönelik yöntemler araştırılmıştır. Özellikle düzgün ızgaralama veri yapısı ile, çok kullanılan bir foto gerçekçi görüntü hesaplama yöntemi olan ışın izlemenin GPU üzerinde hızlandırılması ele alınmıştır. Tezde, geliştirdiğimiz GPU tabanlı ışın izleme metodunun diğer GPU tabanlı yöntemlere göre karşılaştırılabilir veya pek çok sahne için daha hızlı performans sergilediği gösterilmiştir. Yine bu metodun hacimsel verilerin hızlı görüntülenmesinde de kullanılabileceği gösterilmiştir. Ek olarak, GPU tabanlı hızlı stereo görüntü üretimi yöntemleri geliştirilmiş ve bu yöntemlerin GPU tabanlı ışın izleyicilerle etkin bir şekilde kullanılabileceği gösterilmiştir.

Anahtar Kelimeler: GPU, ışın izleme, hacimsel görüntüleme, stereo görüntü üretimi.

ACKNOWLEDGMENTS

The author wishes to express his deepest gratitude to his supervisor Assoc. Prof. Dr. Veysi İşler for his guidance, advice, criticism, encouragements and insight throughout the research.

The author would also like to thank Hacer Yalım Keleş for her valuable cooperation during the thesis.

To My Family

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

ACKNOWLEDGMENTS vi

DEDICATON vii

TABLE OF CONTENTS viii

LIST OF FIGURES xi

LIST OF TABLES xv

CHAPTER

1 INTRODUCTION 1

- 1.1 Goals and Contributions 3
- 1.2 Photorealistic Rendering 4
  - 1.2.1 Global Illumination 4
  - 1.2.2 Ray Tracing 6
- 1.3 Outline 10

2 GPU BASED REAL-TIME PHOTOREALISM 11

- 2.1 The Rendering Pipeline 12
  - 2.1.1 GPU Pipelines 14
  - 2.1.2 GPU as a Parallel Stream Processor 16
  - 2.1.3 High Level Programming Languages for GPU 18
- 2.2 Rasterization Based Photorealistic Rendering Methods on GPU 19
- 2.3 GPU Based Ray-tracing 25
  - 2.3.1 Organization of a GPU Ray Tracer 28

3 EACD ACCELERATION STRUCTURE FOR GPU BASED RAY TRACING 33
- 3.1 Accelerated Ray Tracer 35
- 3.2 Accelerated Traversal Kernels 40
  - 3.2.1 DDA Traversal 40
  - 3.2.2 PC Traversal 42
  - 3.2.3 Anisotropic Chessboard Distance Traversal 42
  - 3.2.4 Minimum Branching ACD Traversal Algorithm 43
  - 3.2.5 Extended Anisotropic Chessboard Distance Traversal 47
- 3.3 Construction of The Acceleration Grid 48
  - 3.3.1 Creation of ACD Grid 48
  - 3.3.2 Extending ACD to EACD 49
- 3.4 Results and Discussion 51
  - 3.4.1 Branching vs. Non-Branching Kernel Implementation 52
  - 3.4.2 Fragment Processor Utilization 54
  - 3.4.3 Timing Results 56
  - 3.4.4 Comparison to Other GPU Ray Tracers 58

4 EACD BASED REAL-TIME VOLUME RENDERING ON GPU 69
- 4.1 Volume Ray Casting 70
  - 4.1.1 Homogeneous Space Ray Integration 71
- 4.2 Distance Based Homogeneous Region Leaping 72
- 4.3 EACD Ray Casting 73
  - 4.3.1 Implementation 74
  - 4.3.2 Construction of Homogeneous EACD Regions 76
- 4.4 Results And Discussion 77

5 GPU ACCELERATED STEREOSCOPIC RAYTRACING 81
- 5.1 Previous Work 82
- 5.2 The Ray Tracer 83
- 5.3 Stereoscopic Ray Tracing by Reprojection 84
- 5.4 GPU Based Stereoscopic Reprojection 87
  - 5.4.1 CPU Created Gather Table 89
  - 5.4.2 Two-Pass GPU Gather Table 90
  - 5.4.3 One-Pass GPU Gather Table 91
- 5.5 Results and Discussion 92

5.5.1 Missing Object Problem in Reprojections 92
5.5.2 Reprojections of Reflections and Refractions 93

6 CONCLUSION 96

REFERENCES 99

APPENDIX

A SHADER SOURCE CODES 109
A.1 Grid based ray traversal source codes 109
A.1.1 Cg source code for branching DDA ray traversal 109
A.1.2 Cg source code for multi-pass DDA ray traversal 110
A.1.3 Cg source code for branching PC ray traversal 111
A.1.4 Cg source code for multi-pass PC ray traversal 112
A.1.5 Cg source code for branching ACD/EACD ray traversal 114
A.1.6 Cg source code for multi-pass ACD/EACD ray traversal 115
A.1.7 Cg source code for intersector kernel (for ACD/EACD traversal) . . 117
A.2 Direct volume rendering source code 118
A.2.1 Volume rendering with ACD/EACD 119
A.3 Stereo reprojection source codes 121
A.3.1 CG source code for the stereo reprojection with two-pass gather table generation 121
A.3.2 GLSL source code for the reprojection with one-pass gather table generation 123
VITA 125

LIST OF FIGURES

FIGURES

Figure 1.1 (a) Image from the viewpoint of the camera. (b) Same scene showing some of the light paths passing through the camera. Blue line denotes the path LDE, green denotes LDSDE and the white denotes LSE. . . . 5

Figure 1.2 Ray traced images 7

Figure 1.3 Recursive ray tracing algorithm 8

Figure 1.4 Path traced image of a simple scene 9

Figure 2.1 Standart graphics pipeline for rasterization based rendering. 13

Figure 2.2 GPU graphics pipeline. Green boxes denote user programmable sections of the pipeline. 14

Figure 2.3 Internal structure of a GPU. 16

Figure 2.4 (a) Stream processing, (b) Stream processing on the GPU. Green boxes are kernels. 17

Figure 2.5 Planar reflections. 20

Figure 2.6 Approximate reflections on curved surfaces with cube mapping. 21

Figure 2.7 Transformation steps for projective texturing. 22

Figure 2.8 Projecting texturing to simulate (a) shadows and (b) spotlight region. . 22

Figure 2.9 (a) Shadow mapping. (b) Close-up view of the shadow mapping artifacts. 23

Figure 2.10 (a) Standart OpenGL shading. (b) Projective texturing added to improve spotlight borders. (c) Shadow mapping applied. (d) Reflections added. 26

Figure 2.11 Examples of three different data structures for storing the triangle vertices. (a) All triange vertices are stored in a single texture, in which three consequtive vertices define a triangle. (b) Vertices of the triangles put to different textures separated by corners. (c) Indexed access to vertex data. 29
Figure 2.12 An example organization of the scene database and the acceleration structure. 3D grid texture (acceleration structure) points to triangle lists. Elements of triangle lists are triangle indices which point to triangle coordinates and normals. Each texel of the material id texture points to material information for the corresponding triangle. 30
Figure 2.13 Each screen pixel corresponds to a ray. Origin and direction textures are used to define a ray. 31

Figure 3.1 (a) Ray tracing kernels (b) Ray state transition diagram. 37
Figure 3.2 Input/outputs of the ray tracer kernels. 38
Figure 3.3 DDA traversal loop 41
Figure 3.4 PC traversal loop 43
Figure 3.5 Apex voxels and the intersection lines of a rectangular region depending on the ray direction. 44
Figure 3.6 ACD traversal loop 46
Figure 3.7 EACD traversal loop 47
Figure 3.8 (a) CD acceleration grid has single isotropic distance value per voxel (b) ACD acceleration grid stores a distance value for each direction quadrant. Only the (+x,+y) quadrant values are shown (first and second values are the macro distances along +x and +y axes respectively) (c) EACD acceleration grid stores different distance values for each primary axis. (d),(e) and (f) demonstrates CD, ACD, and EACD traversals of an example ray. EACD traversal significantly reduces the number of traversal steps in this situation. 48

Figure 3.9 (a) ACD kernels for 2D. (b) Creation of the ACD grid for upper-left quadrant. The upper-left ACD kernel of (a) is applied from top-left to bottom-right. Yellow box denotes current voxel. Green frame is the covered region by the kernel. (b) Kernel is overlaid and added with the region. (c) Minimum of the sums is selected as the distance value. Note that, distance of the non-empty voxels (gray boxes) are initialized to 0, while empty voxels are initialized to ∞ prior to kernel application. 50

Figure 3.10 Finding the EACD macro region for the lower left cell. Only (+x,+y) direction quadrant is shown (octant for 3D). Arrows denote the orientation of the distance values (a) is the base ACD grid. (b) and (c) are the axial distance grids along +x and +y directions respectively. (d) is the resulting EACD distances. 51

Figure 3.11 (a) Number of active rays per pass (b) Traversal time per pass (c) Average ray traversal time per pass. 53

Figure 3.12 Ray casted bunny image (a) after 16 trace steps (b) after all trace steps completed. Note that 99.5% of the rays are already terminated in (a) and the most of the image is rendered. Grid resolution is 128x128x128. 55

Figure 3.13 Illustration of the traversal step counts for (a) DDA, (b) PC, (c) ACD and (d) EACD. Brightness of the image is set to 170% and contrast is set to 155% for better visual clarity. 57

Figure 3.14 (a) Bunny (69451 tris) (b) Jacks (24528 tris) (c) Sphereflake (88562 tris) (d) Tree (67454 tris) (e) Lattice (125388 tris). 63

Figure 4.1 ACD and EACD sample points are aligned with the primitive sample points of a primitive ray caster. In the shown case, homogeneous region leaping requires just 2 samples as opposed to 9 samples needed by a primitive ray caster. 73

Figure 4.2 Cg like pseudo-code for the EACD volume ray casting. Note that most of the operations work on vectors. 75

Figure 4.3 Tested volume datasets (a)engine, (b) mrbrain, (c) teapot 77

Figure 4.4 Illustration of loop counts for (a)Ray caster, (b)PC, (c)ACD, (d)EACD. Brightness and contrast is adjusted for visual clarity. Darker regions indicate lower loop counts. 79

Figure 4.5 Teapot images rendered with ACD using different segmentation thresholds. (a)T_o:0.01, T_a:3, (b)T_o:0.02, T_a:3, (c)T_o:0.1, T_a:3, (d)T_o:0.01, T_a:1, (e)T_o:0.01, T_a:10, (f)T_o:0.01, T_a:20 80

Figure 5.1 Kernels for the ray tracer. 83

Figure 5.2 2D illustration of stereoscopic perspective projection. e is eye separation distance. d is focal length. Point A is behind the projection plane, thus projects to positive parallax. C is in front of the projection plane which projects to negative parallax. Since B is on the projection plane, it is projected to zero parallax. 85

Figure 5.3 Reprojection problems. (a) Overlapped pixel and (b,c) bad pixel problems. 86

Figure 5.4 Stereoscopic reprojection algorithm 88

Figure 5.5 Two-Pass GPU based gather table creation. VP denotes data processed by vertex processors, while FP denotes data processed by fragment processors. *oldR* and *R* holds for `prev_right_pixel` and `right_pixel` respectively as calculated in Figure 5.4. 90

Figure 5.6 One-Pass GPU based gather table creation. GP denotes parts of data processed by geometry processors, while FP refers to fragment processing. 91

Figure 5.7 Anaglyph rendering of a dynamically animated scene (a) both eyes separately rendered, (b) right eye reprojected from the left eye's image. 95

LIST OF TABLES

TABLES

Table 3.1 Instruction counts of the traverser kernels. In the cells, first number is the fragment program instruction count (including texture lookups). Second number is the texture lookup count. 52
Table 3.2 Average data dependent branching per ray for the traversal kernels. . . . 60
Table 3.3 Time results in milliseconds of non-branching kernel implementations (grid size is 128x128x128). 60
Table 3.4 Ray casting time results in milliseconds (grid size is 32x32x32). *Frame* is the total rendering time. 61
Table 3.5 Ray casting time results in milliseconds (grid size is 64x64x64). *Frame* is the total rendering time. 61
Table 3.6 Ray casting time results in milliseconds (grid size is 128x128x128). *Frame* is the total rendering time. 62
Table 3.7 Ray tracing time results of the bunny scene in milliseconds (grid size is 128x128x128). 64
Table 3.8 Ray tracing time results of the tree scene in milliseconds (grid size is 128x128x 128). 65
Table 3.9 Ray tracing time results of the jacks scene in milliseconds (grid size is 128x128x128). 66
Table 3.10 Ray tracing time results of lattice scene in milliseconds (grid size is 128x128x128). 67
Table 3.11 Ray tracing time results of the sphereflake scene in milliseconds (grid size is 128x128x128). 68

Table 4.1 Performance results of the ray casting methods. Results are in milliseconds. Last column is the speedup achieved by EACD compared to primitive ray casting. 78

Table 5.1 Frame times of Monoscopic (MONO), Separate stereoscopic (SS), Reprojected with CPU based gather table (RCPU), Reprojected with two pass GPU based gather table (RGPU2), and Reprojected with one-pass GPU based gather table (RGPU1). Visible: Visible surface ray tracing, FullRT: Full ray tracing with maximum ray depth of 3. Times are in milliseconds. 93

Table 5.2 Gather table creation times. 93

CHAPTER 1

INTRODUCTION

The rendering research can be crudely categorized into two classes, photorealistic rendering and real-time rendering. This distinction makes sense because high quality rendering methods are generally offline and computationally very expensive, while real-time rendering solutions are fast but not photorealistic in most cases. The ideal case is to have photorealism in real-time.

Photorealistic rendering is the synthesis of realistic images, where a computer-generated image looks like a snapshot of the real world. The goal is to model a visually correct light distribution in synthetic environments. This involves simulation of sophisticated radiance transfers between the surfaces and the participating media to capture correct inter-reflections between the objects. Although very accurate physical simulation of light is quite demanding and not practical for most applications, a careful approximation of the light behavior is generally enough for creating visually convincing images. Photorealistic rendering involves solving global illumination which computes direct and indirect light distribution in an environment. In order to capture the effects of global illumination, different methods have been devised such as ray tracing, path tracing, radiosity, photon mapping and their variants. These methods can produce very realistic images by solving the global illumination partly or completely. However, since obtaining an adequate image quality can be very time consuming and therefore non-interactive, these methods are generally referred as offline rendering methods.

On the other extent, in real-time rendering, images are generated immediately. Each frame is rendered while a user watches or performs fast interaction in real-time. This gives a sense of motion in an interactive manner which is referred to as motion realism. A system should preferably display more than 30 Hz (30 frames per second) for comfortable sense of motion and a frame rate of at least 60 Hz should be targeted for good immersion. In order to obtain such frame rates, a real-time rendering system mostly takes the advantage of dedicated graphics hardware. Fast and realistic real-time rendering is at the heart of a virtual reality system, which is widely used for training and simulations, such as pilot training with visual flight simulation or driver education through vehicle simulation, architectural walkthroughs, product design, military combat simulations, complicated surgery training, and so on.

The ultimate goal of real-time rendering is to combine photorealism and motion realism together. Since solving global illumination is slow, traditional real-time rendering relies on simplified light interaction models, therefore generates simplified reality. In most cases the simplifications will make the virtual world look computer-generated, unrealistic and plastic-like, making it harder for a user to project herself into the virtual world. Even small inaccuracies such as missing shadows or reflections may distract the human visual system and destroy the sense of reality. Furthermore, effects like reflections and shadows are critical visual cues. The human mind intuitively uses these cues to estimate position, size, distance and shape of objects. Ignoring these effects in an interactive real-time rendering system makes it harder for a user to navigate and interact with the environment which results in a less realistic experience.

There is a lot of research in the real-time photorealistic image synthesis. A part of the research is focused on making use of the hardware resources as efficient as possible. It is shown that, by using specialized data structures, algorithms and memory access patterns in accordance with the underlying hardware, the rendering can be accelerated by an order of magnitude. Therefore, besides using algorithmically optimized acceleration structures, traversal algorithms, light transport computations, sampling techniques, etc.; it's also very important to develop specific methods for making efficient use of the hardware.

Graphics processing units (GPU) found in the graphics cards are used to accelerate graphics drawing operations in real-time rendering tasks. Almost all virtual reality systems, games and other applications which require real-time rendering employ GPUs for fast image generation. Early GPUs were generally only capable of scan converting and shading triangles, where shading is based on simple local illumination models. The graphics pipeline of such hardware is also called fixed function pipeline, since they are not programmable and process the geometry with some fixed chain of operations. Modern GPUs, on the other hand, are very flexible and programmable processors. Graphics rendering heavily depends on parallel arithmetic calculations. Therefore, graphic hardwares are designed accordingly. GPUs are massively parallel processors and very powerful in terms of arithmetic operations. There are hundreds of parallel processing units on a modern GPU. As of today, consumer level graphics cards are capable of performing around 1 TFLOPS (trillions of floating point operations per second), which is far beyond the reach of a CPU. Enormous arithmetical power and increasing flexibility hold great opportunities for using GPUs in non-rasterization based complex photorealistic rendering techniques including ray tracing.

Based on these observations, the motivation, goals and the contributions of this thesis are given in the next section.

1.1 Goals and Contributions

Particle or ray tracing is at the core of many photorealistic rendering techniques. Therefore in this work we focused on accelerating ray tracing which can be used as a tool for fast photorealistic rendering. Increasing flexibility and arithmetic power of the GPUs found in contemporary graphics cards make them good candidates to be used in acceleration of computationally expensive ray tracing. The aim of this work is to investigate efficient methods for GPU based ray tracing in order to achieve real-time rendering of photorealistic images. In order to reach this goal, the architectural constraints and powerful aspects of the GPU should be taken into account. In other terms, data structures and algorithms should be revised accordingly or new techniques conforming to the GPU programming model should be developed.

Our first goal is to accelerate ray tracing using programmable graphics pipelines by developing GPU friendly techniques. To be more specific, we studied distance transformation grids based acceleration structures. We aim interactive frame rates for at least moderately complex scenes. As the first contribution, we devised a novel acceleration structure based on distance transformation grids called as the extended anisotropic chessboard distance (EACD), and developed an efficient GPU friendly ray traversal algorithm running on this acceleration structure. GPU friendly implementations of some well-known grid based acceleration techniques and their comparison were also given in the thesis as other contributions.

Real-time volume rendering has many uses in medicine, geography, and many other fields that require fast visualization of volumetric data. Our second goal is to devise a GPU based real-time volume ray caster. Connected to this goal, we extended the EACD ray traversal algorithm for real-time volume rendering and employed homogeneous region encoding to visualize data in real-time as the second main contribution.

The third goal is to develop a fast GPU based stereoscopic ray tracing technique to be used in conjunction with GPU based ray tracers and volume ray casters. Therefore, as the third contribution, we devised GPU accelerated stereoscopic reprojection methods.

Ultimately as an outcome of this work and other works similar to this one, we believe that with the methods making efficient use of the GPU, the gap between high quality photorealistic rendering and lower quality real-time rendering will get closer.

1.2 Photorealistic Rendering

In this section, a quick overview of some widely used photorealistic rendering methods are given. The purpose is to make the reader aware of the well-known techniques. Among the techniques, ray tracing is explained in greater detail since this thesis is focused on fast ray tracing with GPUs.

1.2.1 Global Illumination

Global illumination methods synthesize photorealistic images by simulating the light behavior. Being different from local shading techniques, global illumination takes the light interaction between all surfaces of an environment into account. It solves light transportation and thus can simulate reflections, refractions, shadows, caustics and other related physical phenomena. Light transfer can be calculated mathematically using the well known rendering equation (1.1) [49]. Although the rendering equation has some shortcomings such as omitting the participating media or sub-surface scattering, it is considered as the mathematical foundation for the global illumination algorithms [22].

$$L_o(x' \to x) = L_e(x' \to x) + \int_S f_r(x'' \to x' \to x) L_i(x'' \to x') V(x', x'') G(x', x'') dA'' \quad (1.1)$$

The rendering equation is a Fredholm integral of second type. It states that the outgoing radiance L_o from the point x' to the x is the sum of emitted radiance L_e and the integral over the set of all surface points S of incoming radiances L_i from the point x'' times the surface BRDF (Bidirectional Reflectance Distribution Function) f_r times the visibility V and the geometry G terms. The integral part calculates the reflected radiance. Note that this is a recursive equation: in order to find out the reflected radiance it's required to solve the incoming radiance which is computed using the same equation.

When talking about global illumination and thus the light transport, it is useful to describe the type of surface reflections along the path. Heckbert introduced a well known

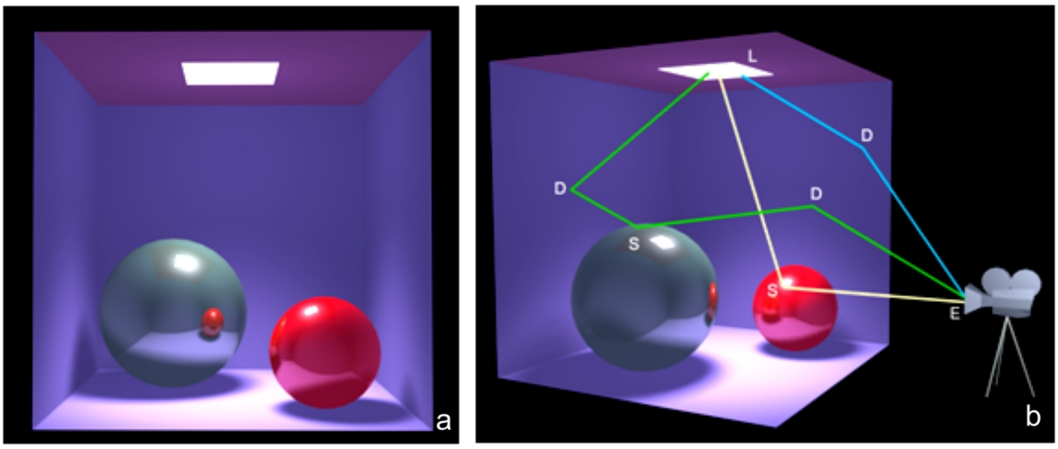


Figure 1.1: (a) Image from the viewpoint of the camera. (b) Same scene showing some of the light paths passing through the camera. Blue line denotes the path LDE, green denotes LDSDE and the white denotes LSE.

notation to describe a light path with the points it passes through [41]. In his notation a light can pass through the following points:

L point on a light source

E the eye

D point where ray is reflected diffusely

S point where ray is reflected specularly

Important light paths can be described using this notation. Figure 1.1 demonstrates some of the possible light paths. For instance, a path starting from the light source and reflected diffusely twice and reaching the eye is denoted by LDDE. Any number of diffuse and specular reflections may occur along the path. To simplify the notation regular expressions can be used, where: (k)+ means one or more, (k)* means zero or more, (k)? means zero or one of k event and (k|k') means either a k or k' event.

Ray tracing, radiosity, photon mapping and path tracing are well known and widely used photorealistic rendering techniques. Not all of these techniques provide full global illumination, since each can handle certain types of light interactions. For instance classic ray tracing can simulate specular reflections from mirror like surfaces but can not handle diffuse reflections. On the other hand, radiosity can render only diffuse reflections but can not cope with the specularly reflected lights. Similarly photon mapping is not efficient in rendering of specular reflections. Sometimes these techniques are combined to have a more complete solution.

Radiosity methods can compute light transfer between diffuse surfaces [33]. It assumes that all surfaces are perfect Lambertian (diffuse) and solves the radiosity values of all surfaces. It is a geometry based method, which means that the radiosity solution is calculated once, independent from the eye location. Calculated radiosity values are then used for rendering the images (as long as lights and surfaces do not move relative to each other). Rendering is done by interpolating the calculated radiosity values with a simple shading method such as Gouraud (taking the values as the intensity or color). Radiosity considers LD*E light paths. Specular reflections can not be calculated efficiently with radiosity. Therefore it can be used in conjunction with ray tracing for specular surfaces.

Photon mapping is another global illumination method such that it can be used to simulate caustics, translucency, participating media, soft shadows, diffuse reflections [47]. It is a two pass method. In the first pass, photons emitted from the light sources are traced. This step is very similar to ray tracing essentially, and same acceleration techniques can be used for tracing photons. When a photon hits a surface, it's intersection position, energy and incoming direction is stored in a spatial data structure called photon map. Then, the photon is reflected to a direction computed according to the BRDF of the surface. In the second pass, the photon map is used to estimate the density of the photons at surface locations by using nearest N photons. This gives the estimation of the illumination on the location. By using the calculated photon density, radiance of the point is estimated. The rendering equation can be divided into emission, direct illumination, specular reflection, caustics and indirect illumination terms. Photon map can be used as the auxiliary data structure for other techniques, or can be used to calculate some terms of the rendering equation. For instance, in a path tracer it can be used for importance sampling. Since direct illumination and specular or glossy reflections require too many photons, those terms can be rendered by ray tracing in another pass.

1.2.2 Ray Tracing

Ray tracing is a well known photorealistic image synthesis method [5, 97]. It can accurately simulate reflections, refractions, shadows and other various light phenomena by tracing reflected or emitted ray trajectories. High quality and believable images of virtual environments can be rendered using ray tracing as demonstrated in Figure 1.2. In Whitted style ray tracing (also known as the classic ray tracing) [97], eye rays are generated and traced through the scene. Upon a surface hit, shadow rays for each light source and reflection/refraction rays are generated depending on the material properties of the surface. The material properties

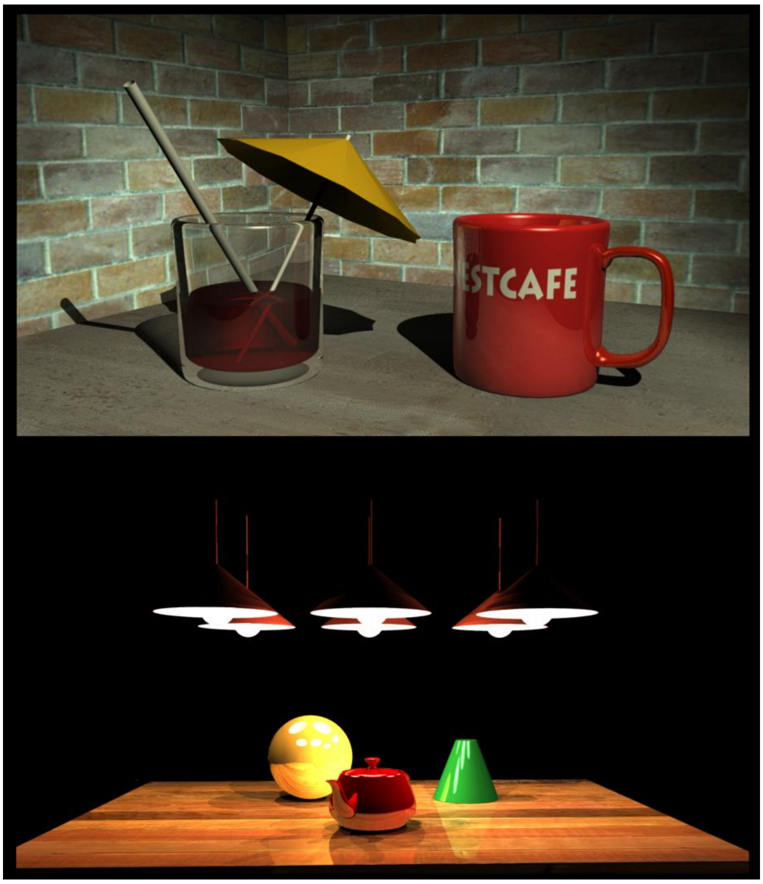


Figure 1.2: Ray traced images

include surface reflectivity and refractivity information. Using the Heckbert's light transport notation ray tracing can follow LD?S*E paths. Rays are typically traced beginning from the eye. For each pixel, a ray from the eye to the pixel location is fired. Therefore the path can be translated as follows: rays coming from the eye can be reflected zero or more times specularly and afterward can be reflected zero or once diffusely through the light source. In other words, classic ray tracing works backwards from the eye to the light source. It makes sense because there are infinite number of light bounces and paths in the real world but only the paths reaching to the eye contribute to the synthesized image. Ray tracing is view dependent, which means the image has to be rendered from scratch if the view position or direction is changed.

The C like pseudo code of the classic recursive ray tracing algorithm is given in Figure 1.3. As seen in the figure, ray tracing is a recursive technique. It follows specularly reflected and refracted ray paths and builds a ray tree. At each bounce the contribution of the surface to the image is computed and accumulated to the color of the corresponding screen pixel. If a

```
rayTrace()
{
    for (each pixel)
    {
        ray = fireRay(eye_position, getPixelPosition(pixel));
        pixel_color = trace(ray);
    }
}

color trace(ray)
{
    color = black;
    nearest_object = NULL;  nearest_distance = BIG_VALUE;
    for (each object) //find the nearest intersection
    {
        point = insersect(ray,object);
        if (distance(getOrigin(ray),point) < nearest_distance)
        {
            nearest_object = object;
            nearest_point = point;
            nearest_normal = computeNormal(object,point);
        }
    }
    if (nearest_object)
        color = shade(ray,nearest_point,nearest_normal,getMaterial(nearest_object));
    return color;
}

color shade(ray, point, normal, material)
{
    color = black;
    for (each light) // local illumination & shadowing
    {
        shadow_ray = fireRay(point,getPosition(light));
        point_is_lit = true;
        for (each object)
        {
            intersection_point = insersect(shadow_ray,object);
            if (distance(point,intersection_point)<distance(point,getPosition(light))
                {point_is_lit = false; break;}
        }
        if (point_is_lit)
            color = color + localIllumination(light,ray,point,normal,material);
    }
    if (isReflective(material)) // reflections
    {
        reflection_ray = reflectRay(ray,point,normal);
        color = color + getReflectivity(material) * trace(reflection_ray);
    }
    if (isTransparent(material)) // refractions
    {
        refraction_ray = refractRay(ray,point,normal,getIndexOfRefraction(material));
        color = color + getRefractivity(material) * trace(refraction_ray);
    }
    return color;
}
```

Figure 1.3: Recursive ray tracing algorithm

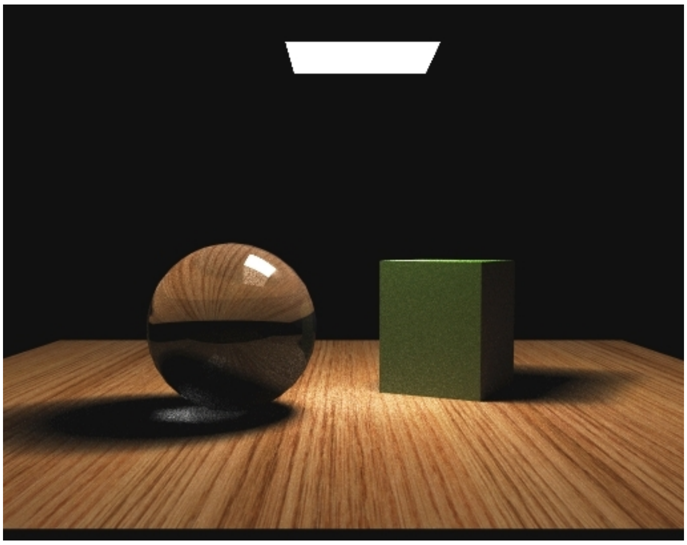


Figure 1.4: Path traced image of a simple scene

ray hits a diffuse surface, only direct lighting is accounted; otherwise new rays are spawned to trace the reflection and refraction paths. When a ray hits a diffuse surface, misses all surfaces, or if the recursion depth is greater than a threshold the ray is terminated. Other termination criteria are also possible.

In the same paper that Kajiya introduced the rendering equation [49], he also proposed a method called path tracing to solve the equation. Path tracing makes use of ray tracing to compute the light transportation. It can simulate all possible light paths which is described as L(S|D)*E. That means it can correctly render diffuse and specular reflections, refractions, caustics, soft shadows, glossiness and other physical events. Some of these effects are visible on the path-traced image of a very simple scene as shown in Figure 1.4.

Because the rendering equation involves complex recursive integrals, analytic solution is not possible except for the simplest scenes. Instead, path tracing uses Monte Carlo integration methods to compute the radiance values falling onto the pixels. Simply stated, Monte Carlo methods try to estimate the solution using probability theory and stochastic sampling. As a result, the path tracing relies on high number of ray paths to estimate the radiance. Since the number of paths to be traced are infinite, path-tracing tends to generate high frequency noise in the image due to the undersampling of ray paths. Depending on the surface and scene characteristics, typically 100 to 10000 paths per pixel (or even more) should be traced for satisfactory results. In Figure 1.4 although 400 rays are fired for each pixel, there is still visible noise. Especially diffuse surfaces only and smooth light variations can be rendered using less number of paths per pixel. Different sampling techniques (stratified, quasi random, metropolis etc.) have been proposed in order to reduce variance for faster convergence to the solution.

Path tracing employs ray tracing for stochastic sampling of light paths. Since our work accelerates ray tracing on the GPU, it can also be used for the acceleration of path tracing.

1.3 Outline

The rest of the thesis is organized as follows: GPU based photorealistic rendering techniques are discussed in Chapter 2. Background information about programmable graphics architectures and rasterization based realistic rendering research on the GPU are given in this chapter. Additionally, basic organization of a GPU based ray tracer is introduced in the chapter.

Our EACD based ray tracing acceleration technique is given in Chapter 3. Additionally, we devised efficient GPU specific versions of previously known grid based acceleration methods. Our technique is compared to these known methods experimentally. We also compare our work to the other GPU ray tracers.

The following chapter is an extension of the work introduced in Chapter 3. Real-time volume rendering by using EACD is explained in this chapter. Our method uses distance transformations and homogeneous region encoding to accelerate volume ray casting. We tested the renderer by using some of the well known volume dataset and compared the results to other grid based volume ray casters.

In Chapter 5 GPU based stereoscopic ray tracing is introduced. The stereoscopic image is generated by reprojecting the image of one eye to the other one. We devised a GPU friendly solution to the reprojection problem. The solution is especially well suited to GPU based ray tracers, since no CPU intervention is necessary during the reprojections. We tested the method with our GPU based real-time ray tracer in a dynamically animated scene.

In the last section conclusions and possible future work about GPU based ray tracing is given.

CHAPTER 2

GPU BASED REAL-TIME PHOTOREALISM

Early GPUs could only scan convert simple primitives with some limited adjustable settings. However, during the last decade they evolved into programmable parallel processors specifically tailored for rasterization based graphics rendering. The increased speed and programmability of modern GPUs made it possible to render more realistic images of complex scenes in real-time. GPU based rendering research has been shifted from fast rasterization of triangles to a more physically accurate simulation of light and increasing the level of realism. Nowadays, rendering photorealistic images in interactive rates with graphics hardware attracts great deal of attention. As mentioned in the first chapter, realistic rendering is very important in real-time applications such as virtual reality, visual simulations, training, video games and other fields that require accurate simulation of real world phenomena. GPU based photorealistic rendering techniques are briefly reviewed in this chapter. While doing this we did not make a comprehensive review. Instead, we mentioned some basic techniques in order to give a background on the topic.

Besides the rasterization, GPUs are especially good at arithmetically intense operations. As of date, graphics boards are able to perform arithmetical computations at the speed of 1 TFLOPS (trillions of floating point operations per second) [66]. The processing units inside the GPUs are light weight with limited capabilities, and are optimized for graphics processing. However, the simplicity of the processing units and the parallel nature of graphics rendering makes it possible to use many of these units concurrently. Consequently, modern GPUs are massively parallel processors. They can handle thousands of threads very efficiently. Moreover, due to the architectural differences from the CPU, GPU performance has been typically doubled every six to twelve months. This is a much steep acceleration rate than Moore‘s law of eighteen months which is still valid for CPUs.

On the other hand, GPU based general purpose processing has some important restrictions. Limited input/output registers, inefficient random memory access, inability to write dynamic memory locations, lack of a sophisticated branch prediction units, limited recursion capabilities are some of the major difficulties. In order to utilize computational units of a GPU as much as possible, all of these limitations should be considered and specific methods conforming to the GPU processing model should be developed. The reader may refer to [34, 71, 20] for more details on GPU programming.

There are also some additional restrictions and overheads imposed by the graphics APIs. The communication between CPU and GPU, or in other words CPU intervention, should be avoided as much as possible. Excessive API calls and (relatively) slow data transfers between the host system and the graphics card results in under utilization of the GPU.

The rest of this chapter is divided into three sections. The graphics pipeline and programmable graphics architectures are overviewed in the first section. The next section describes interactive realistic rendering by means of traditional GPU rasterization. Basic constructs and previous work on GPU based ray tracing is given in the last section.

2.1 The Rendering Pipeline

A 3D virtual environment, also referred to as scene, basically consists of object, light and camera descriptions. There are many ways to model objects in 3D. One popular way is to use triangle meshes to represent object surfaces. Other methods include volumetric modeling, implicit modeling, splines, blobs etc. Refer to [27] for more on modeling objects for graphics rendering. All other representations should be converted to triangles, lines or points before rasterized by the GPU. Triangle meshes are easy to represent and relatively cheap to render. Triangle meshes consist of vertex data and the connectivity information which is used to assemble triangles from the vertex stream. Each vertex is associated with a position attribute. There may be other attributes that can be associated with the vertex such as normals, color, texture coordinates etc. Primitives can be associated with shading related information such as materials and textures. This information is used to determine the final color of the surface by employing a shading method. Shading process makes use of textures, lights, fog, basic surface colors, opacity and other information to determine the ultimate color of the surface pixels.

The graphics pipeline is a chain of processes which takes the scene data from the appli-

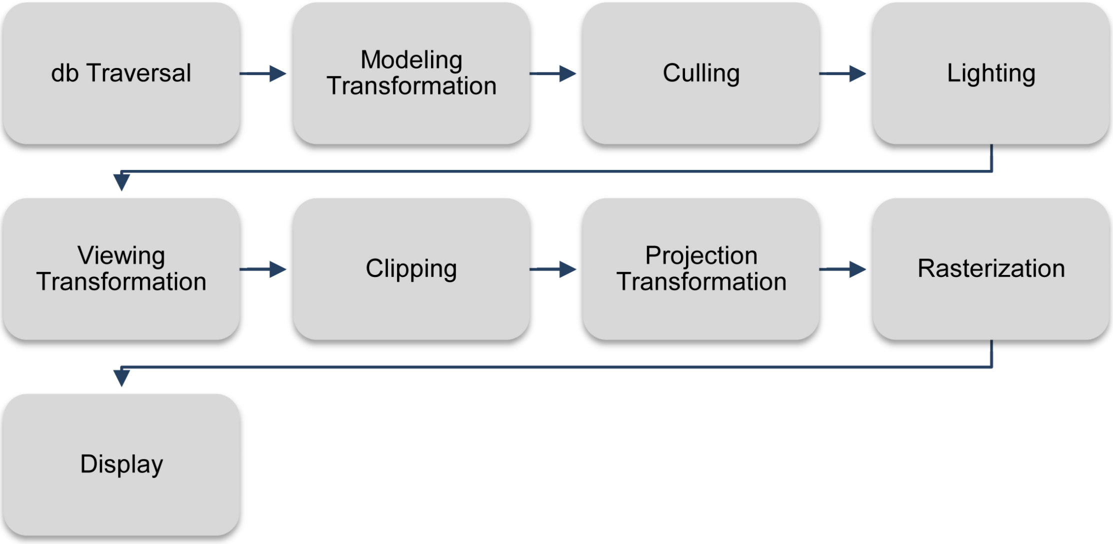


Figure 2.1: Standart graphics pipeline for rasterization based rendering.

cation at the front, and produces the rendered image at the end. The processes inside the pipeline may be different for different rendering techniques. For example ray tracing has a very different set of operations performed in the pipeline from that of a z-buffered Gouraud shaded rendering architecture; such as, ray tracing does not have a projection transformation step or a frustum clipping phase.

The traditional pipeline for a rasterization based rendering is given in Figure 2.1. Each triangle sent to the pipeline is exposed to a series of computations. At the end of the pipeline the triangle is either rasterized or rejected (and not rasterized). In the figure, *dbTraversal* phase consists of feeding the pipeline with triangles to be rendered. Vertices are stored in a local coordinate system for each object. Modeling transformation transforms the object geometry from the local coordinates to the world coordinates. Culling phase eliminates unseen triangles. Lighting phase determines the color of vertices by using the shading parameters with an illumination model. This way of computing colors of the vertices is called per vertex lighting. When the objects are modeled coarsely, visual artifacts such as mach banding may occur in per vertex lighting technique. The next phase of the pipeline is viewing transformation, which transforms the scene in world coordinates to the viewing coordinates of the observer. Clipping is done against the view frustum in the next phase. Projection transformation transforms 3D geometry into 2D by projecting the vertex coordinates to the view plane. For z-buffering, vertex depth information (z coordinate) is retained during the transformation. In the rasterization phase, triangles are scan converted which is the operation of

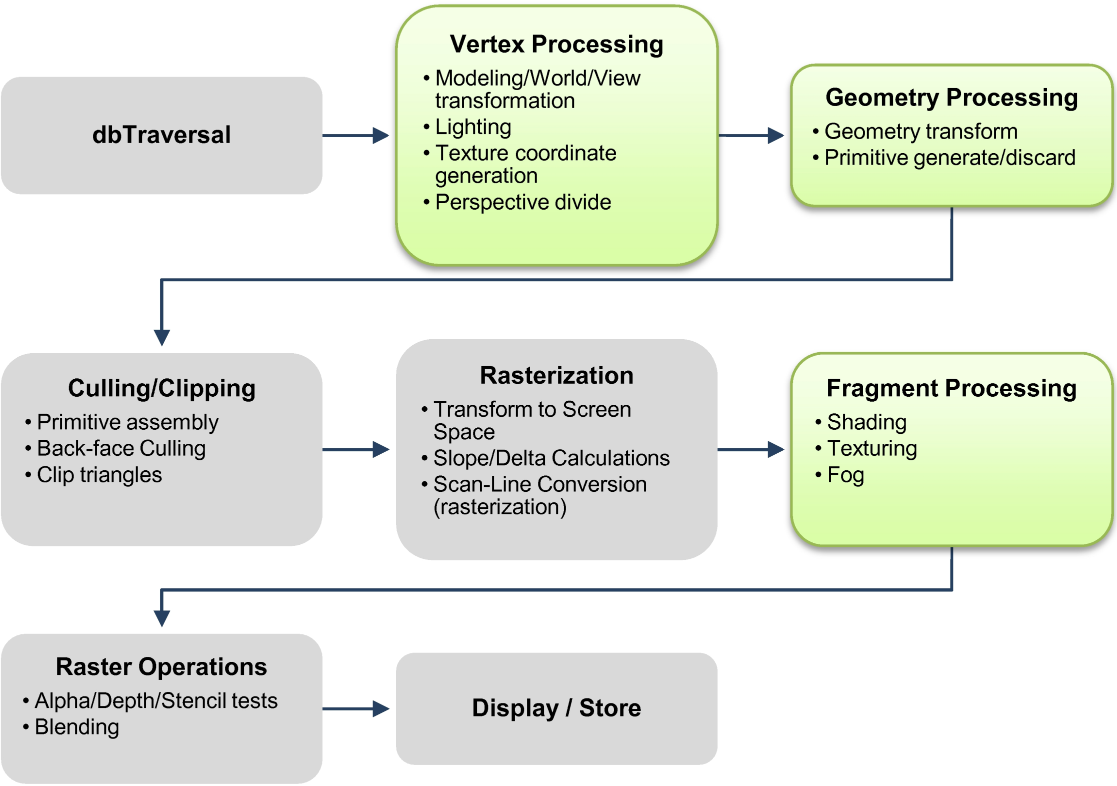


Figure 2.2: GPU graphics pipeline. Green boxes denote user programmable sections of the pipeline.

computing the screen pixels of the projected triangles. The output of the pipeline is rendered pixels. Rendered pixels are then sent to frame buffer for displaying, or to an offscreen buffer in order to be used later.

2.1.1 GPU Pipelines

Figure 2.2 shows how graphics processors implement the traditional graphics pipeline. As you can see, it is very similar to the traditional pipeline. Typically multiple pipelines run in parallel on a GPU, so that multiple pixels can be rendered simultaneously.

In a fixed function pipeline (i.e. not programmable) only way of controlling the operations is to set some states or registers before sending triangles. The state adjustments make some certain decisions along the pipeline. For example, if triangles are to be lit, lighting computation state should be enabled, the light position and other parameters should be initialized, and a couple of material attributes should be set prior to the rendering. Generally, material attributes can not be anything more than diffuse, specular, emission, ambient colors, transparency, glossiness and a combination of image textures. Moreover, shading type is generally limited to flat or Gouraud (interpolated vertex colors). In order to overcome some limitations imposed by the fixed function pipelines to an extent, multi-pass rendering methods can be used. Multi-texturing capabilities (mapping two or more textures at once with a user defined blending function) and register combiners found in graphics hardware further remedies the problem. Early GPUs had fixed function pipelines only.

In order to overcome the limitations imposed by fixed function operations, some parts of the pipeline were replaced by programmable processing units. This kind of GPU pipeline is known as the programmable pipeline. Modern GPU architectures have programmable pipelines. In Figure 2.2 green parts of the pipeline denote the programmable sections. The operations listed inside the boxes, which were performed by the fixed function pipeline previously, can be programmed by the user. This makes it possible to employ different illumination models, texturing, shading techniques, deformations etc. By the addition of geometry processors on the latest GPU architectures, it is also possible to discard geometric primitives sent to the pipeline or to create completely new primitives on the fly.

Note that, "fragment shader", "fragment program" or "pixel shader" are just different names of the same thing; programs executed by fragment processing units, and will be used interchangeably. A fragment is a rendered pixel or sub-pixel that has the potential of modifying the value of the corresponding image pixel. More than one fragments can contribute to the same pixel. Pixels represent the final values as seen on the screen. Similarly, the terms "vertex program"/"vertex shader" and "geometry program"/"geometry shader" points to the programs executed by vertex and geometry processing units respectively.

Figure 2.3 shows the hardware structures implementing the pipeline. Green boxes handle the corresponding programmable parts of the pipeline. Vertex processors typically transform vertex information and send the processed vertices through the pipeline, while geometry processors are like vertex processors; in addition they can generate or discard geometric primitives. Along the pipeline the geometry is scan converted to fragments by rasterization. All of the processors can read data from textures (1, 2 or 3D images). There are hundreds of vertex, geometry and fragment processing units on a modern GPU. Vertex and fragment processors were different processors with different capabilities up the latest generation (and there were no geometry processors). On the latest generations, all of the processors have identical capabilities, and they are assigned to the pipeline stages on demand. This is known as unified shader architecture. With this architecture, there is a departure from the pipeline organization on the hardware level. However, since graphics rendering requires

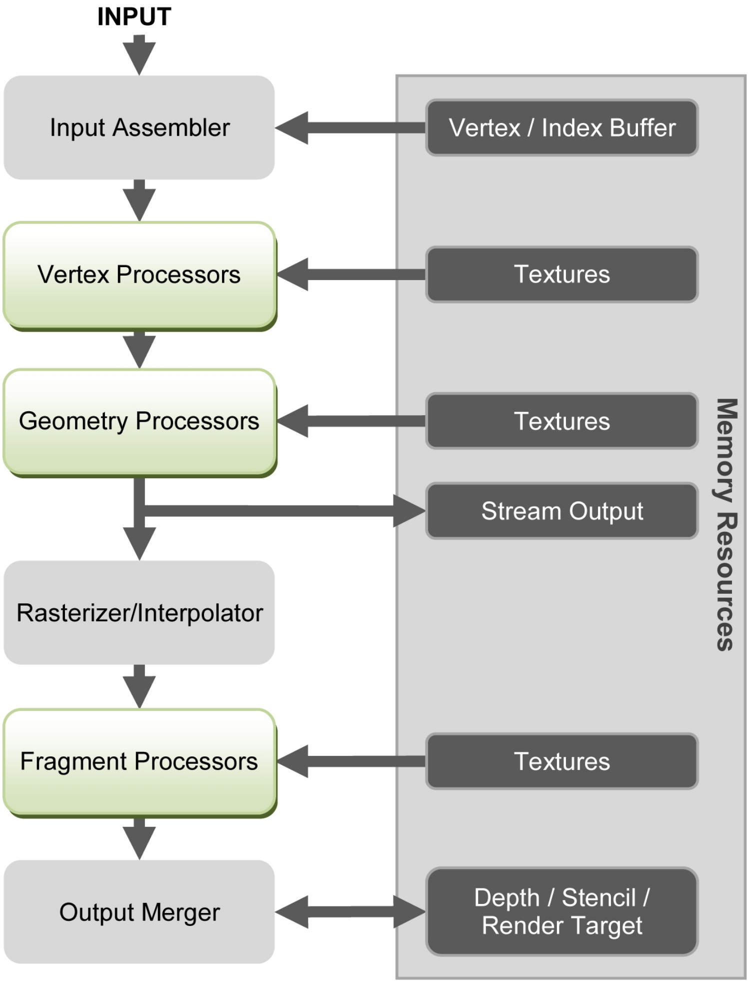


Figure 2.3: Internal structure of a GPU.

certain operations to be done in certain order, the pipelined organization is still provided by the drivers. For general purpose computations, the pipeline organization may be bypassed if required.

2.1.2 GPU as a Parallel Stream Processor

Inside the GPU, there are many streaming processors running in parallel. Therefore, GPUs are an implementation of parallel stream processing. A stream can be described as an ordered set of data. Streams are processed by functions called as kernels, generally by executing a series of instructions for each element in sequence. Kernels accept a number of input streams and generate one or more output streams. A number of kernels can be chained to accomplish complex operations. Stream processing favors arithmetic intensity, data parallelism, high compute-to-bandwidth ratio, data locality and infrequent global data access [70]. Especially

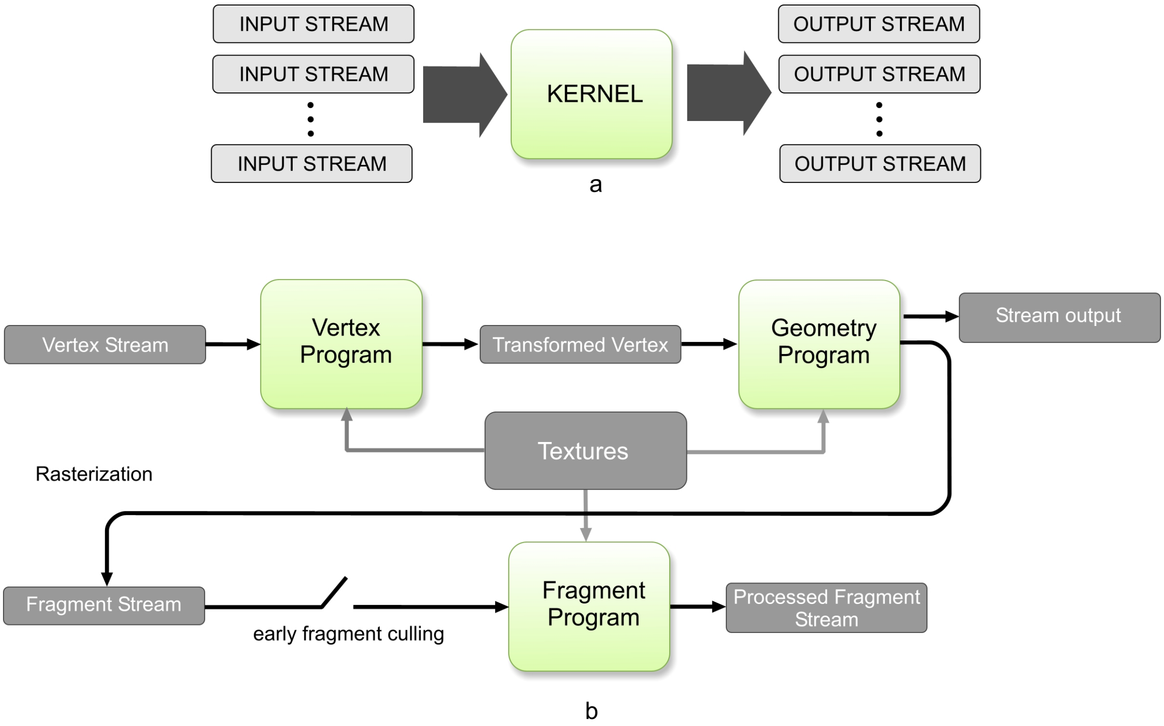


Figure 2.4: (a) Stream processing, (b) Stream processing on the GPU. Green boxes are kernels.

media and graphics applications are good candidates for efficient stream processing.

Figure 2.4 depicts the rendering pipeline as stream processing elements. In the context of graphics rendering, elements of the input stream may include of vertex attributes (position, normal, color, etc.), texture elements (i.e. texels), fragments (pixel candidates) and the output is the stream of geometry or fragments. The streaming processors inside the GPU execute vertex, geometry and fragment programs (kernels). These kernels can process multiple streams and output multiple streams. The output streams of the geometry program can be circulated to feed the pipeline on the subsequent passes. Modern GPUs allow early fragment culling based on the outcome of the stencil or depth tests. Leveraging early fragment culling facilitates killing fragments before reaching the fragment processor, and thus saves processing power. Therefore, early fragment culling can be used for efficient computational masking. Later on the pipeline, fragment programs process the rasterized fragment stream and send the processed fragments to the raster operations units for the final composition. Fragment processors can write output values to textures (render to texture), so that the computed values can be used as input to the subsequent kernel passes. However, an output value can only be written to a single memory location (a pixel) defined by the position of the projected geometry, and it is not possible to change this location within the fragment program. This restriction makes it hard to implement efficient scattering operations on the GPU.

Vertex, geometry and fragment kernels are executed in sequence and once on a rendering pass. For example, in order to process a stream by two different fragment programs, it should be rendered once with the first program and then the output of the first pass should be processed by the second program.

Despite the difficulties of GPU based general purpose programming, carefully designed algorithms can greatly benefit from the power of the hardware. Future GPUs are expected to evolve into more CPU like devices with more generic instruction sets, better addressing capabilities, richer registers and data types while keeping their high speed parallel computing advantages.

2.1.3 High Level Programming Languages for GPU

GPUs can be programmed using the shading language provided by the graphics API such as Direct3D [14] or OpenGL [35]. In addition to low level and high level shading languages provided by these APIs, there are several other GPU languages and libraries for programming. For most of the time, it is advisable to use high level languages since compliers can optimize the generated code according to the target GPU architecture. In this section, some of the well known GPU programming technologies are briefly described.

Mark et al. [60] explain a C like high level programming language called Cg (C for graphics). Cg supports both of the major 3D graphics APIs (OpenGL and Direct3D). Independent from Cg, these APIs have their own high level shading languages also. Direct3D's high level shading language (HLSL) [14] and OpenGL shading language (GLSL) [80] are very similar to each other and indeed have the same roots with Cg. A Cg tutorial in can be found in [26]. In this work, we used Cg for the majority of GPU programming while some minor code sections were implemented with GLSL.

Another GPU programming language worth attention is Brook for GPU[53]. It is a compiler and runtime implementation of the Brook stream program language for modern graphics hardware. Brook for GPU's main goal is to provide high level programming facilities for general purpose computation on the GPU. Brook is an extension to standard ANSI C. It incorporates the ideas of data parallel and arithmetic intensive computation into a familiar and efficient language. Brook3D back end compiler supports DirectX and OpenGL shader language profiles to create GPU programs. Brook runtime library takes the responsibility of managing buffers, textures and other daunting tasks of GPU based programming. A similar programming language is Sh [46]. Sh is a high-level meta programming language. It acts as a language embedded in C++.

CUDA is a computing architecture, which enables more direct access to the NVIDIA GPUs [67]. CUDA provides a new programming interface and uses standard C language for software development. It has a standalone runtime driver which inter-operates with Direct3D and OpenGL graphics drivers. It has native multi GPU support to be able to instrument more than one GPUs plugged in a computer system. CUDA technology can process thousands of threads concurrently.

AMD's CTM (Close To the Metal) Device [18] consists of an array of parallel floating point processors and supporting components to feed the array. It facilitates low level access to the AMD GPUs. Similar to CUDA, the motivation is to utilize GPU power as much as possible without API overheads and difficulties of the shader oriented pipeline programming.

2.2 Rasterization Based Photorealistic Rendering Methods on GPU

Real-time realistic rendering on the GPU is one of the hot research areas in computer graphics. As an early work, Diefenbach described some methods to utilize traditional graphics pipeline to generate global illumination effects in his PhD thesis [21]. Nielsen [64] explained additional and improved methods. Great deal of related real-time rendering research is summarized in [3]. There are numerous work in the literature about rasterization based realistic rendering. In the rest of this section, we summarize several basic and popular ones among these techniques. Note that, none of these techniques can handle global illumination effects to the full extent. Instead, each work provides solutions to a small subset of global illumination effects such as reflections, refractions, caustics or shadows individually. Several of these methods should be combined to obtain a realistic image.

Reflections/Refractions

Planar reflections can be rendered with rasterization in real-time as decribed in [21, 61]. For this purpose, it is possible to use environment maps or to render a mirrored duplicate of the scene. Scene duplication is adequate if the reflector surface is perfectly flat. It requires one rendering pass for all surfaces those lay on the same reflection plane. In order to create mirrored scene duplicate, the camera view matrix is multiplied by a reflection

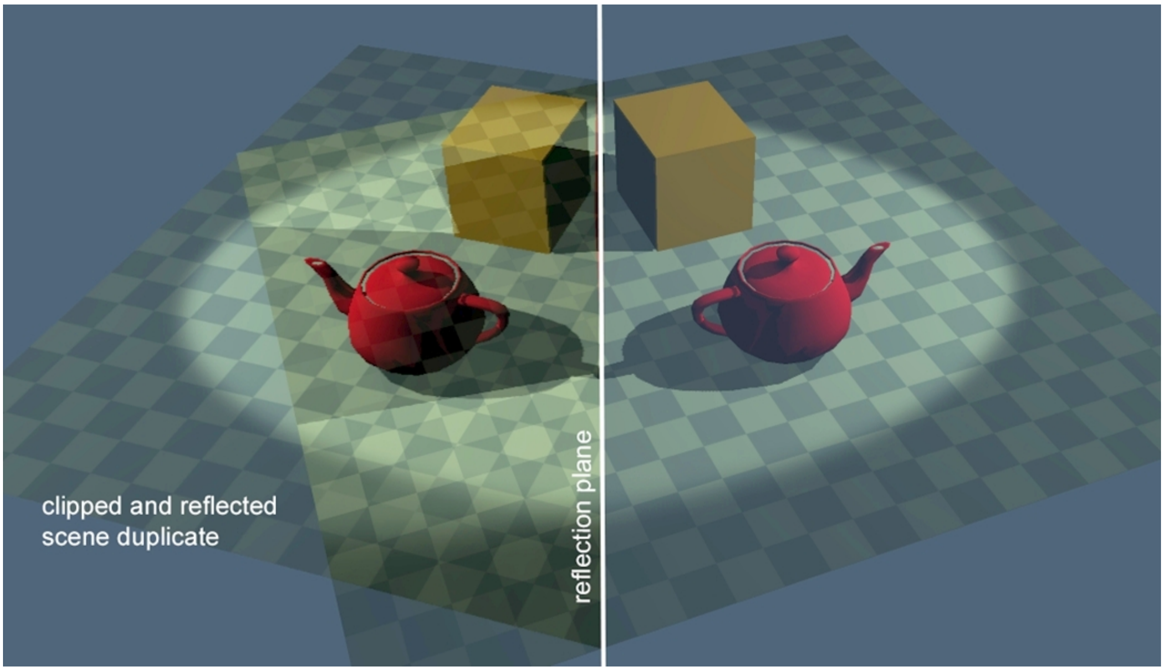


Figure 2.5: Planar reflections.

matrix. For correct lighting, the light positions and directions should also be reflected. The view transformation matrix M for the mirroring is prepared as follows:

$$F = R(n, (0,0,1))T(-c) \tag{2.1}$$
$$M = FS(1,1,-1)F^{-1}$$

In 2.1, c is a point on the reflector surface and n is the normal of the reflector surface, T is the translation matrix, S is the scaling matrix, and R is the rotation matrix that maps the surface normal to Z axis (assuming row major matrices and column vectors). The scene is mirrored by this reflection matrix and clipped against the plane of the reflector surface. This is required otherwise the objects behind the reflector surface are drawn incorrectly in front of the reflector when the mirrored duplicate of the scene is rendered. The reflected scene is either rendered on top of the original reflector surface using stencil buffers or can be rendered to an off screen reflection buffer and then blended with the reflector surface. The rendered image of the reflected scene is then mapped to the reflector surface using projective texturing. It is also possible to project the reflector surface coordinates to the screen space and calculate the corresponding texture coordinates. In this case, the surface is rendered as a 2D polygon with 2D texture coordinates in screen space and the rendered image is pasted onto this polygon. Planar refractions basically use the same method; only the view

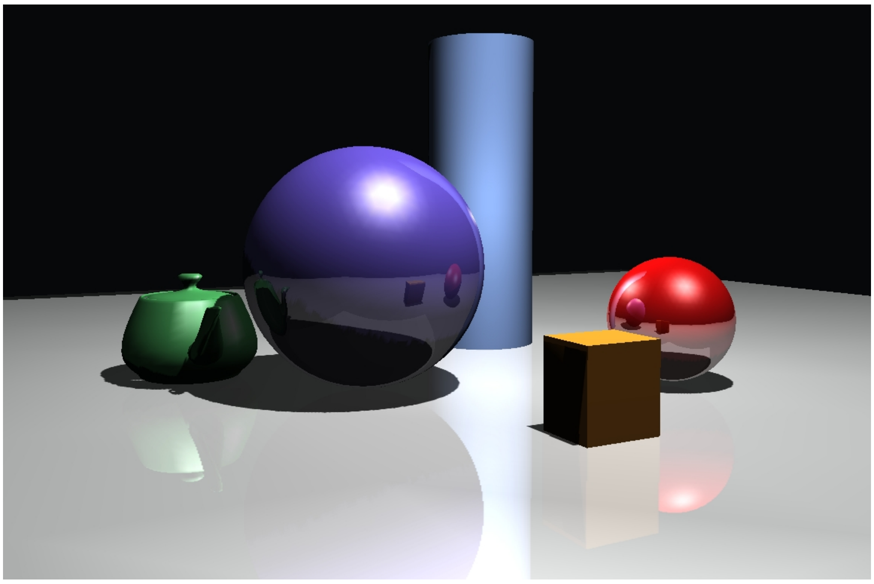


Figure 2.6: Approximate reflections on curved surfaces with cube mapping.

transformation matrix is altered according to the refracted scene.

It is also possible to render approximate curved reflections by rasterization based rendering [7, 69]. For curved reflections and refractions, typically the environment surrounding the reflective surface is rendered to the faces of a cube map texture. Then the cube texture is used as the enviroment map. A simple scene using this technique is shown in Figure 2.6.

Projective Texturing

Projective texturing is not a light effect but is a useful technique to simulate shadows and light patterns. Projective texture mapping for real-time rendering is first described in [83]. In OpenGL, texture coordinates are transformed by a texture transformation matrix just like vertex positions are transformed by a model-view matrix. If automatic texture coordinate generation is activated, texture coordinates are automatically calculated and assigned to vertices based on object space or world space vertex positions. Projective texturing uses automatic coordinate generation and modifies the texture transformation matrix. As a result, the texture is mapped on the surfaces just like as if the image is projected by a slide projector.

Figure 2.7 illustrates the transformation steps in projective texturing. As seen in the figure, the steps are similar to the steps of the view-projection transformation for the camera. As a result of projection transformation, polygons far away from the projector will cover smaller texture area (since the texture coordinates are scaled by the inverse projector

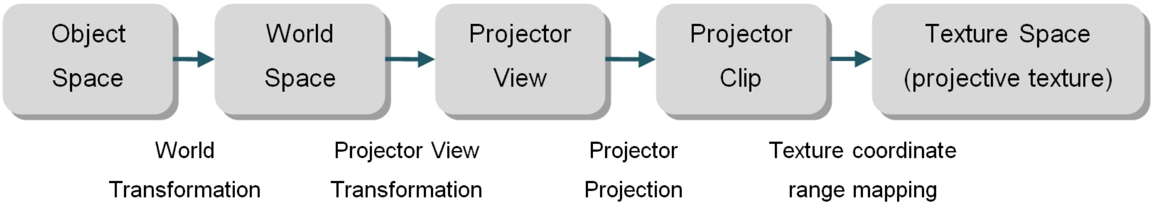

Object Space
World Space
Projector View
Projector Clip
Texture Space (projective texture)
World Transformation
Projector View Transformation
Projector Projection
Texture coordinate range mapping

Figure 2.7: Transformation steps for projective texturing.

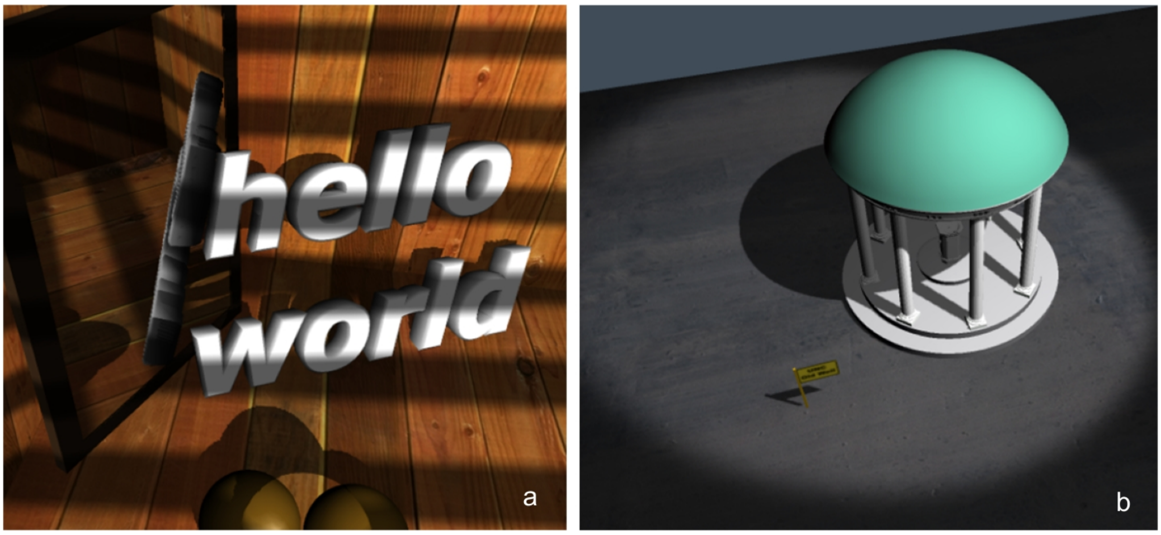


Figure 2.8: Projecting texturing to simulate (a) shadows and (b) spotlight region.

distance). So, as the distance from the projector increases, the projected texture image gets bigger. Projective texturing can be used for many different purposes including shadows, reflections, illumination, slide projector effects and reprojecting the photographs of objects onto scene geometry. Shadow and spotlight effects rendered by projective texturing are shown in Figure 2.8.

Shadows

In rasterization based real-time rendering, majority of the shadow techniques are variations of either shadow volumes (geometry based) or shadow mapping (image based) techniques. Shadow mapping is an image based shadowing technique as described in [98]. It is a multipass technique, in which scene must be rendered twice for each shadow casting lights. In the first pass, shadow map is generated. In this pass, whole scene is rendered from the view point of the light. Because only depth values are needed, only depth buffer is enabled and captured in this pass. As a result of this pass, the depth buffer contains the distance information

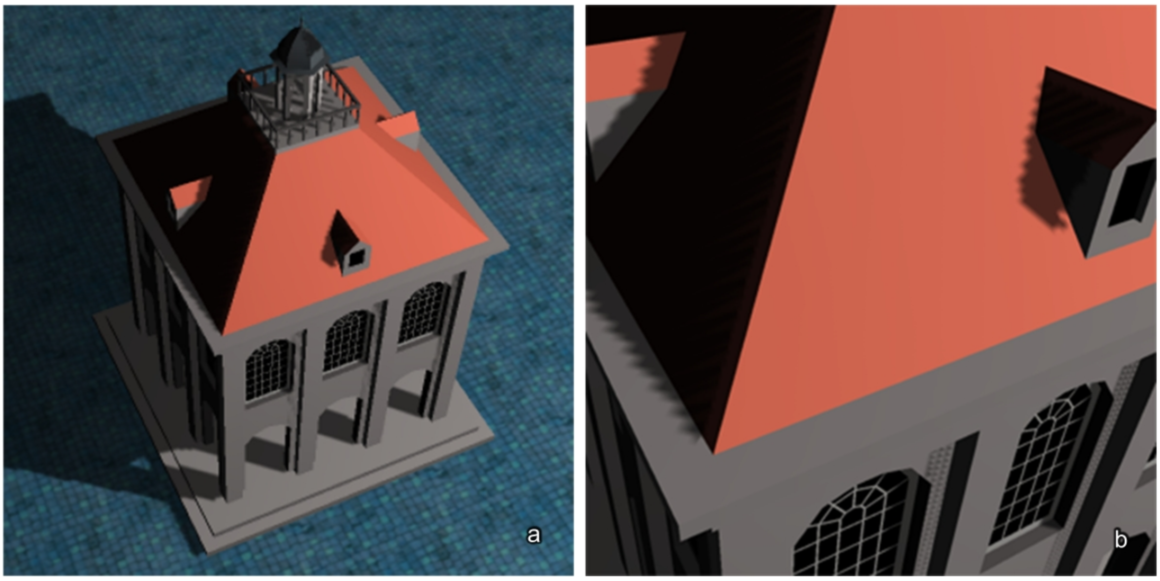


Figure 2.9: (a) Shadow mapping. (b) Close-up view of the shadow mapping artifacts.

between the light source and the points seen by the light. In the second pass, scene is rendered normally from the view point of the camera. In this pass, shadows are overlaid to the scene by using projective texturing. During this operation, depth values of the pixels being rasterized are transformed into the view space of the light and compared to the depth values captured in the first pass. If the captured depth value is lower than the transformed depth value of the pixel, the pixel is considered as in shadow. Figure 2.9 demonstrates shadows rendered with the shadow mapping technique. Most of the graphics accelerators have built-in support for shadow mapping.

Depth map should be created and projected for each shadow caster light source. A major problem with shadow mapping is aliasing. Because the shadow map is projected using perspective projection, projected pixels gets bigger as they get far away from the light. This produces large aliasing effects for distant pixels especially when the camera is close to the distant shadowed pixels. The aliasing effect is demonstrated in Figure 2.9-b. Classic texture filtering methods does not work for shadow mapping because bilinear, trilinear or anisotropic filtering of depth values invalidates the distance information and results in shadow artifacts. Other filtering methods such as percentage closer filtering can be utilized for shadow maps [76]. Most graphics accelerators support percentage closer filtering in hardware. It is also possible to use fragment programs for shadow filtering. On the other hand, since there is a limit for the dimensions of the depth texture, aliasing may still occur (due to undersampling of depth information). Several methods have been developed to minimize the aliasing problem. Stamminger et al. [85] modifies the projective transformation matrix of the depth maps to obtain higher sampling rate where the viewpoint is near to shadowed pixels. There are lots of tecniques developed to eliminate aliasing. Some of them includes using additional geometry on object silhouettes to smooth the shadow borders, filtering shadow images, or using hybrid of shadow mapping and shadow volumes.

Another problem is depth buffer precision. For large scenes, 16bits or 24bits of depth resolution, which is standart for most graphics hardware, may not be enough. To minimize the problem, near and far clip planes of the light frustum can be relocated so that they tightly confine the scene geometry inside the frustum. A bias factor that changes the vertex depth values without changing the projected screen coordinates can also help to avoid the problem [76, 99].

Shadow volumes, another popular approach for rasterization based real-time shadows, was first described by Crow [15]. Unlike shadow mapping, it is a geometry based method. Precise shadows without aliasing problems can be rendered using shadow mapping. The idea is to create shadow volumes by projecting the light to each vertex of the shadow casting object. Pixels falling inside the shadow volume is considered as occluded from the light source. In order to create shadow volumes, front facing and back facing polygons with respect to the light source are grouped. The silhouette edges which separate front facing polygon groups from the back facing polygon groups are extruded along the direction away from the light source.

Heidman described utilizing stencil buffers to render shadow volumes on the graphics hardware [42]. Shadow volumes are generally rendered with this technique on the GPU. He proposed rendering front facing shadow volume surfaces and back facing ones in separate passes. In this case, the number of front facing surfaces in front of an object is greater than the number of back facing surfaces, if the object is in shadow. In order to count front facing and back facing surfaces he used the stencil buffer. When a front facing pixel is rendered, the stencil value of the pixel is incremented by one. Similarly it is decremented by one when a back facing pixel is rendered. After the stencil buffer is filled, the scene is rendered by masking out the pixels with positive stencil values (i.e. shadowed areas are not drawn). This method is also known as *depth pass* shadow testing since it alters stencil values only for the pixels passing the depth test (the pixels in front of the object). Heidmann's method has some problems when the eye itself is in a shadow volume. In this case all stencil values are biased by -1 since the camera sees only the back facing surfaces of the enclosing volume. This bias results in incorrect shadow rendering since it inverts the shadowed regions. Bilodeau and Songy developed another shadow volume technique which works correctly for all camera positions [6]. Instead of counting the shadow surfaces in front of the object, they counted the surfaces behind the object. Therefore this method is also known as *depth fail* shadow testing.

Shadow volumes reuqire huge pixel fill rate because shadow volumes cover most of the screen with high depth complexity. Another problem is that, sharp egdes shadows which looks unrealistic in most cases. In the literature, lots of newer techniques were proposed for realistic smooth borders and for reducing the required fill rate by optimizing the shadow volumes. Prior to the programmable vertex processors, geometry of shadow volumes were created using CPU and then rendered with the hardware using stencil buffers. However with the current graphics cards, it is possible to create silhouette edges on the fly by the GPU.

Figure 2.10 demonstrates projective texturing, shadow and reflection effects combined on a single image. Inclusion of these light effects greatly contribute to the realism of the image.

2.3 GPU Based Ray-tracing

Ray tracing is computationally expensive, and thus considered as an off-line rendering method until recently. However, it has some important advantages which can not be overlooked. Logarithmic complexity and automatic occlusion culling, parallel scalability, coherence, efficient shading, correct and easy rendering of specular reflections and transmission effects are built-in capabilities for a typical ray tracer. Lately, with the advent of more capable hardware, interactive ray tracing research gained more popularity. Wald et al. [93] discuss interactive ray tracing, and conclude that especially for geometrically complex environments rendering speed is comparable to that of the traditional rasterizer hardware. Ray tracing is expected to be a viable alternative to raster based graphics rendering in the not so distant future [45]. Some of the recent works focused on accelerating ray tracing algorithmically [77, 91], while some others centered upon specialized ray tracing processors [38, 81, 82, 100]. Utilizing graphics processors of the commodity graphics cards for ray tracing is another research area drawing increasing amount of attention. Originally, graphics processors are meant to rasterize and shade simple primitives such as triangles or lines. However today's graphics processors are programmable parallel processors. Huge processing power and steep acceleration rate in speed led many researchers to develop GPU specific solutions to known problems. Many graphics and non-graphics related problems were successfully mapped to the GPU programming model [34]. Among these, GPU based ray tracing acceleration is

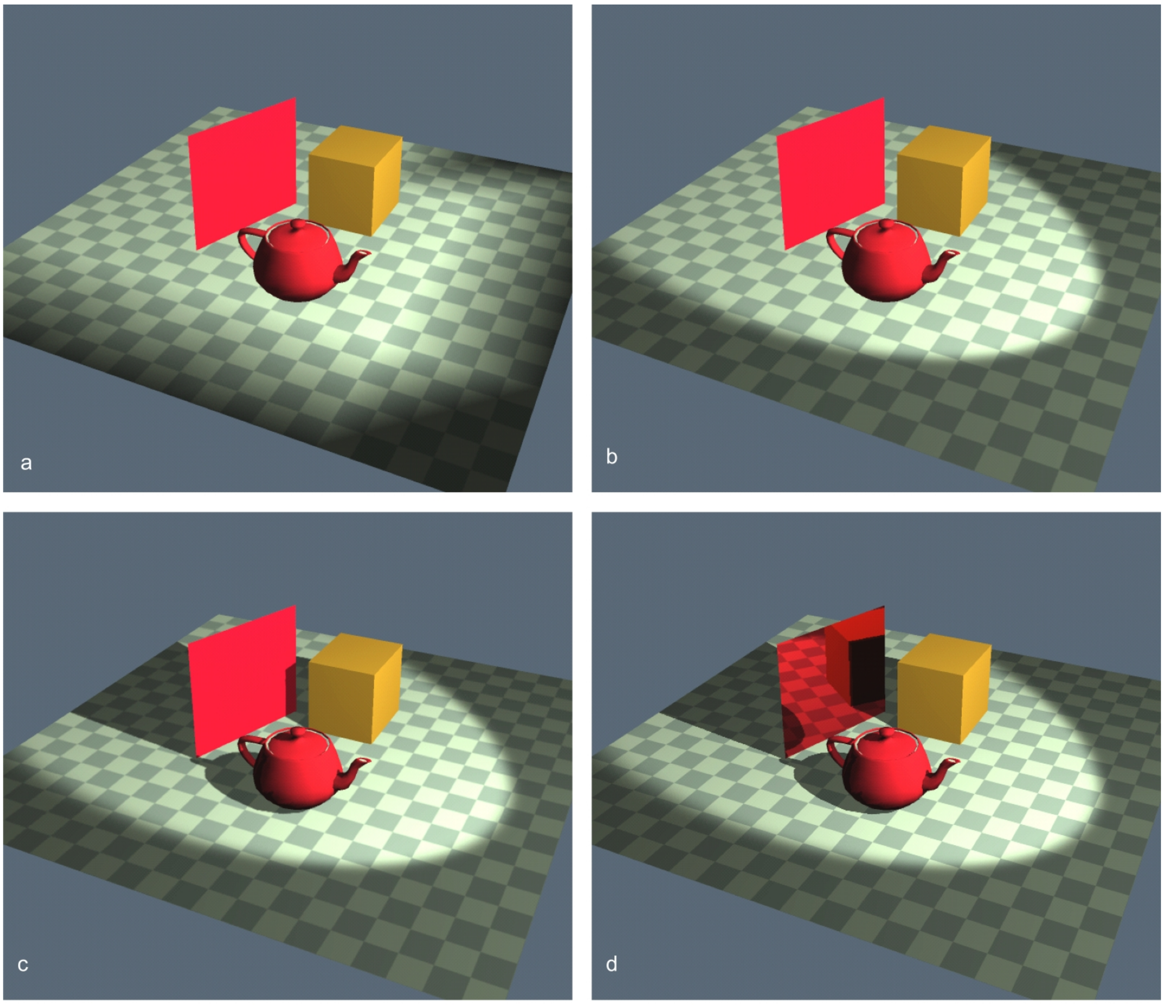


Figure 2.10: (a) Standart OpenGL shading. (b) Projective texturing added to improve spotlight borders. (c) Shadow mapping applied. (d) Reflections added.

relatively new. In [11, 74, 73] it is shown how to use GPU for ray tracing computations. Karlsson et al. [50] have implemented a ray tracer that runs fully on GPU utilizing empty space skipping data structures, while Weiskopf et al. [94] have developed a GPU based non-linear ray tracer.

One of the first works on GPU based ray tracing is proposed by Purcell et al. [74]. They implemented a complete ray tracer on a GPU simulator. Their implementation relied on assumptions about specific GPU capabilities, which are available or can be easily emulated by current GPUs. Their ray tracer works on triangle meshes using uniform voxel grid acceleration structure. They demonstrated that a GPU based ray tracer could prove to be faster than CPU based implementations. They also concluded that GPU assisted ray tracer could be competitive with traditionally accelerated rendering of rasterizing hardware.

Carr et al. [11] used GPU as a co-processor for accelerating ray-triangle intersection tests. They conceptualize programmable pixel processors as crossbars. They did not use an acceleration structure on the GPU and test every triangle with every ray using the crossbar. Triangle information is sent to the crossbar within the vertex attributes of screen sized quads. In their work, strong points of CPU and GPU are emphasized and consequently only intersection tests, where GPU is strong, are handled by the graphics hardware. They group rays using CPU and send them as batches to the GPU for ray-intersection tests, and process the results using CPU. They achieved 34% speedup using GPU aided ray tracing over pure CPU implementation.

Purcell et al. [75] used a modified photon mapping algorithm fully running on GPU. Note that "fully" does not mean that the entire program works on GPU. CPU assistance is inevitable in buffer handling and communications, initialization, memory management and some other API stuff. Although interactive feedback rates achieved for small viewports and very simple scenes, their method is progressive can not converge to the full rendering in real-time. They did not use triangle meshes. Instead, their test scenes were composed of several geometric primitives which are very fast to ray trace.

Foley et al. [28] implemented kD tree acceleration structure for ray tracing on GPU. They devised two kD tree traversal method suitable for the GPU. Among these, kD-restart method eliminates the stack operations. As a result, in order to continue the traversal with next tree leaf, the search is simply restarted from the root. The other variant is called as kD-bactrack. This variant modifies kD-restart to maintains linear worst-case bounds during leaf search. In order to achieve this, the bounding box and the link to the parent are stored for each node. By this way, the search is restarted from the closest ancestor containing the next traversal point, instead of starting from the root node. They identify data recirculation (necessity to reload intermediate values for each kernel run), and poor load balancing (low GPU utilization when small number of fragments are processed) as the main bottlenecks of a GPU based ray tracer.

In a recent paper Horn et al. [44] optimized the kD-restart algorithm, by introducing packetization, push-down and short-stack concepts. Packetization processes rays as groups (ray packets) by taking the advantage of four-wide SIMD instructions of the GPU. Push-down localizes the rays into sub-trees and restrarts the leaf searches from the root of the sub-tree. Short-stack is a small, fixed depth stack implementation. They used a single pass kernel rather than multi pass, and reported an order of magnitude faster performance than the original kD-restart algorithm.

Popov et al. [72] proposed another kD tree acceleration method. Similar to [44] they used ray packets to better exploit the parallelism of the GPU. In order to remove stacks, they used a kD tree with ropes and extended the search algorithm to handle ray packets. Ropes link each leaf node to the smallest node which enclose all possible adjacent nodes via each of the six faces of the node.

Bounding volume hierarchy (BVH) based methods were also successfully used for ray tracing on GPU. Among them Thrane and Simonsen [87] implemented a GPU based ray tracer by using a couple of different BVH construction criteria. They interleave scene data with the acceleration structure and proposed a stackless traversal on the volume hierarchy.

Carr et al. [12] used BVH data structure to accelerate animated scenes on the GPU. Their method is based on geometry images [36], in which geometry information is mapped to a 2D image. They utilize GPU to create volume hierarchies in real time by creating image pyramids from geometry images. Their method allows for real time ray tracing of animated scenes with deformable objects.

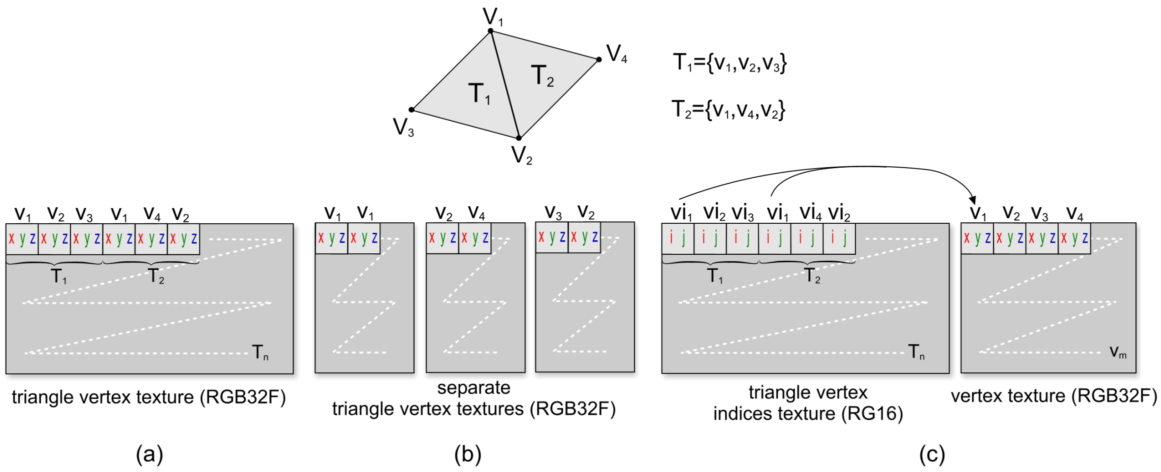
In a recent paper Günther et al. [32] extended BVH traversals to handle ray packets. They proposed a fast CPU based BVH construction algorithm which utilizes surface area heuristic by using streamed binning. Their GPU based ray tracer can handle large scenes. In the paper they demonstrated GPU based ray tracing on a large scene which consists of 12.7 million triangles.

Roger et al. [79] used a hybrid technique and rendered the primary rays with rasterization, while utilized ray tracing for the secondary rays. Their method is based on ray-space hierarchy and applicable to animated scenes. Instead of rebuilding or updating the scene hierarchy, ray-space hierarchy is built on the GPU for every frame. Traversal is also done on the GPU. They also propose fast stream reduction methods to prune empty branches after traversing the hierarchy levels.

2.3.1 Organization of a GPU Ray Tracer

The data structures of a GPU based ray tracer are encoded as 1D, 2D or 3D textures with various color formats, vertex streams or pixel buffers. The data typically include object database, textures and other material information, intermediate and final values, color buffers and acceleration structures. GPU constant registers are also used to transfer lightweight data such as light positions, scene information, number of objects etc.

Triangle meshes are very capable in modeling almost all kinds of virtual environments. Most GPU ray tracers use triangle meshes for the scene description. All three vertex co-



(a) (b) (c)

Figure 2.11: Examples of three different data structures for storing the triangle vertices. (a) All triange vertices are stored in a single texture, in which three consequtive vertices define a triangle. (b) Vertices of the triangles put to different textures separated by corners. (c) Indexed access to vertex data.

ordinates of a triangle may be stored in a single texture, or three different textures may be used for each of the vertices. Vertex normals, texture coordinates and other vertex attributes may be stored similarly. Figure 2.11 shows three different configurations for storing the triangle vertices. Other configurations are also possible. In the figure, *RGB32F* denotes Red-Green-Blue color format with single precision floating point components. *RG16* denotes Red-Green color format with 16-bit integer components (other two component formats may also be used). Note that, 16 bits is enough to index $2^{2\times16}$ vertices. $V_1..V_m$ are the vertex coordinates and $T_1..T_n$ denote triangles. $vi_1..vi_m$ are vertex indices of the triangles. Each of these configurations has strong and weak points. Figure 2.11-*a* and *b*, consume more memory than *c*. However, *c* requires indirect access (dependent texture reads), which slows down reading of vertex coordinates. For simple scenes, a 1D texture may be enough to keep the scene vertices. In this case, vertices are indexed with a single index parameter instead of two. For larger scenes, 2D or 3D textures can be used. In this case, two or three indices (one for each dimension) can be used as in Figure 2.11-*c*. Alternatively a single number can be used which will eventually be decomposed into two or three indices. In *c*, two values (i, j) are used for indexing the vertices therefore two component texture format, RG16, is used. Other primitive types such as planes, spheres, etc. may be modelled with a number of parameters describing the geometry (origin, axis, radius, dimensions etc.) and stored in textures similarly.

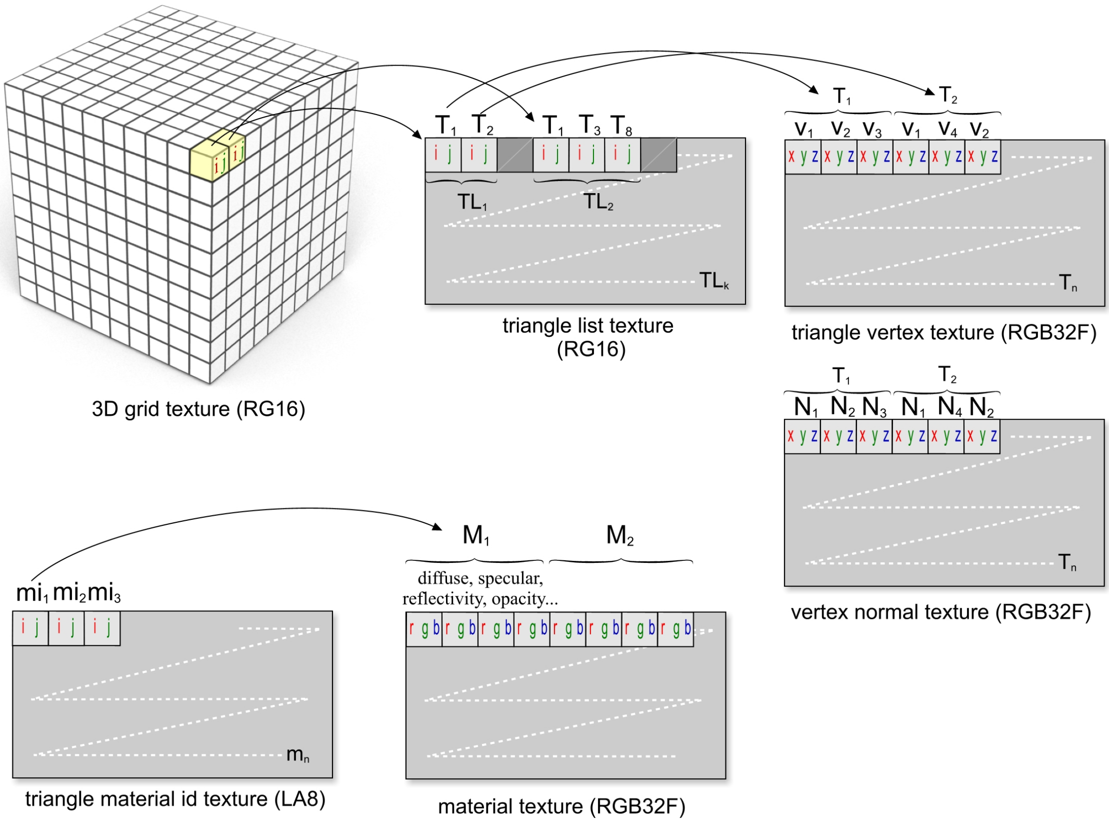

T_1 T_2
T_1 T_3 T_8
i j i j i j i j i j
TL_1
TL_2
TL_k
triangle list texture (RG16)
T_1 T_2
V_1 V_2 V_3 V_1 V_4 V_2
x y z x y z x y z x y z x y z x y z
T_n
triangle vertex texture (RGB32F)
3D grid texture (RG16)
T_1 T_2
N_1 N_2 N_3 N_1 N_4 N_2
x y z x y z x y z x y z x y z x y z
T_n
vertex normal texture (RGB32F)
M_1 M_2
diffuse, specular, reflectivity, opacity...
mi_1 mi_2 mi_3
i j i j i j
m_n
triangle material id texture (LA8)
r g b r g b r g b r g b r g b r g b r g b r g b
material texture (RGB32F)

Figure 2.12: An example organization of the scene database and the acceleration structure. 3D grid texture (acceleration structure) points to triangle lists. Elements of triangle lists are triangle indices which point to triangle coordinates and normals. Each texel of the material id texture points to material information for the corresponding triangle.

Material information of the primitives (triangles, spheres, etc.) may be stored in another texture. In that case, for each primitive a material id (index to the material) is stored in a material index texture. Material parameters such as diffuse, specular, reflection colors can be kept in a material texture.

The acceleration structure, if any used, is embedded into one or more textures. Most of the major acceleration structures were implemented and shown to work with varying efficiencies on the GPU. For 3D grids, 3D textures or 2D encoding of the volume via 2D textures may be used. 3D grid structure for GPUs was first used in [74]. In this structure, voxels point to triangle lists, and triangle lists point to individual triangles as shown in Figure 2.12. In our ray tracer we chose this organization to store 3D grids. The acceleration structure is created by CPU in a preprocessing step, however it is also possible to make use of GPU for this job. For the encoding of other acceleration structures such as kD trees or

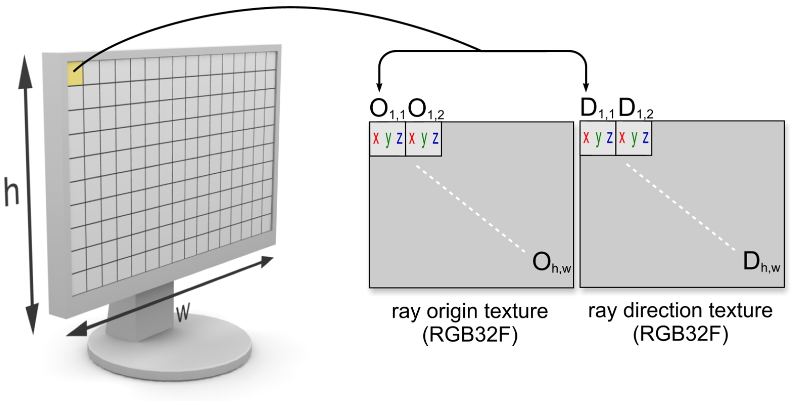


Figure 2.13: Each screen pixel corresponds to a ray. Origin and direction textures are used to define a ray.

bounding volume hierarchies the reader may refer to [12, 28, 32, 72, 87, 91].

Rays can be represented by two textures. One of the textures keeps the ray origin coordinates, and the other one keeps the ray direction vectors. As shown in Figure 2.13 each screen pixel corresponds to the ray passing through that pixel. In order to process rays by using a fragment program; the fragment program is loaded to the GPU and a screen sized quad is drawn to the screen. During scan conversion one fragment is generated for each pixel, and the fragment program is run for each one of them. Fragment program reads the ray information by fetching the corresponding texels of the ray textures. After rays are processed, results are written to a screen sized output buffer. If more than one passes are required, the output values can be used as the input for the next pass. By this way, multiple fragment programs (kernels) may be chained to perform complex operations on rays.

Tracing a ray or a couple of rays at a time is suitable for CPU. However such an action will quickly nullify the speed advantage of the GPU because of the overheads incurred with excessive amounts of kernel executions and under-utilization of GPU processors. For a good GPU utilization, rays should be packed to be processed as big chunks. Ray tracing is a recursive operation which creates ray trees: When a ray hits a surface, shadow rays are created and traced toward the light source. If the ray is not intercepted by a surface along the way, the intermediate (local) shading result is calculated using an illumination model (such as phong illumination). Local illumination should be computed for each light source. Then reflection and refraction rays are fired and their results are accumulated to the partial result according to the surface reflectivity and refractivity. Shading procedure for a classic recursive ray tracer is given in Figure 1.3 (*shade* procedure). However, GPUs do not support recursive functions. Therefore in order to avoid recursive function calls, and to conserve coherency and parallelism as much as possible, rays can be traced as blocks (whole screen can be taken as a big block). When a group of ray is intersected to a set of surfaces, they are shaded and accumulated to the final result as in blocks. Then secondary ray blocks (reflection, refraction and shadow rays) can be fired. In other words, before the shading procedure begins for a block ray, reflection or refraction rays are fired and traced in block-level recursive manner. When secondary rays conclude, results are accumulated backwards. The intermediate values during shading are kept in a block-level stack until all child nodes of the ray tree is traversed.

In the next chapter, our EACD based GPU ray tracer is explained in detail.

CHAPTER 3

EACD ACCELERATION STRUCTURE FOR GPU BASED RAY TRACING

In recent years, there have been several studies on mapping ray tracing to the GPU. Since graphics processors are not designed to process complex data structures, it is crucial to explore data structures and algorithms for efficient stream processing. In particular ray traversal is one of the major bottlenecks in ray tracing and direct volume rendering methods. In this chapter we focused on the efficient regular grid based ray traversals on GPU. A new empty space skipping traversal method is introduced. Our method extends the anisotropic chessboard distance structure (ACD) and employs a GPU friendly traversal algorithm with minimal dynamic branching. Additionally, several previous techniques have been redesigned and adapted to the stream processing model for the comparison. We experimentally show that our traversal method is competitive to other acceleration structures and considerably faster and better suited to the GPU than other grid based techniques.

Ray traversal is one of the most time consuming parts in ray tracing [47] and direct volume rendering methods [57]. Many acceleration structures have been proposed in the past. Havran explains and compares many of these traversal techniques in detail [39]. Because GPUs are parallel stream processors they favor simple localized data access, exploiting instruction level parallelism and arithmetically intensive kernels for maximum efficiency [70]. GPU friendly data structures and algorithms should conform to the stream processing model for efficient processing. 3D regular grid based methods are ideal for hardware stream processing, as they can be represented and accessed efficiently by using 3D texturing capabilities of the GPU. Moreover, traversal algorithms running on them are relatively simple and can be made fit into the GPU programming model with some modifications. Some of the fastest known grid based algorithms use distance transformations to accelerate ray traversals [13, 84, 101]. A number of these techniques were developed for ray casting based direct volume rendering. Essentially, distance based methods utilize distance fields calculated in a preprocessing stage. Distances to the nearest objects are stored in the distance field. Distances can be calculated using different metrics such as Euclidian (3.1), city block (3.2) or chessboard (3.3):

$$Dist_{euclidian}(\delta_x, \delta_y, \delta_z) = \sqrt{\delta_x^2 + \delta_y^2 + \delta_z^2} \tag{3.1}$$

$$Dist_{cityblock}(\delta_x, \delta_y, \delta_z) = |\delta_x| + |\delta_y| + |\delta_z| \tag{3.2}$$

$$Dist_{chessboard}(\delta_x, \delta_y, \delta_z) = max(|\delta_x|, |\delta_y|, |\delta_z|) \tag{3.3}$$

where, $(\delta_x, \delta_y, \delta_z)$ is a vector between two voxels (in our case, it is a 3D vector between a voxel and the nearest non-empty voxel). Distance based algorithms accelerate traversals by skipping empty regions with the information encoded in distance fields. It is worth mentioning here that there are some works not relying on distance fields yet can skip macro (empty) regions [19]. We focused on accelerating ray traversals using regular grids and distance transformations. Our ray tracer completely runs on the GPU and uses GPU friendly data structures and algorithms. Firstly, we introduce a new chessboard distance metric based traversal algorithm. Sramek and Kaufman's [84] acceleration data structure is extended and a GPU based minimum branching traversal algorithm is devised, which we call as extended anisotropic chessboard distance (EACD) traversal. Secondly the previous grid based traversal algorithms are redesigned to fit the parallel stream processing model of GPU. The methods presented also suit well to the streaming SIMD model of the modern CPUs. Finally, efficient implementations and comparisons between the GPU specific versions of the regular grid based traversal techniques are demonstrated.

In this work, three of the previously known traversal methods were adapted to GPU. These methods include Amanatides and Woo's [4] Digital Differential Analyzer (DDA) based ray traverser, Cohen and Sheffer's [13] proximity clouds (PC) and Sramek and Kaufman's [84] anisotropic chessboard distance (ACD) based ray traverser. We choose these techniques because firstly all of them work on grids. Secondly they cover both empty space skipping and non-empty space skipping traversals. In DDA based traversal methods, grid is traversed in a face connected cell incremental fashion. That is one of the neighboring cells is chosen for the next traversal step. Distance field methods (PC, ACD) on the other hand, can skip range of empty cells in big steps. This way, a more inclusive comparison between our method and the previous works is made possible.

In particular Amanatides and Woo's algorithm [4] is a variant of 3D-DDA. It is simpler and requires fewer operations than Fujimoto et al.'s original 3D DDA algorithm [29]. It does not perform empty space skipping. On the other hand we show that it can be implemented on GPU very efficiently using SIMD operations and without data dependent branching inside the traversal loop. Cohen et al.'s [13] proximity clouds utilizes 3D distance fields, which can be represented with 3D textures. The traversal algorithm is not complex. The simplicity of data structure and the traversal algorithm make it a good candidate for stream processing. Non-synthetically generated scenes generally contain empty spaces between the objects. Proximity clouds can skip empty regions in big steps to accelerate traversals. In order not to miss any possible intersections, it switches to face connected cell incremental stepping mode, when a ray is in close proximity to an object. The switching requirement is one of the biggest drawbacks of this method when adapted to GPU since it implies data dependent branching. It is possible to use different distance metrics in proximity clouds such as chessboard, city-block or Euclidian. We have used city-block metric since the distance fields can be created quickly and it results in longer traversal steps. Refer to [13] for a discussion of using different distance metrics.

Sramek et al.'s method [84], anisotropic chessboard distance traversal, is based on chessboard distance metric. Their algorithm is better than proximity clouds in some aspects, such that switching to face connected incremental cell stepping mode is not required and a ray can skip the whole empty region which it resides in. The original implementation can handle not only regular and Cartesian grids but also rectilinear grids. In our work, we have developed a different and GPU friendly traversal algorithm utilizing ACD grids.

In the next section, the overview of our accelerated GPU based ray tracer is given. In the following sections, you can find efficient GPU based redesign and implementations of the previous algorithms. Afterward, our GPU based minimal branching chessboard distance traversal algorithm (EACD) and the underlying data structure is explained in detail. Then the test results are presented with a discussion.

3.1 Accelerated Ray Tracer

We implemented a Whitted style [97] ray tracer with full shadow, reflection and refraction effects in order to compare the traversal methods studied in this work. Basic constructs of our ray tracer are similar to Purcell et al.'s work [74]. Differently, we utilize depth buffer for early fragment culling and use a secondary grid as the acceleration structure. Data layout and buffer semantics are designed in such a way that memory I/O is minimized and more work is done per byte read or written. Where required, data packing is used to save bandwidth. The implementation of the system is done with OpenGL and Cg [60].

Triangle meshes are used for scene description. The scene database is stored in 2D textures. The database consists of vertex coordinates and normals, connectivity information and shader parameters. Two indices (2D texture space coordinates) are used for indexing triangles in the database. 2D indexing makes sense and allows us to address data directly without index to texture coordinate conversions.

The grids are stored as 3D textures. We refer to the grid that keeps the scene partitioning information as the partition grid. Partition grid texture has 2-component 16-bit color format which points to a triangle list. If the voxel is empty, index is set to (-1,-1). Triangle lists further point to triangle indices. Finally, triangle vertices are accessed using the triangle indices. Aside from the partition grid, another grid called as the acceleration grid is utilized for the traversals. Acceleration grids have lower component resolution (maximum 8 bits/color channel) to reduce the memory size and bandwidth requirements.

An off-screen rendering context (PBuffer) with six render buffers and a depth buffer is used for the kernel I/O. 4-component 32-bit floating point color format is used for the render buffers. The system relies on early z-culling by means of depth bounds testing for computational masking. In the course of tracing, rays are given states and specific kernels operate only on the rays of a given specific state. Possible states for the rays are *creating*, *traversing*, *intersecting*, *intersected* or *shaded* and *out*. Ray state values are kept in the depth buffer for efficient masking. In the GPU that we used, it is required to create and modify depth buffer with `LESS`, or `GREATER` depth test function. Additionally, the test direction should not be changed for the subsequent rendering passes. Otherwise, depth buffer optimization breaks down and early fragment culling does not work.

As depicted in Figure 3.1, the system consists of five main kernels with a couple of smaller kernels for ray counting, intersection position/normal calculation, and depth mask modification. Additionally, there is a kernel to fire shadow rays which is very similar to the eye ray generator kernel. A single run of the traversal kernel followed by the intersection kernel is called as a trace step.

Prior to the rendering, ray states are initialized by setting depth buffer to 1 (*creating* state) and depth test function is set to `LESS`. Therefore, in order to be able to write new ray states, monotonically decreasing values are assigned to *traversing* and *intersecting* states in each trace step. Only the rays with the greatest state value are processed by the next kernel.

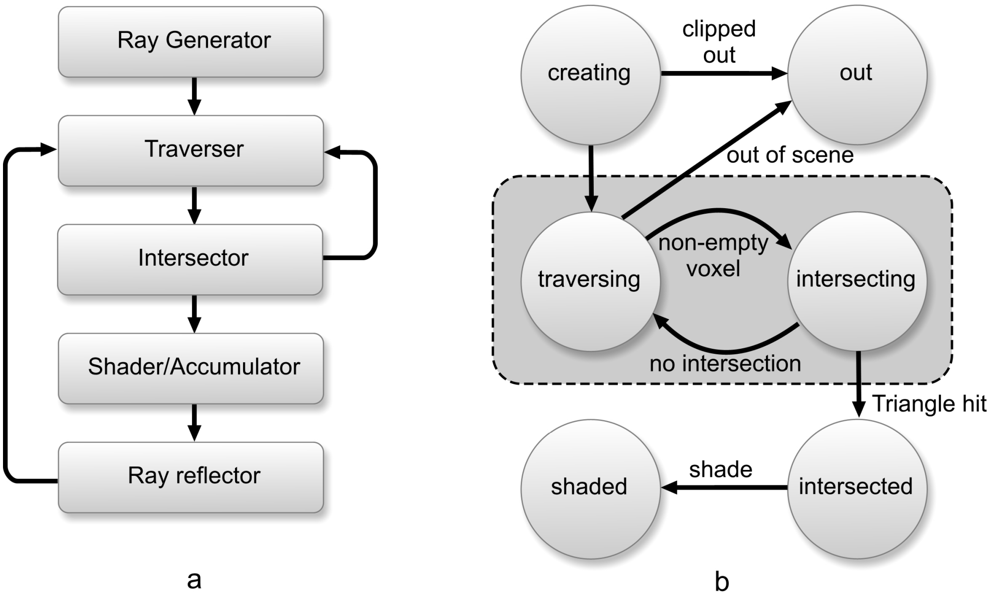


Figure 3.1: (a) Ray tracing kernels (b) Ray state transition diagram.

For the trace step n, where $n \geq 0$, the state value of ray R is calculated as below:

$$statevalue\ (n) = \begin{cases} 1 & \textit{if R is being created} \\ 0.9 - 2\delta n & \textit{if R is traversing} \\ 0.9 - 2\delta n - \delta & \textit{if R is intersecting} \\ 0.1 & \textit{if R is intersected or shaded} \\ 0 & \textit{if R is out} \end{cases}$$

For δ, we use 0.0001 which is sufficiently small to generate enough unique decreasing values for the *traversing* and *intersecting* states. After a main kernel, another kernel is run to write the new state values to the depth buffer, according to the output of the previous kernel.

Figure 3.2 shows detailed input/output streams of the main kernels. As seen in the figure, ray generator kernel creates eye ray origins and directions as two output streams. Eye rays are clipped against the bounding box of the scene. Since our traverser kernels require voxel indices for some computations, rays hitting the scene bounding box are transformed from the world space to the grid space (with unit voxel dimensions). This way, the integer part of the coordinates directly gives the current voxel index. Transformed rays are stored in a different buffer. Grid space rays are used by the traverser kernels. On the other hand, the intersector kernel uses the world space rays since the scene database is stored in world coordinates. After eye rays are generated, depth is set to *out* state for out-of-scene rays and

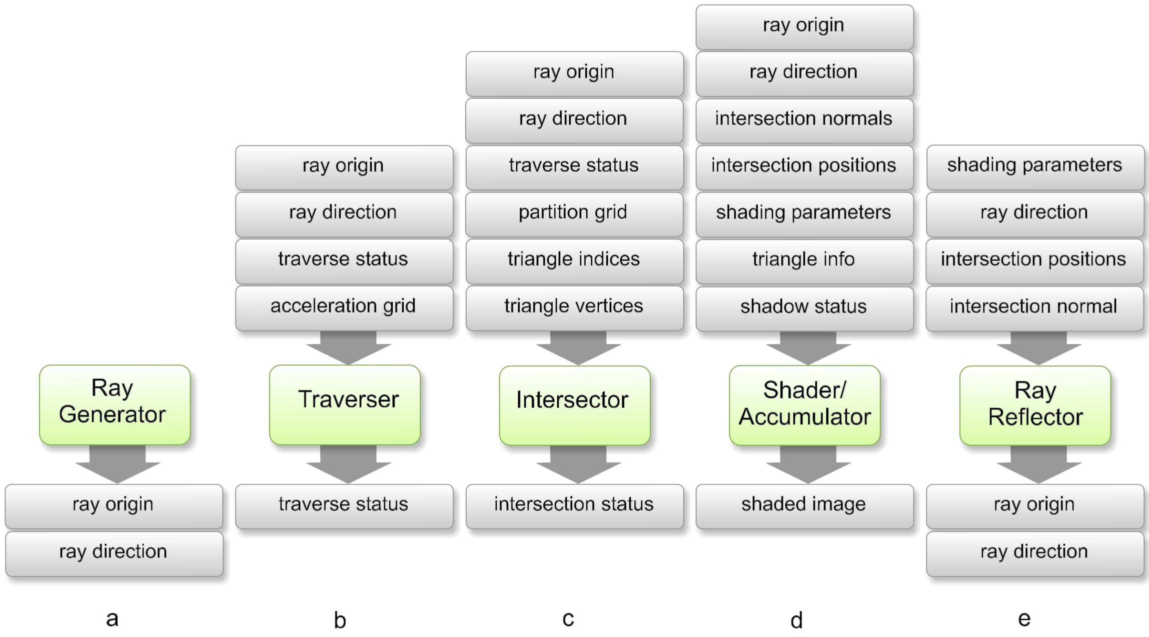


Figure 3.2: Input/outputs of the ray tracer kernels.

to *traversing* state for remaining rays.

The second kernel, which is the main focus of this work, is the traverser. Traversal algorithms are implemented by the traverser kernels. Dynamic loop statements are used inside the intersector and traversal kernels. Traversal loops are repeated until a non-empty voxel is found or the ray gets out of the scene. There are two variants of traverser kernels. One of them is executed in the first trace step, while the other one is called in the subsequent trace steps. The only difference between the two is that the first one begins with checking if the ray starts in an empty voxel and loops until a non-empty voxel is found. The other kernel steps a single voxel first and then loops until a non-empty voxel is found. The output of the traverser kernel is used by the intersector kernel or again by the traverser kernel in the next trace step. Output values are slightly different for each traversal method. As is common to all kernels, the index of the current voxel and parametric distance value to the voxel boundary (used during intersection tests) is written to the output stream. When the traverser kernel is done, depth buffer is modified so that the rays in non-empty voxels are set to *intersecting* state, while the remaining rays are set to *out* state. Then the rays in *intersecting* state are counted. If the count is zero, the shader kernel is called. Otherwise, the execution continues with the intersector kernel.

The third kernel is the intersector kernel, which is employed to perform ray-triangle intersection tests for the rays in *intersecting* state. The intersector reads traverser outputs and accesses the voxel triangles from the scene database. Möller and Trumbore's ray-triangle intersection algorithm is used for intersection test [63]. The output of the kernel is the barycentric coordinates of the intersection point if there is a hit. Otherwise, no output is created. The depth value is set to *intersected* state for intersected rays, or *traversing* state for the other rays. Note that the state value of the *traversing* state is calculated by incrementing the trace step count by one, so that the traverser kernel in the next step can process them. The execution continues with the traverser kernel if there are still some non-intersected rays.

The fourth main kernel is the shader/accumulator kernel, which performs shading calculations for the *intersected* rays. Shader kernel is executed once for each light source and the resulting color values are combined with the intermediate shading results of previous passes. Prior to the shader kernel, intersection positions and normals are calculated by using the results of the intersector kernel. A simple Phong shader is used for the rendering, although it is straightforward to implement more complex shaders. If shadowing is enabled, some extra steps are necessary before executing the shader kernel in order to determine shadowed pixels. These extra steps constitute what we call as the shadow pass. In shadow pass, secondary rays are fired (shadow rays) from the intersection positions toward the light position. The shadow rays are traversed to test if there is an intersection along the path. If an intersection is found, the point in consideration is not visible to the light and thus marked as shadowed (tagged as *intersected*). Shadow pass is nearly identical to casting of eye rays. Almost the same traversal and intersector kernels with minimal modifications aimed for performance optimizations are used for shadow pass. Ultimately, before executing the shader kernel all buffers keeping information about the rays and intersection results are saved to off line buffers. Afterward, shadow pass is realized for each light source. At the end of the shadow pass, the pixels tagged as *intersected* state are said to be in shadow with respect to the current light source. Shader kernel gets the shadow results along with other parameters and illuminates pixels accordingly.

The kernels described up to this point are enough to realize a basic ray caster with shadowing. A basic ray caster considers eye rays only. On the other hand, it is possible to extend the system to perform full ray tracing. In full ray tracing, new rays for reflection and refraction are generated at the intersection points and traced recursively. However, GPUs do not support recursion. In order to facilitate recursion on GPU, we implemented a buffer stack class. The buffer stack operates on a whole draw buffer instead of simple data types and has a typical stack interface. It stores the buffer contents in textures. Buffers keeping the current ray states and the intersection results are pushed into stacks before starting a new reflection or refraction pass. When the pass is finished, old buffers are popped from the stacks. Thus, the recursion is realized in buffer level instead of ray level. Ray reflector kernel is used to fire reflection or refraction rays. The kernel calculates the reflection or refraction directions and generates new rays from the current intersection points toward the calculated directions. Similar to the ray generator kernel, new rays are tagged as traversing state. As stated before, prior to the reflector kernel, current contents of the buffers are pushed into buffer stacks. Then, reflection (or refraction) pass takes place. At this point, as shown in Figure 3.1-a, tracing is continued from the traverser kernel using the reflection (or refraction) rays instead of the eye rays. The recursion continues until a user defined recursion depth is reached.

3.2 Accelerated Traversal Kernels

During the implementation of the traversal algorithms we tried to vectorize instructions, minimize required intermediate states, minimize data dependent branching and provide balance between the memory and arithmetic operations. Firstly, efficient GPU implementations of the previous DDA and PC traversal algorithms will be presented. Next, GPU based minimal branching ACD and EACD traversal algorithms and the related data structures will be introduced.

3.2.1 DDA Traversal

DDA performs face connected cell incremental stepping and does not benefit from empty space skipping. The original algorithm requires two floating point comparisons and one floating point addition at each step. Although it is possible to port the algorithm directly to the GPU, this results in under utilization of GPU. Operations are scalar and data dependent flow control is used heavily to avoid floating point arithmetic. CPUs are fairly optimized for efficient flow control and branching, on the other hand they are relatively weak on intense floating point arithmetic compared to GPUs. We have redesigned the algorithm to use vector operations as much as possible. We also removed the data dependent flow control from the loop body.

In the original algorithm there are several comparison instructions to determine the step direction. Consequently, there are four main code paths in the loop body. Moreover in each path, ray is tested if it is inside the grid boundaries or not. In order to eliminate flow control, vectorized conditional set instructions are used. Because most GPUs are based on SIMD

```
// while voxel is empty & inside the grid
while ( voxel == 0 )
{
  // find minimum parametric distance
  tmin = min(t.x, min(t.y,t.z));
  // determine the step direction
  incr.xyz = (t.xyz == tmin.xxx);
  // advance
  t.xyz += tStep*incr;
  voxelIdx.xyz += cellStep *incr;
  // read next voxel
  voxel = tex3D(texAccelGrid, voxelIdx*invGridSize).a;
}
```

Figure 3.3: DDA traversal loop

architecture, making the comparisons in a single vector instruction instead of several scalar instructions separated with different control paths will results in more efficient operation. Moreover, the ray-box containment tests are postponed and unified with the loop control statement. To achieve this, border color of the acceleration grid texture is set to a specific `OUT` value and texture wrap mode is set to `CLAMP_TO_BORDER`. Hence, if an out-of-grid voxel is sampled, the sampler returns with the `OUT` value. Mono 8-bit color format is sufficient for the acceleration grid. We assign 1 for non-empty voxels and 0 for empty voxels. Border color (e.g. `OUT` value) is set to 0.5. As a result, box containment tests are eliminated and traversal loop continues until the voxel value is non-zero.

Parametric ray equations are used during the traversals. Throughout the context, vectors are shown in uppercase, while vector components are given in lowercase. A point on a ray R can be calculated as: $R(t) = O + Dt$, where O is the origin, D is the direction vector and $t \geq 0$ is the parameter of the ray. The Cg code of the traversal loop is given in Figure 3.3. Some variables should be initialized prior to the loop in the setup phase. Among these, `voxelidx` is the current voxel the ray is in. `cellStep` is a 3-vector denoting the increment to the next voxel in major axis directions, which is calculated as $sgn(D)$, where $sgn()$ is the signum function. `tStep` is the vector of signed parametric distances required to cross a whole voxel, which is calculated as $sgn(D)/D$. `invGridSize` is the reciprocal vector of the grid dimensions. It is used to convert integer voxel indices into [0,1] ranged texture coordinates. Finally, `texAccelGrid` is the acceleration grid texture. Note that $1/D$ may produce floating point specials $\pm\infty$, which can cause `NAN` (not-a-number) result in the subsequent operations [65]. To prevent errors $1/D$ can be clamped to $[-\Gamma, +\Gamma]$ range, where Γ is a sufficiently big number.

3.2.2 PC Traversal

In the original algorithm distance information is stored in the background (empty) voxels. Instead, we keep distance values in the acceleration grid. Acceleration grid has 2-component (luminance-alpha) 8-bit texture format, capable of representing the maximum distance value of 255. Distances are measured using city block metric. Luminance component is set to the distance value, while alpha is set to the voxel status (empty or non-empty). Similarly to the DDA traversal, alpha component is set to 0 for empty and 1 for non-empty voxels, while alpha of the border color is set to 0.5. Because distances are calculated from the voxel centers, rays may miss non-empty voxels if they advance as much as the distance value. Distance values are subtracted by 1 and stored as subtracted to prevent this situation.

PC traversal requires switching between DDA and space skipping. In the original algorithm traversal in skip mode continues in a loop until a ray is close to a non-empty voxel. Then face connected traversal continues until the ray is no longer in vicinity of a non-empty voxel. In this case two inner loops within a bigger loop are required. We found that replacing inner loops with an if statement is more efficient as it requires less flow control. In each step the distance value is checked: If it is 0, DDA steps are taken; otherwise, PC stepping is performed. You can see our GPU implementation of PC traversal in Figure 3.4. In the figure, `invD` is reciprocal of ray direction D. `invGridSize`, `cellStep` and `tStep` are the same as in DDA. `tStart` is the initial parametric distance to voxel borders from the origin. Calculation of `tStart` is explained in the next section. Lastly `C` is the distance function constant as stated in [13]. Note that since the texture sampler normalizes the color values into [0,1] range, it is required to scale and round samples to recover the integer values. However, latest GPUs support integer numbers. So, the algorithm can be altered and an integer component texture format may be used for reading integer distance values without any conversion.

3.2.3 Anisotropic Chessboard Distance Traversal

Sramek and Kaufmann [84] use chessboard metric for the distance computations. Their chessboard distance traversal method has a couple of advantages over the proximity clouds: Firstly, switching between the stepping modes is not necessary. Secondly, an empty region can be skipped to the full extent, thus it is not necessary to subtract distance values by one as in PC. Their algorithm can also work on rectilinear grids with some additional costs. They observed that in distance based traversals, ray steps get shorter as the ray gets closer to the objects, and many small steps are taken until the ray gets far away from the close vicinity.

```
// while voxel is empty & inside the grid
while ( voxel.a == 0 )
{
  // If close to non—empty voxels, then DDA stepping
  if ( voxel.r == 0 )
  {
    // same as DDA
    tmin = min(t.x, min(t.y,t.z));
    incr.xyz = (t.xyz == tmin.xxx);
    t.xyz += tStep*incr;
    voxelIdx.xyz += cellStep*incr;
    // update ray position
    pos.xyz = O+t*D;
  }
  else { // otherwise PC stepping
    // convert distance to integer
    dist = round( voxel.r * 255);
    // advance
    pos.xyz += C*dist;
    voxelIdx.xyz = floor(pos);
    // update ray parameter
    t.xyz = tStart + cell*invD;
  }
  // read next voxel
  voxel = tex3D( texAccelGrid, voxelIdx*invGridSize );
}
```

Figure 3.4: PC traversal loop

To alleviate this problem, they propose using anisotropic empty regions depending on the ray directions. Rays are classified by the component sign of their directions $(\pm x, \pm y, \pm z)$ giving 8 direction octants. Thus in ACD, instead of a single symmetric distance, 8 empty region distances are computed and the appropriate value to be used is determined by the component signs of the ray direction. Anisotropic distance calculation requires applying small kernels over the grid in 8 passes (one pass for each direction octant). Note that although these 8 macro regions around a voxel form an anisotropic shape, each octant defines a cubic region individually in the grid space (assuming unit voxel dimensions).

The original algorithm divides a traversal step into slave steps and a master step, and uses data dependent branching to choose appropriate steps to minimize arithmetic operations. Although this approach may be good for CPUs it decreases the intensity of arithmetic operations, increases flow controls and ultimately results in poor GPU utilization.

3.2.4 Minimum Branching ACD Traversal Algorithm

In this part, we will introduce our GPU based traversal algorithm for anisotropic chessboard distance transformation grids. In essence, the traversal algorithm is similar to the DDA. For

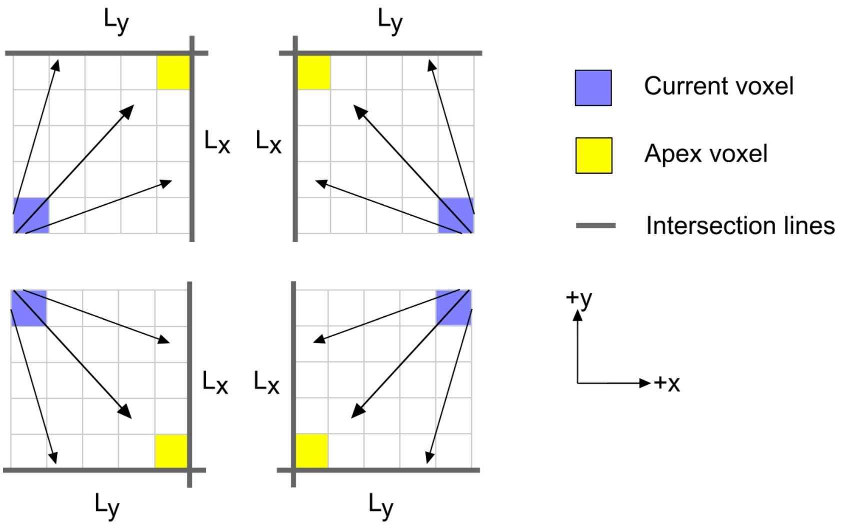


Figure 3.5: Apex voxels and the intersection lines of a rectangular region depending on the ray direction.

the sake of simplicity and conciseness, the algorithm will be explained in 2D without loss of generality. It is straightforward to extend the operations into 3D.

The acceleration grid G has dimensions $w \times h \in Z^+$. A voxel $V \in G$ is identified by the indices i, j where $i, j \in Z$ and $0 \leq i < w$, $0 \leq j < h$. V stores the chessboard distance to the nearest full voxel. So $V(i,j) = \Delta = (\delta_x, \delta_y)$. If $\delta_x = \delta_y$ distances along x and y directions are equal. The parametric equation of a ray R can be decomposed into x and y components as:

$$r_x(t) = o_x + d_x t$$
$$r_y(t) = o_y + d_y t$$

We traverse empty regions in a face connected fashion. Therefore it is required to determine the set of lines (planes) for the region that the ray is possibly intersecting. Intersection lines depend on the ray direction as depicted in Figure 3.5. The corner voxel inside the region which is adjacent to the both intersection lines is called the *apex* voxel. Given a voxel $V_{i,j}$ and a direction D, the intersection line set L is defined as,

$$\begin{aligned} L(i,j) &= \{x = L_x(i), y = L_y(j)\} \quad , where \\ L_x(i) &= i + sat(sgn(d_x)), \\ L_y(j) &= j + sat(sgn(d_j)), \end{aligned} \tag{3.4}$$

Where *sat* (saturate) function clamps values to [0,1] range. Note that since *sat* can be applied as an instruction modifier, it can be executed without performance penalty. If the ray is currently in voxel $V_{i,j}$ and the distance vector is Δ, the indices of the apex voxel i_a ,j_a for the macro region are calculated as:

$$(i_a, j_a) = sgn(D)(\Delta - (1,1)) + (i,j) \tag{3.5}$$

Substituting 3.5 into 3.4, we get the intersection line equations for the region:

$$\begin{aligned} x = L_x(i_a) &= i + sgn(d_x)(\delta_x - 1) + sat(sgn(d_x)) \\ y = L_y(j_a) &= j + sgn(d_y)(\delta_y - 1) + sat(sgn(d_y)) \end{aligned}$$

The parametric distances to the intersection points on the intersection lines are:

$$\begin{aligned} (t_x, t_y) &= \frac{L(i,j) - O}{D} \\ t_x &= \frac{i + sgn(d_x)(\delta_x - 1) + sat(sgn(d_x)) - o_x}{d_x} \\ t_y &= \frac{j + sgn(d_y)(\delta_y - 1) + sat(sgn(d_y)) - o_x}{d_y} \end{aligned} \tag{3.6}$$

The parametric distance t_{min} for ray's next position:

$$t_{min} = min(t_x, t_y)$$

Then next position of the ray can be calculated as:

$$P = R(t_{min}) = \epsilon + O + Dt_{min}, \tag{3.7}$$

$$\epsilon = sgn(D) * 0.0001$$

And the voxel indices for the new position are:

$$(i,j) = (\lfloor p_x \rfloor, \lfloor p_y \rfloor)$$

Unfortunately, current GPUs have questionable arithmetic precision [43]. Therefore we found that adding ϵ to position is required to avoid floating point round-off problems. Otherwise ray may get stuck and never advance to the next voxel due to the precision errors. On the other hand, using ϵ may result in skipping some non-empty voxels when a ray traverses very close to voxel corners. In order to ensure that no triangles are skipped in the intersection tests, we use slightly expanded ($\pm 10^{-6}$) voxel borders for triangle-box containment tests during scene partitioning.

Some of the operations in above equations can be taken out of the loop body and done in the traversal setup. In order to avoid decrementing δ_x, δ_y by 1 each time during the traversal, we store them pre-subtracted in the acceleration grid. The 3.6 can be rewritten as:

$$T = T_{start} + T_{\Delta} \tag{3.8}$$

$$T_{start} = \left(\frac{sat(sgn(d_x)) - o_x}{d_x}, \frac{sat(sgn(d_y)) - o_y}{d_y} \right)$$

$$T_{\Delta} = \left(\frac{i_a}{d_x}, \frac{j_a}{d_y} \right)$$

Where the apex voxel indices (i_a, j_a) are (assuming distance values are pre-subtracted):

$$(i_a, j_a) = (i + sgn(d_x)(\delta_x), j + sgn(d_y)(\delta_y)) \tag{3.9}$$

```
// while voxel is empty & inside the grid
while( voxel.a == 0 )
{
  // compute signed distance
  dist.xyz = round(sgnScaled*voxel.r);
  //find apex voxel
  apexVoxel.xyz = (voxelIdx + dist);
  // compute tmin
  t.xyz = tStart + apexVoxel*invD;
  tmin = min(t.x, min(t.y,t.z));
  // advance
  pos.xyz = oEps+tmin*D;
  // find voxel indices
  voxelIdx.xyz = floor(pos);
  // read next voxel
  voxel = tex3D( texGrid, voxelIdx*invGridSize+offs);
}
```

Figure 3.6: ACD traversal loop

T_{start} is computed once in the traversal setup, whereas T_{Δ} is computed inside the loop. The Cg code for the traversal loop is given in Figure 3.6. Texture format for the acceleration grid is 2-component 8-bit (luminance-alpha). The semantics of the components are similar to the PC traversal. To be able to store 8 distance values per voxel, the size of the acceleration grid is twice as big as the partition grid. It can be considered as voxels are divided into 8 sub-voxels and each sub-voxel keeps the distance information of a specific direction octant. A factor and an offset is used to access to the appropriate sub-voxel. `invGridSize` is the mentioned factor and `offs` is the mentioned offset value. `offs` is calculated as $(invGridSize/2) * (sat(-sgn(D))$. `dist` is the signed distance value and `apexVoxel` is the indices of the apex voxel as in 3.9. `sgnScaled` is calculated in traversal setup as $255 * sgn(D)$. `oEps` is pre-calculated as $\epsilon + O$ of 3.7. T_{start} is denoted by `tStart`.

3.2.5 Extended Anisotropic Chessboard Distance Traversal

Even though ACD traversal reduces the number of ray steps considerably, it is possible to improve the structure further for faster traversals. As expressed in the previous section ACD uses a single distance value for each direction octant. Instead we allow different distance values along x, y and z axes. That is three values are defined for each octant. As illustrated in Figure 3.8 this facilitates non-cubic macro regions, and rays can traverse with greater steps especially in long thin or narrow empty spaces or in close object vicinities. The traversal is almost the same as the explained ACD traversal algorithm. The only difference is instead of a single distance value for all axes, 3 distance values are used representing the distances along the major axes. A 4-component texture format (red-green-blue-alpha) is used for the data, where red-green-blue components store distance values and alpha component keeps the voxel type information (empty or non-empty). EACD traversal code is given in Figure 3.7.

```
// while voxel is empty & inside the grid
while( voxel.a == 0 )
{
  // compute signed distance
  dist.xyz = round(sgnScaled*voxel.rgb);
  //find apex voxel
  apexVoxel.xyz = (voxelIdx + dist);
  // compute tmin
  t.xyz = tStart + apexVoxel*invD;
  tmin = min(t.x, min(t.y,t.z));
  // advance
  pos.xyz = oEps+tmin*D;
  // find voxel indices
  voxelIdx.xyz = floor(pos);
  // read next voxel
  voxel = tex3D( texGrid, voxelIdx*invGridSize+offs);
}
```

Figure 3.7: EACD traversal loop

The memory cost of the acceleration grid for EACD is three times as much as the ACD. In order to reduce this cost, a packed color format (RGB5A1) is used for the grid texture. This format can represent the distance ranges of up to 32 voxels along each axis.

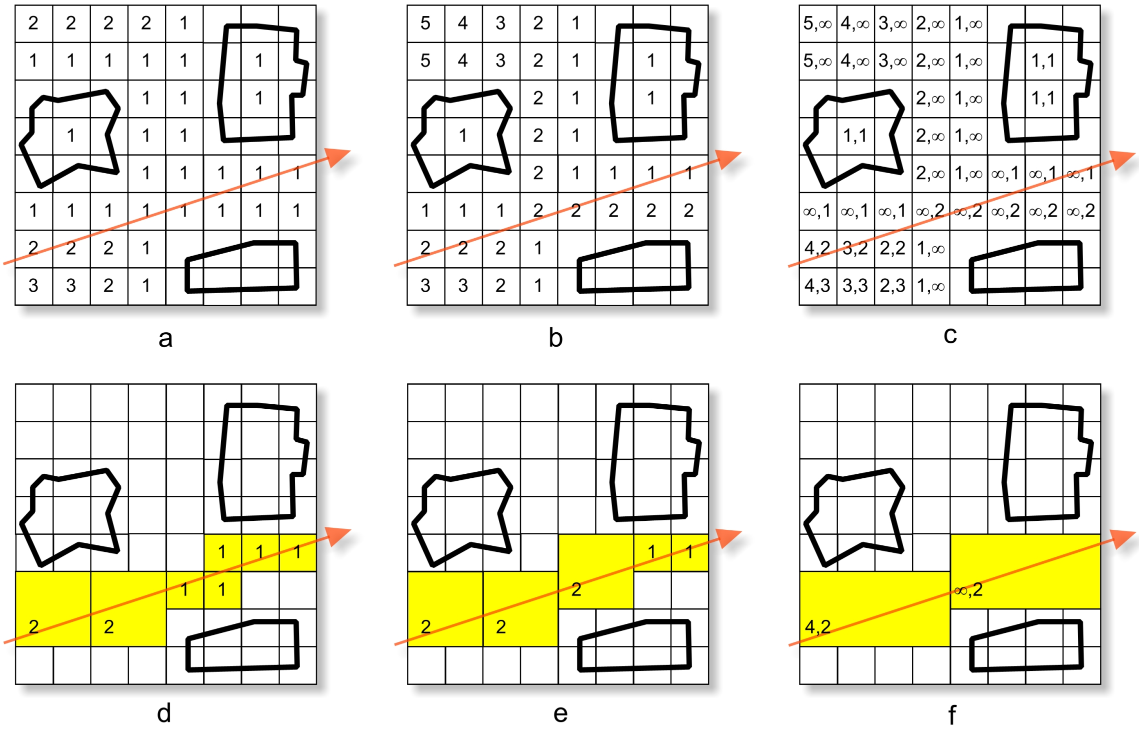


Figure 3.8: (a) CD acceleration grid has single isotropic distance value per voxel (b) ACD acceleration grid stores a distance value for each direction quadrant. Only the (+x,+y) quadrant values are shown (first and second values are the macro distances along +x and +y axes respectively) (c) EACD acceleration grid stores different distance values for each primary axis. (d),(e) and (f) demonstrates CD, ACD, and EACD traversals of an example ray. EACD traversal significantly reduces the number of traversal steps in this situation.

3.3 Construction of The Acceleration Grid

Construction of EACD grid can be carried out in different ways. We use a heuristic with a simple greedy search. The heuristic strives to find the largest empty region by extending the ACD regions. Finding the largest non-cubic empty regions can be a very time consuming process. Therefore, instead of searching the largest empty spaces from scratch, we make use of ACD acceleration grid and extend regions along the main axes. Consequently, building the EACD grid involves two phases: Creating ACD grid and extending to EACD grid.

3.3.1 Creation of ACD Grid

The strategy to create distance transformation is based on the idea of propagating local distances over the grid cells [8]. Firstly, the cell contents of the distance map are initialized to infinity for empty voxels, and to zero for non-empty voxels. Then a kernel, as shown in Figure 3.9-a, is overlaid onto each cell of the map in a specific direction (such as beginning from bottom-left to top-right). Each element of the mask is summed with the value of the corresponding cell of the distance map. The resulting value of the cell is the minimum of these sums and the initial value of the cell. Figure 3.9, depicts these steps for a single voxel. Generation of anisotropic chessboard distance maps involves applying eight different masks to the grid data beginning from one of the corners toward the opposite diagonal corner. Consequently, eight kernels are applied to the volumetric grid, and one (anisotropic) distance map is created for each direction octant. These eight maps are interleaved into a single big grid with eight times the size. The computational complexity of ACD transformation is O(n), where n is the number of voxels. Our single threaded non-optimized ACD creation implementation took 12 secs on a 2.4GHz Intel Q6600 processor for the whole grid. It is possible to create the octant maps in parallel to cut this time down.

3.3.2 Extending ACD to EACD

Before searching empty ACD regions to extend them to EACD, we create six axial distance fields representing distances to the nearest full voxels along the $\pm x$, $\pm y$ and $\pm z$ axis directions. Similar to the ACD grid, the auxiliary grid distances are computed in linear time.

Using the auxiliary axial distance grids we determine, in a greedy manner, how much a cubic ACD region can be extended. This operation is done for all empty voxels of the ACD grid. The extension is carried as follows: Border voxels of the empty region are walked and maximum possible extension along the main axes is computed. The region is then extended along the axis giving the maximum volume. This step is repeated once more, for one of the remaining two axes which results in a larger volume. Figure 3.10 depicts computing EACD macro region for an empty grid cell (corresponds to an empty voxel) in 2D. Our single threaded non-optimized EACD creation implementation took 370 secs on a 2.4GHz Intel Q6600 processor for the whole grid (for maximum region size of $32\times32\times32$). This figure includes both ACD and EACD creation times. Again, it is possible to speed up the grid creation with parallel processing.

Note that the empty regions defined around a voxel by EACD is always larger than or equal to the regions defined by ACD. This is because of the fact that EACD starts with ACD and extends the regions along the main axes. Larger regions results in longer ray steps during the traversal. Especially if a region is extended along the same axes with the ray direction (i.e. the axes defined by component signs of the ray direction), the longer ray steps

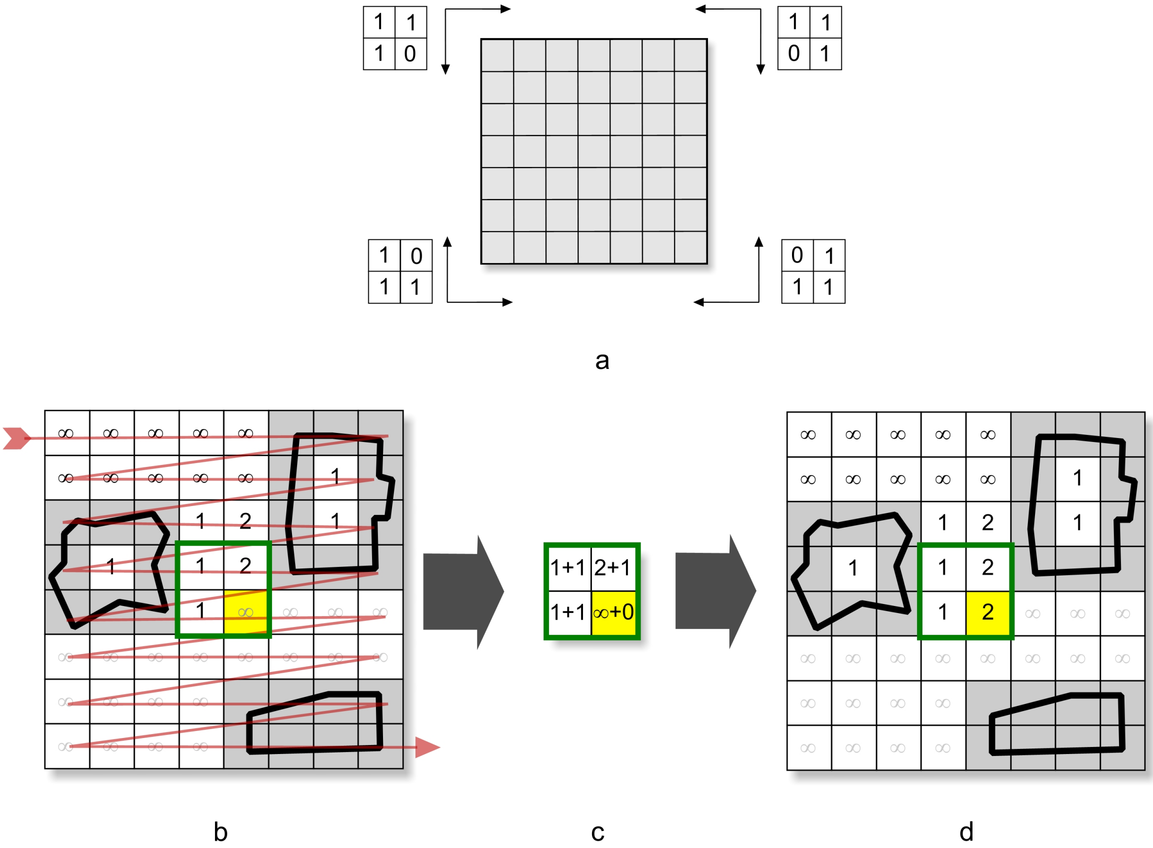


Figure 3.9: (a) ACD kernels for 2D. (b) Creation of the ACD grid for upper-left quadrant. The upper-left ACD kernel of (a) is applied from top-left to bottom-right. Yellow box denotes current voxel. Green frame is the covered region by the kernel. (b) Kernel is overlaid and added with the region. (c) Minimum of the sums is selected as the distance value. Note that, distance of the non-empty voxels (gray boxes) are initialized to 0, while empty voxels are initialized to ∞ prior to kernel application.

are possible. In any case, the length of the ray steps will always be greater or equal within EACD regions. If a lower precision color format is used, like RGB5A1 as we did in our tests, the maximum representable distance value is clamped. This affects the maximum size of the empty regions. However, this will not be a problem if the scene does not have very large empty regions. In this case a higher precision color format may be used, such as RGBA8, at the cost of greater memory consumption.

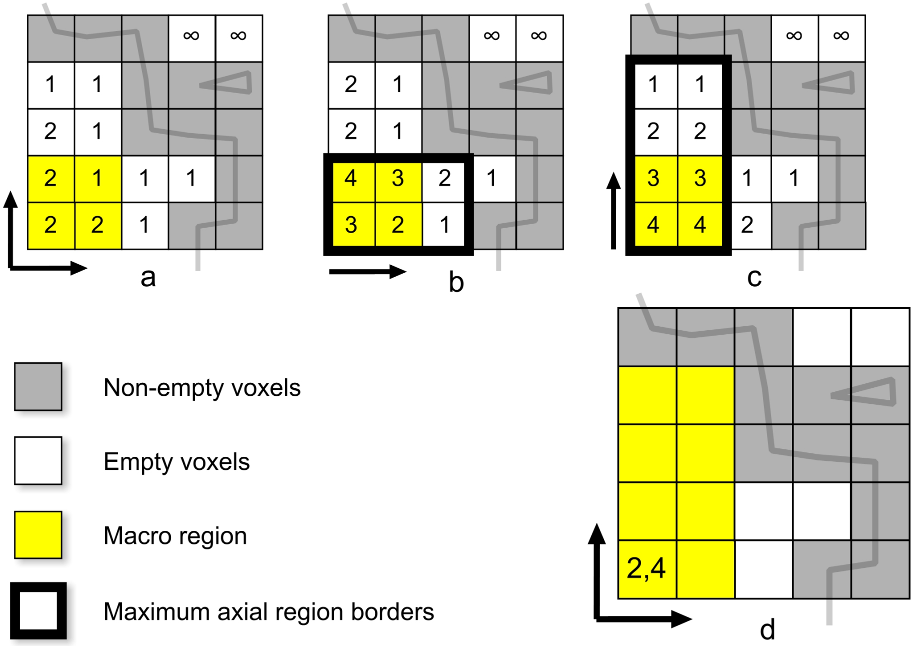


Figure 3.10: Finding the EACD macro region for the lower left cell. Only (+x,+y) direction quadrant is shown (octant for 3D). Arrows denote the orientation of the distance values (a) is the base ACD grid. (b) and (c) are the axial distance grids along +x and +y directions respectively. (d) is the resulting EACD distances.

3.4 Results and Discussion

In order to test the traversal algorithms, some of SPD and the 70K Stanford bunny models were used [37, 89]. Ground plane in some SPD models was reduced in size to make better use of the grid space. The rendered images are shown in Figure 3.14. Tests were performed on a 580 MHz Nvidia GeForce 7800GTX graphics board with 512MB of memory. Release 93.71 drivers and Cg toolkit 1.4 were used. The resolution is 512×512 for all of the images.

In order to collect the rendering statistics, we implemented additional versions of the traverser kernels. The additional versions count the number of loops and the number of texture lookups performed. The collected values are written to a secondary color buffer (by using multiple render targets), without affecting the main rendering operation. After a fragment program is executed, its counter values are read from the buffer. This way, we are able to determine loop counts, the number of texture accesses and approximate bandwidth requirements of each kernel, per pass basis. Execution times, on the other hand, are taken by the original versions of the kernels.

Among the results, the traverser kernel performances are focused in particular since we propose a new traversal algorithm. In a ray tracer, the total rendering time also greatly depends on the intersection testing times. On the other hand, in direct volume rendering [57] there are no intersection tests and the main determining factor of rendering performance is the traversal part. The traversal methods explained here can be applied to volume rendering easily as shown in our previous work [24].

3.4.1 Branching vs. Non-Branching Kernel Implementation

Data dependent branching is used for the loop bodies of the traverser kernels . The other way of implementing the kernels might be to use multi-pass (non-branching) rendering to simulate data dependent loops. Both approaches have some advantages and disadvantages. Note that, multi-pass approach may be the only option for GPUs not supporting dynamic branching. To compare the performances of branching and multi-pass approaches, we eliminated the dynamic loop instructions and built non-branching versions of the traverser kernels.

Table 3.1: Instruction counts of the traverser kernels. In the cells, first number is the fragment program instruction count (including texture lookups). Second number is the texture lookup count.

	EACD	**ACD**	**PC**	**DDA**
Setup/Write	32/4	32/4	34/4	20/2
Loop	16/1	16/1	25/1	12/1

Traverser kernels include a setup phase and a write phase in addition to the loop body. Setup phase consists of a number of texture fetches to reload last ray position and direction possibly with some data unpacking. Additionally, initial and final values of some variables should be computed in the setup and write phases. As shown in Table 3.1 the majority of kernels consist of setup/write phases. The number of instructions and texture fetches are given in the table. Note that the instruction count is not the only factor determining the performance. In a multi-pass implementation, setup/write phases may consume more bandwidth and computational power than the looping implementation since the kernels are called more times. This is especially true for DDA, which has to carry out possibly many small steps, and thus needs many rendering passes. Space skipping techniques suffer less from this condition, as they require less number of traversal passes. Another problem with the multi-pass implementation is that, depending on the number of passes, the overhead of API

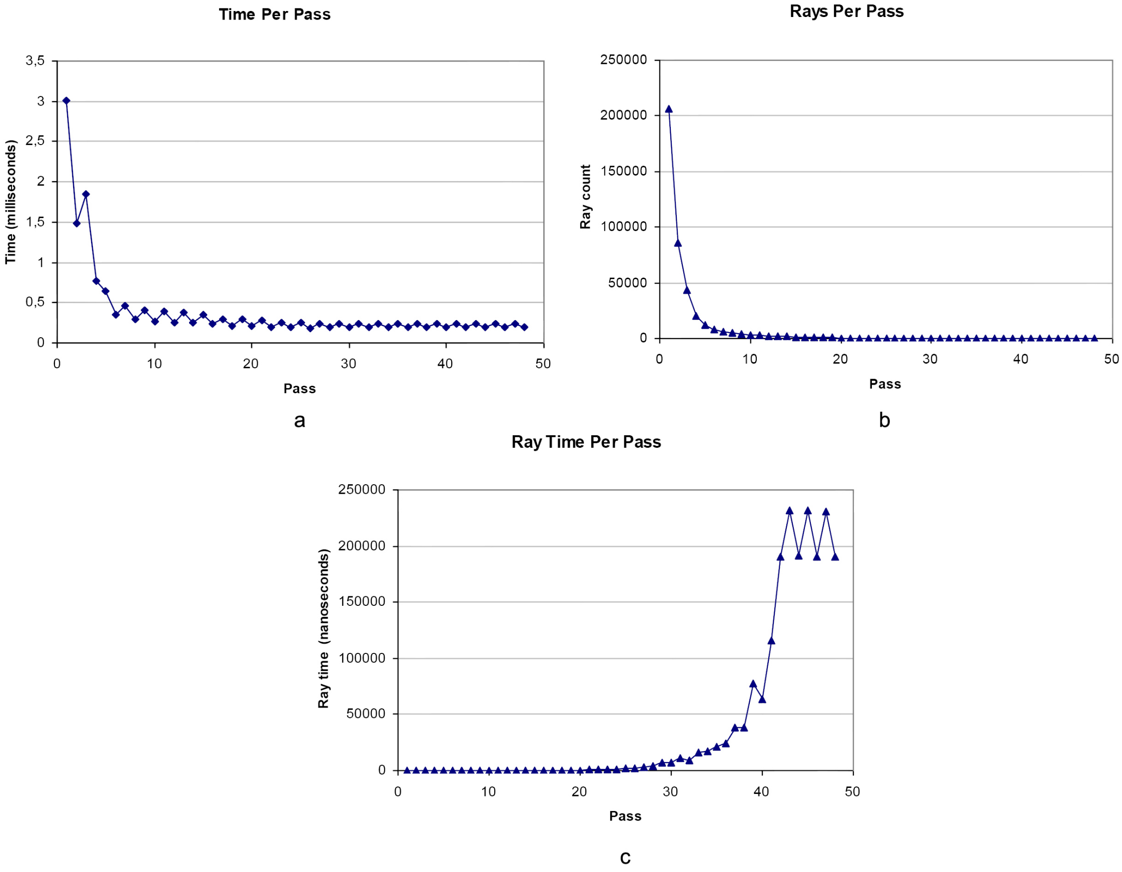


Figure 3.11: (a) Number of active rays per pass (b) Traversal time per pass (c) Average ray traversal time per pass.

calls (kernel switching, state settings, etc.), and some necessary buffer operations, such as modification of depth buffer and depth queries, negatively impacts the overall performance. Especially for the latest generation of GPUs, excessive API calls may be a bottleneck in a multi-pass implementation.

However, data dependent branching is not cheap and can easily results in under utilization of GPU due to deep pipelining and SIMD style parallelism. GPU process fragments as small groups. Fragment processing units of modern GPUs support SIMD style branching: If some of the fragments in a group take a branch while the remaining fragments take another branch, both branches are executed. Therefore all running fragments in the group should follow the same execution path for the highest performance. The group in this context is a rectangular block of fragments. As the block size in which all fragments follow the same execution path grows, the branching performance increases. There is an optimal block size for the best branching performance. The block size depends on the GPU model. Refer to

[54] for dynamic branching benchmarks of different GPUs with respect to varying block sizes. Unfortunately it is hard to make all fragments of a block to follow the same execution path, since the neighboring rays gradually loose coherency and tend to choose different paths during rendering. One way to overcome this problem is to reduce the frequency of data dependent branching. Table 3.2 shows the average number of branching performed by scene rays. Clearly, empty space skipping greatly helps to reduce the branching frequency. The table can also be interpreted as how quickly rays completed the traversal. It is observed that EACD has lower branching frequency and thus requires less traversal steps compared to the other methods.

In non-branching approach, it is very costly to wait for all rays to reach non-empty voxels before the intersection test, which results in large number of traversal passes. Instead, we employed a simple heuristic similar to Purcell et al. [74]; if 20% of the traversing rays require intersection, traversal is interrupted and the intersector kernel is run. This cuts down the number of traversal steps and the traversal times greatly, although increases the total intersection times. For the best results, this heuristic should be fine tuned for each scene and even for different camera setups. DDA benefits largely from the fine tuned multi-pass approach due to high branching frequency. EACD, on the other hand, issued slightly worse and sometimes slightly better rendering times. In order to evaluate non-branching kernels, ray casting traversal times for all test scenes are given in Table 3.3. These figures can be compared to the results of the branching kernels given in Table 3.6. Test results demonstrate that empty space skipping techniques are better than DDA both in multi-pass and branching approaches, and EACD gives the best performance compared to other space skipping techniques.

3.4.2 Fragment Processor Utilization

In order to benefit from the computational power of GPU, fragment processors should be utilized as much as possible. Crudely, utilization can be expressed as the ratio of the number of processed fragments over the number of rasterized fragments. Throughout the execution, the number of active rays to be processed decreases in each trace step, which means that the utilization drops in each subsequent step. Although there is a decline in the utilization, early rejection of the fragments before reaching the shader unit greatly helps to keep the fragment processors busy with useful fragments. Consequently the performance drop is not as sharp as expected for the most of the rendering. We check this situation by measuring the average traversal time per ray in each pass. Figure 3.11 graphs the per-pass results for

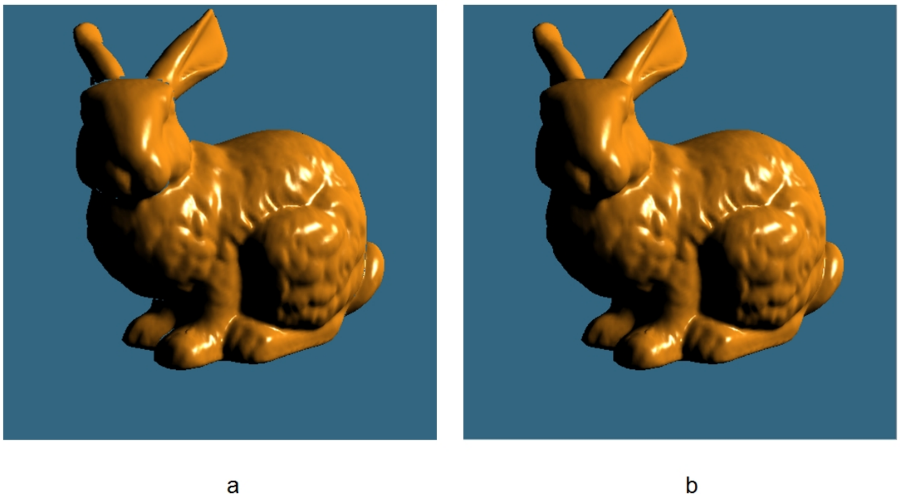


Figure 3.12: Ray casted bunny image (a) after 16 trace steps (b) after all trace steps completed. Note that 99.5% of the rays are already terminated in (a) and the most of the image is rendered. Grid resolution is 128x128x128.

the bunny scene. In the graphs, as the number of active rays decreases in each step, the processing time decreases in a similar shape. Average time per ray is calculated for each step by dividing the total step time to the number of active rays in that step. The "time per ray" graph shows that for the majority of the rays (>98%) time spent is well below 100 nanoseconds. Especially, after around step 20 efficiency declines quickly. In fact, this is expected because the timings include not only the kernel processing time but also API and CPU overheads. Moreover, even if there is just a single active ray left, a screen sized quad is rasterized. Therefore, when the number of active rays is low enough, the mentioned overheads devastate the fragment processing time. However, only a small fraction of the rays suffer from this condition.

In an animation sequence, one possible way to reduce the overheads incurred in the later trace steps might be to cut rendering when the percentage of active rays is lower than a defined threshold. For example, in the bunny scene, 99.5% of the rays are already terminated in step 16. Our other test scenes behave similarly. Figure 3.12 shows the image rendered immediately after step 16. For this image, the partial render time is 65% of the full rendering time (using EACD). When the camera stops moving, the rest of the image can be rendered progressively. It is also possible to approximate the unfinished pixels of the partially rendered image by applying simple image reconstruction filters. On the other hand, this kind of rendering is reasonable for fine grid resolutions. In low resolutions, partial rendering saves less time and exhibits more artifacts. This is because of the fact that the number of trace steps is already low and the artifacts due to the unfinished parts become more apparent since the voxel sizes are bigger.

3.4.3 Timing Results

The traversal algorithms were tested using several grid resolutions. In order to rule out external factors affecting the measurements as much as possible (such as file access, background processes etc.); each test was run many times and the minimum of the timing results is taken. Figure 3.13 illustrates the amount of ray traversal steps taken by each algorithm for the bunny scene. In the images, brighter pixels represent greater traversal step counts. As clearly seen in Figure 3.13, DDA requires the greatest number of steps among all, since it steps only one voxel at a time regardless of the empty regions. PC on the other hand can skip isotropic macro regions in larger steps, while it behaves similarly to DDA in the vicinity of non-empty voxels. ACD performs better than PC in the close proximity of non-empty voxels. This is because of anisotropic macro regions and the fact that rays can take bigger steps according to their directions. EACD performs the best among all the methods. The average traversal loop count (data dependent branching performed) per fragment is 34.6, 4.43, 2.56 and 1.94 for DDA, PC, ACD and EACD, respectively, for this particular scene. Thus, although DDA has the least expensive kernel it should loop many more times than space skipping techniques, resulting in much longer execution.

Tables 3.4, 3.5 and 3.6 show that the ray casting speedup due to EACD is as much as 870%, 358% and 170% compared to DDA, PC and ACD respectively. The speedup increases especially for the scenes where rays have to pass thru non-cubic empty regions. Since both ACD and EACD use the same algorithm essentially, their performances should be similar in the worst case. On the other hand, maximum distance limit imposed by the low precision texture format may prevent full speedup possible with EACD. This is especially apparent in the tree scene with 128x128x128 grid dimensions, where there are large empty regions around the object. It is possible to use a higher precision texture format to alleviate this problem, if GPU has enough memory space. In the sphereflake scene, model fills the grid space more fully and there are many narrow, non-cubic empty spaces where rays can pass thru. Therefore the EACD performance is better compared to the tree scene. As an empty space skipping technique, PC traversal does not perform as fast as EACD and ACD. This is largely due to the longer loop body with relatively higher data dependent branching frequency and shorter ray steps. Despite the fairly efficient loop body, DDA has almost always the worst

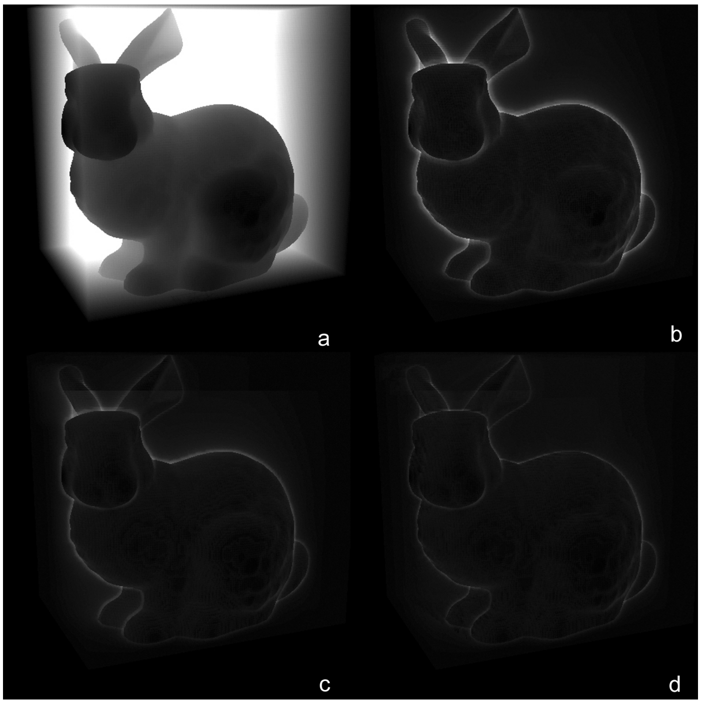


Figure 3.13: Illustration of the traversal step counts for (a) DDA, (b) PC, (c) ACD and (d) EACD. Brightness of the image is set to 170% and contrast is set to 155% for better visual clarity.

performance. Especially in finer grid resolutions DDA can not match the traversal speed of EACD and ACD.

We also compared ray tracing performances of the kernels with varying number of lights and trace depths. The lattice scene is used for the ray tracing tests since majority of the reflected rays stay inside the object and keeps bouncing. Additionally, small curved surfaces reflect rays to different directions lowering the ray coherence rapidly. Test results are given in Tables 3.7, 3.8, 3.9, 3.10 and 3.11, for bunny, tree, jacks, lattice and sphereflake scenes correspondingly. In the tables, *frame* is the total rendering time. The ray tracing setups are; *Ray Cast* (eye rays without shadows), *Ray Cast 1-2SH* (eye rays + one and two lights with shadows), *Ray Trace 1-3R* (eye rays + reflections of depth 1 to 3, without shadows), *Ray Trace 1-3R/2SH* (eye rays + reflections of depth 1 to 3 + two lights with shadows). As seen in the tables, reflected rays cause greater performance hit than shadow rays. This is because of the fact that reflected rays tend to loose coherence rapidly, while shadow rays are more coherent since they are directed to a single point in space (toward the position of the light source). Tables reveal that the performance gap between EACD and other techniques broadens as the coherency gets lost. This is expected since EACD can reach to non-empty voxels with less number of steps and thus requires less data dependent branching operations.

We measured the traversal times with frame sizes of 256×256 and 1024×1024 in addition to 512×512. When the frame resolution is increased 4 times (doubled along each dimension), traversal is slowed down by around 2 times for all methods. This is most probably the result of increasing ray coherency and GPU utilization, and relatively decreasing API overheads, as the frame size grows.

Additionally, the stall rate of fragment processors due to waiting for texture units is measured using NVPerfKit [68]. NVPerfKit is a tool giving access to some low level GPU performance counters. Stalls due to texture sampling may occur if too many incoherent texture lookups are performed. For ray traced lattice scene, average stall rate is 1.77%, 1.20%, 0.89% and 0.24% for EACD, ACD, PC and DDA kernels, respectively. Ultimately, fragment processor stalls due to texture access do not cause a bottleneck on the system.

Ray-triangle intersection testing is the most time consuming part for EACD and ACD based acceleration. Increasing the grid resolution is a way to reduce the number of intersection tests and thus the total rendering time. However for DDA, in most cases traversal is already the most time consuming part. Therefore, increasing the grid resolution will not improve the overall performance in most of the test scenes since it will largely increase the traversal times. Similar situation holds for the PC traversal. The only exception to this observation is the tree scene. This scene has highly non-uniform triangle distribution; most of the triangles are grouped inside a small number of voxels around the branch tips. This is a typical weakness of the grid based scene partitioning structures. In case of ACD and EACD traversals, increasing the grid resolution will help to improve the performance for all of the test scenes. The problem for these techniques is that, the acceleration structure is 8 to 24 times as big as it is for PC and DDA. The limit for the maximum grid resolution for ACD and EACD was 256×256×256 in our test hardware.

3.4.4 Comparison to Other GPU Ray Tracers

Since we focused on grid based acceleration techniques in this work, non-grid based GPU ray tracing techniques have not been implemented. However, we rendered ray casted images of 70K bunny with one light source on a GPU similar to the other works for a rough comparison. Note that all methods compared below uses the same bunny model. As reported in [12], Thrane and Simonsen's BVH implementation [87] obtained 257 ms (GeForce6800 Ultra), while Carr et al.'s geometry images based BVH technique [12] has an estimated frame time of 360 ms (X800 XT PE). Two kD tree based methods as described in [28] rendered the scene in 690 ms using the backtrack algorithm, and 701 ms using the restart algorithm (X800 XT PE). As for the comparison, we measured 141 ms without shadows and 216 ms with shadows on the average, using EACD (GeForce 6800 Ultra, grid size 128×128×96). From the results we conclude that grid based empty space skipping methods, especially EACD and ACD, is very competitive or better than other techniques for bunny like scenes. Additionally, from the test results EACD and ACD are also competitive at scenes with large empty spaces and moderately even triangle density in non-empty voxels. Purcell's grid based ray tracer is similar to our non-branching DDA implementation [74]. Our implementation has dynamic loops for the intersection tests and use depth buffer for early fragment culling; otherwise the two are almost identical. Therefore non-branching DDA results given in Table 3.3 may be used for a comparison to EACD.

EACD and ACD seem to be suitable for static scenes since they require a time consuming pre-processing stage. Other techniques also suffer from this situation at varying degrees. Among the compared GPU based ray tracers, only Carr et al. [12] focused on animated scenes. It is still possible to use grid based methods hierarchically for non-deforming or articulated model animations, where each model has its own grid. Rays entering the bounding volume of a model may be transformed into the grid space of the object and traced locally. For deformable animations, the acceleration structure should be reconstructed. Wald et al. describe a ray tracer using similar approach [90]. Additionally, in a recent work Wald et al. showed that uniform grids can be used for ray tracing of dynamic scenes [92]. We think that it may be an interesting future work to study efficient ways to create the ACD and EACD acceleration grids, either fully or partially, for animated scenes.

Table 3.2: Average data dependent branching per ray for the traversal kernels.

		EACD	**ACD**	**PC**	**DDA**
Bunny	32×32×32	2.45	3.6	7.12	16.58
	64×64×64	2.98	4.32	7.9	33.39
	128×128×128	3.75	4.96	8.54	67.01
Tree	32×32×32	2.4	4.15	11.73	34.62
	64×64×64	2.47	4.47	12.64	69.41
	128×128×128	3.35	4.74	13.3	139.1
Jacks	32×32×32	3.91	4.77	12.36	21.99
	64×64×64	4.09	5.64	14.58	44.2
	128×128×128	4.62	6.19	14.57	88.63
Lattice	32×32×32	1.96	2.91	7.36	7.31
	64×64×64	3.38	4.91	13.01	15.06
	128×128×128	3.91	5.8	17.08	30.52
Sphereflake	32×32×32	2.87	4.41	11.08	21.21
	64×64×64	3.2	5.08	13.15	42.84
	128×128×128	3.56	5.51	14.69	86.12

Table 3.3: Time results in milliseconds of non-branching kernel implementations (grid size is 128x128x128).

		EACD	**ACD**	**PC**	**DDA**
Bunny	Traverse	22.15	23.05	28.13	100.58
	Frame	71.08	73.43	80.04	152.69
Tree	Traverse	18.25	19.2	32.01	181.7
	Frame	202.59	209.09	292.68	413.85
Jacks	Traverse	17.89	20.77	32.56	132.02
	Frame	70.69	83.89	118.07	197.25
Lattice	Traverse	25.17	29.04	39.31	53.27
	Frame	94.51	106.42	154.79	198.37
Sphereflake	Traverse	18.32	20.12	29.09	88.69
	Frame	77.98	83.44	111.95	187.69

Table 3.4: Ray casting time results in milliseconds (grid size is 32x32x32). *Frame* is the total rendering time.

		EACD	**ACD**	**PC**	**DDA**
Bunny	Traverse	8.43	11.79	18.38	17.86
	Frame	100.31	103.76	112.73	112.22
Tree	Traverse	9.02	11.84	23.43	26.01
	Frame	172.29	174.37	195.45	196.81
Jacks	Traverse	15.6	20.72	35.45	29.58
	Frame	162.85	167.79	189.93	184.2
Lattice	Traverse	14.93	19.56	24.36	17.37
	Frame	258.66	262.97	276.8	269.12
Sphereflake	Traverse	11.03	14.18	26.41	28.97
	Frame	208.24	210.58	231.43	244.6x5

Table 3.5: Ray casting time results in milliseconds (grid size is 64x64x64). *Frame* is the total rendering time.

		EACD	**ACD**	**PC**	**DDA**
Bunny	Traverse	12.06	18.43	30.04	40.38
	Frame	62.7	70.73	84.42	93.79
Tree	Traverse	10.4	21.78	38.83	60.84
	Frame	182.35	186.38	215.52	233.99
Jacks	Traverse	18.49	28.97	59.02	67.1
	Frame	102.4	113.41	147.14	155.03
Lattice	Traverse	22.36	29.05	57.62	47.28
	Frame	126.36	136.62	171.01	159.12
Sphereflake	Traverse	15.13	20.85	43.24	52.79
	Frame	128.72	135	160.8	175.37

Table 3.6: Ray casting time results in milliseconds (grid size is 128x128x128). *Frame* is the total rendering time.

		EACD	**ACD**	**PC**	**DDA**
Bunny	Traverse	21.64	36.73	62.38	114.25
	Frame	70.37	86.46	149	165.2
Tree	Traverse	14.96	19.89	46.8	130.38
	Frame	196.61	204.68	235.35	317.18
Jacks	Traverse	24.48	41.77	84.56	165.3
	Frame	76.45	94.2	139.22	235.69
Lattice	Traverse	29.7	40.08	106.56	126.65
	Frame	95.05	105.74	173.86	193.79
Sphereflake	Traverse	22.73	32.99	73.5	136
	Frame	86	96.52	138.82	207.43

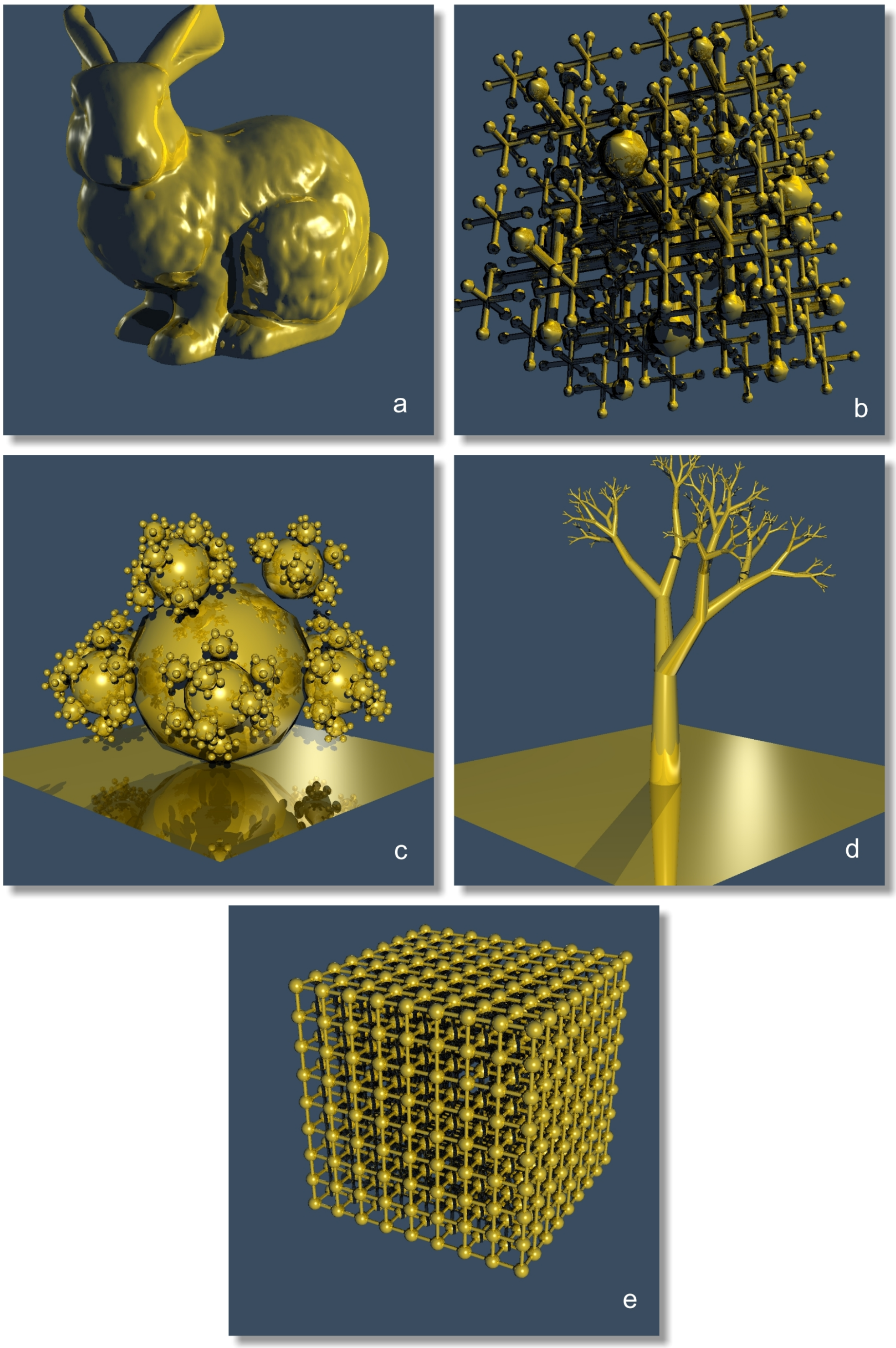


Figure 3.14: (a) Bunny (69451 tris) (b) Jacks (24528 tris) (c) Sphereflake (88562 tris) (d) Tree (67454 tris) (e) Lattice (125388 tris).

Table 3.7: Ray tracing time results of the bunny scene in milliseconds (grid size is 128x128x128).

		EACD	**ACD**	**PC**	**DDA**
Ray Cast	Traverse	21.64	36.73	62.38	114.25
	Frame	70.37	86.46	149	165.2
Ray Cast 1SH	Traverse	46.34	78.07	141.25	332.82
	Frame	146.49	178.13	245.1	436.84
Ray Cast 2SH	Traverse	55.44	92.27	167.31	410.52
	Frame	174.95	212.66	291.29	536.62
Ray Trace 1R	Traverse	61.14	99.49	181.97	446.92
	Frame	201.15	232.49	320.02	586.68
Ray Trace 2R	Traverse	41.93	66.38	116.06	329.65
	Frame	132.29	156.76	208.38	422.86
Ray Trace 3R	Traverse	76.4	121.55	219.44	617.12
	Frame	236.96	282.76	383.46	783.61
Ray Trace 1R/2SH	Traverse	91.5	143.52	256.9	614.38
	Frame	284.98	337.78	453.64	780.88
Ray Trace 2R/2SH	Traverse	104.66	158.53	283.7	779.2
	Frame	324.27	379.61	511.96	1008.14
Ray Trace 3R/2SH	Traverse	35.21	57.23	97.98	264.36
	Frame	111.18	136.44	178.43	347.72

Table 3.8: Ray tracing time results of the tree scene in milliseconds (grid size is 128x128x 128).

		EACD	**ACD**	**PC**	**DDA**
Ray Cast	Traverse	14.96	19.89	46.8	130.38
	Frame	196.61	204.68	235.35	317.18
Ray Cast 1SH	Traverse	36.9	50.4	122.5	366.9
	Frame	523.47	536.86	629.77	880
Ray Cast 2SH	Traverse	42.83	59.22	152.44	424.25
	Frame	620.23	637.16	756.93	1028.88
Ray Trace 1R	Traverse	49.37	63.76	167.6	441.11
	Frame	660.18	673.95	810.93	1069.44
Ray Trace 2R	Traverse	27.41	33.6	92.53	241.3
	Frame	281.92	294.32	357.8	515.68
Ray Trace 3R	Traverse	56.53	71.9	201.91	532.68
	Frame	697.52	712.68	864.63	1199
Ray Trace 1R/2SH	Traverse	66.84	84.62	243.86	600.87
	Frame	817.93	829.77	1020.77	1383.74
Ray Trace 2R/2SH	Traverse	73.92	92.27	265	625.3
	Frame	874.88	892.15	1091	1452.4
Ray Trace 3R/2SH	Traverse	22.38	27.48	69.51	197.56
	Frame	260.86	269.57	322.09	447.96

Table 3.9: Ray tracing time results of the jacks scene in milliseconds (grid size is 128x128x128).

		EACD	**ACD**	**PC**	**DDA**
Ray Cast	Traverse	24.48	41.77	84.56	165.3
	Frame	76.45	94.2	139.22	235.69
Ray Cast 1SH	Traverse	90.47	150.66	361.86	992.87
	Frame	275.29	336.61	558.2	1210.93
Ray Cast 2SH	Traverse	169.4	273.73	712.29	1916.47
	Frame	512.24	617.97	1073.25	2284.95
Ray Trace 1R	Traverse	244.2	382.67	1035.23	2773.18
	Frame	738.44	876.76	1555.87	3302.26
Ray Trace 2R	Traverse	108.39	155.01	453.1	1225.79
	Frame	308.67	355.24	661.82	1440.68
Ray Trace 3R	Traverse	280.16	407.31	1187.6	3176.85
	Frame	794.56	919.97	1725.13	3723.64
Ray Trace 1R/2SH	Traverse	452.35	659.12	1924.38	5089.12
	Frame	1283.88	1492.31	2791.01	5972.57
Ray Trace 2R/2SH	Traverse	616.42	889.5	2620.76	6883.32
	Frame	1751.88	2029.11	3802.17	8090.64
Ray Trace 3R/2SH	Traverse	75.47	110.97	320.68	836.04
	Frame	223.39	263.04	496.35	1002.48

Table 3.10: Ray tracing time results of lattice scene in milliseconds (grid size is 128x128x128).

		EACD	**ACD**	**PC**	**DDA**
Ray Cast	Traverse	29.7	40.08	106.56	126.65
	Frame	95.05	105.74	173.86	193.79
Ray Cast 1SH	Traverse	66.89	92.35	258.15	626.84
	Frame	205	232.76	435.91	802.52
Ray Cast 2SH	Traverse	87.46	125.26	304.67	905.74
	Frame	270.92	308.56	493.49	1096.98
Ray Trace 1R	Traverse	105.32	170.4	408	970.69
	Frame	345.6	409.65	655.17	1219.13
Ray Trace 2R	Traverse	192.26	330.05	760.25	2108.27
	Frame	649.02	787.97	1227.54	2590.06
Ray Trace 3R	Traverse	244.39	421.22	961.26	2788.76
	Frame	833.21	1009.53	1562.62	3409.25
Ray Trace 1R/2SH	Traverse	248.41	387.05	915.503	2873.38
	Frame	801.2	936.01	1474.3	3444.19
Ray Trace 2R/2SH	Traverse	424.17	684.23	1592.51	5141
	Frame	1397.1	1651.37	2582.13	6151
Ray Trace 3R/2SH	Traverse	545.32	888.03	2049	6741
	Frame	1816.2	2149.33	3327.42	8051

Table 3.11: Ray tracing time results of the sphereflake scene in milliseconds (grid size is 128x128x128).

		EACD	**ACD**	**PC**	**DDA**
Ray Cast	Traverse	22.73	32.99	73.5	136
	Frame	86	96.52	138.82	207.43
Ray Cast 1SH	Traverse	69.58	111.8	263.91	577.31
	Frame	273.08	315.11	474.62	797.99
Ray Cast 2SH	Traverse	130.33	224.74	642.73	1222.3
	Frame	514.87	609.88	1050.92	1655.98
Ray Trace 1R	Traverse	200.98	347.93	1189.58	1993.6
	Frame	794.53	941.85	1810.6	2655.78
Ray Trace 2R	Traverse	70.08	93.29	275.09	501.7
	Frame	252.52	276.25	462.15	700.77
Ray Trace 3R	Traverse	182.08	261.77	906.39	1582.4
	Frame	682.43	764.48	1424.56	2127.98
Ray Trace 1R/2SH	Traverse	319.69	478.58	1879.42	2967.79
	Frame	1205.23	1364.74	2795.56	3949.25
Ray Trace 2R/2SH	Traverse	453.74	691.19	2948.48	4363.91
	Frame	1718.93	1956.45	4253.4	5749.95
Ray Trace 3R/2SH	Traverse	51.95	68.27	200.05	353.35
	Frame	190.06	206.98	379	497.69

CHAPTER 4

EACD BASED REAL-TIME VOLUME RENDERING ON GPU

Direct volume rendering methods generate images directly from the volume data without converting it to geometric surface information. Ray casting, splatting [96], shear warp [55] and texture mapping are among the well known volume rendering techniques. In this chapter, accelerated volume ray casting by using our EACD acceleration structure is introduced. A primitive ray caster sweeps the volume through the rays by sampling data at constant ray steps even if the region being traversed is empty. Distance field methods (PC, ACD, and EACD) on the other hand, can skip groups of empty voxels in big steps. In this context, we extended ACD/EACD ray traversals to facilitate the skipping of not only empty spaces but also any homogeneous regions in the volume. During the classification of volume data to form homogeneous regions, we impose some error thresholds. These thresholds are determined experimentally in such a way that, a human eye can not capture the image artifacts caused because of passing over minor differences of the data in homogeneous regions.

Some of the early acceleration methods used hierarchical data structures such as kD trees [86] and octrees [57] to skip empty regions of volume data. Several other works extended the usage of hierarchical data structures so as to provide acceleration of rendering homogeneous regions as well as empty regions [17, 56]. Some other techniques explored other forms of encoding schemes such as look-aside buffers, proximity clouds [13], and shell encoding [88] to skip empty spaces. On the other hand, recent algorithms and data structures were proposed in order to take advantage of internal parallelism and to utilize the dedicated graphics hardware as much as possible[58, 59, 52].

Specialized capabilities of the graphics hardware are extensively used for accelerating volume rendering. Especially, fast texture mapping and filtering facilities of the GPU are frequently employed for real-time volume visualization. In the texture mapping approach, image is rendered by mapping 3D volume slices to the screen aligned planes. Cullip and Neumann's [16] and Cabral's [10] work used this method to accelerate volume rendering. Similarly, Van Gelder and Kim [31] shade the volume data with directional lighting and render the pre-lit data with 3D texturing on hardware. Westerman and Eartl [95] used graphics hardware to shade and render volume data on the fly with ray casting. Meißner et al. [62] introduced a method, where shading and classification of the interpolated data is done by the graphics hardware. Rezk-Salama et al. [78] used fixed function graphics pipelines with multi texturing to render volume data. Engel et al. [23] utilized programmable graphics pipelines for per-pixel lighting with pre-integration. With the advent of programmable GPUs, faster ray traversal methods with empty space skipping and early ray termination had become possible [58, 59, 52, 51].

In this work, we focused on using distance encoding schemes and GPU friendly ray traversals to make efficient use of the programmable graphics hardware. Our method benefits from homogeneous region skipping and integration by using EACD acceleration structure. The remainder of the chapter is organized as follows: Volume ray casting and factorization of the ray integral that we used during homogeneous space leaping is given in the first section. Distance based homogeneous region skipping technique is explained in the second section. Next, our EACD based direct volume renderer is introduced. The test results are presented and discussed in the last section.

4.1 Volume Ray Casting

Raw 3D volume data contains scalar density values in each grid voxel. Optical parameters have to be determined for each voxel in order to display interaction of light with the volume densities and to obtain realistic-looking rendering results. This is performed in a preprocessing step by applying a transfer function, which maps density values in each voxel to opacity and color values. In this work, we used the classification method proposed by Mark Levoy [57]. In addition to opacity and color information, approximate surface gradients are also determined with this classification method. As a result, we use opacity and approximate surface gradients for the optical properties.

During the traversal, ray is sampled trough the ray direction from front-to-back viewing order with a constant step size. At each sample point, the effects of the optical properties are integrated with the effects of the incoming sample's optical properties to obtain the accumulated color and opacity values trough that ray. This operation is achieved with the well known numerical solution for the ray integral (equation (4.1)).

$$\begin{aligned} C_r &= C_1\alpha_1 + (1-\alpha_1)C_2 \\ \alpha_r &= \alpha_1 + (1-\alpha_1)\alpha_2 \end{aligned} \tag{4.1}$$

4.1.1 Homogeneous Space Ray Integration

The sampling and reconstruction operation is the most time-consuming part during the ray traversal. In order to minimize this cost, our method tries to group large number of voxels with similar optical properties to form homogeneous region by using EACD encoding. There is no need to sample and compose data multiple times for a ray segment inside a homogeneous region. Our method exploits the ray integral factorization method as described in [30]. By this way, we make only one fetch operation to obtain the optical properties of the whole homogeneous span. These properties are then used to calculate accumulated color and opacity values for the entire span. Ray integral factorization along a ray is expressed as in equation (4.2).

$$\begin{aligned} & C_r = \sum_{i=1}^{n}\left[C_i\alpha_i\prod_{j=1}^{i-1}(1-\alpha_j)\right] \\ & = \sum_{i=1}^{m}\left[C_i\alpha_i\prod_{j=1}^{i-1}(1-\alpha_j)\right] + \\ & \left[\prod_{k=1}^{m}(1-\alpha_k)\right]\sum_{i=m+1}^{n}\left[C_i\alpha_i\prod_{j=m+1}^{i-1}(1-\alpha_j)\right] \end{aligned} \tag{4.2}$$

$$where, m \in [1, n]$$

According to this factorization, when a ray enters into a homogeneous region and passes through n samples until exiting the region, color and alpha values inside the region are accumulated according to equation (4.3).

$$\begin{aligned} C_{r_i} &= C(1-(1-\alpha)^n) \\ \alpha_{r_i} &= 1-(1-\alpha)^n \end{aligned} \tag{4.3}$$

where, C_{r_i} and α_{r_i} are the accumulated color and opacity values respectively for the homogeneous region i, with n samples. The final accumulated color of the ray is computed using regular ray integration formula (4.4).

$$C_{final} = \sum_{i=1}^{k}\left[C_{r_i}\prod_{j=1}^{i-1}(1-\alpha_{r_j})\right] \tag{4.4}$$

According to equation 4.4, k is the number of regions located in sequence through the ray direction.

4.2 Distance Based Homogeneous Region Leaping

Distance based methods compute the distance values to the nearest non-empty voxel from each of the voxels. In homogeneous region encoding, the distance value represents the distance from the voxel to the nearest neighboring region. Thus, instead of empty/non-empty type, there is same-region/different-region type of classification. This type of classification essentially affects the computation of distance fields and the traversal algorithm itself.

Previous work on distance fields based volume ray casting basically rely on proximity clouds (PC). Cohen et al.'s PC [13] utilize isotropic 3D distance field. Isotropy comes from the fact that a single distance value is computed per voxel, representing the maximum interval that a ray can advance regardless of the direction. When Euclidian metric is used, the distance value represents the radius of the homogeneous region centered in the voxel. In the original method, distance values are stored in background (empty) voxels. On the other hand, we do not classify voxels as empty or non-empty. In homogeneous region encoding, there are simply regions with similar optical properties. Therefore we use a secondary 3D grid for keeping the values of the distance field. Each voxel has a corresponding distance value. In the vicinity of a different region, the PC traversal switches to the primitive ray casting with constant stepping mode.

Sramek and Kaufmann's [84] anisotropic chessboard distance traversal method (ACD) does not need to switch between the stepping modes. In the original algorithm, empty regions (background voxels) can be skipped fully. The authors demonstrated the technique with a volume renderer. Their algorithm can work not only for regular voxels but also for rectilinear voxels with some additional cost. In ACD, instead of a single isotropic distance, eight distance values are stored to define the extents of empty regions. The appropriate distance value (region size) to be used is determined by the component signs of the ray direction. As explained in Chapter 3 we devised a GPU friendly ray traversal technique by using homogeneous ACD regions. Both ACD and EACD volume renderers use the same algorithm with a minor difference. The details of the algorithm are given in the next section.

In this work, we use the same homogeneous ray integration for all of the methods. The ray integration within the homogeneous segment of ray is performed with the factorization method explained in Section 4.1.1. Number of points sampled by the ray in this region (n),

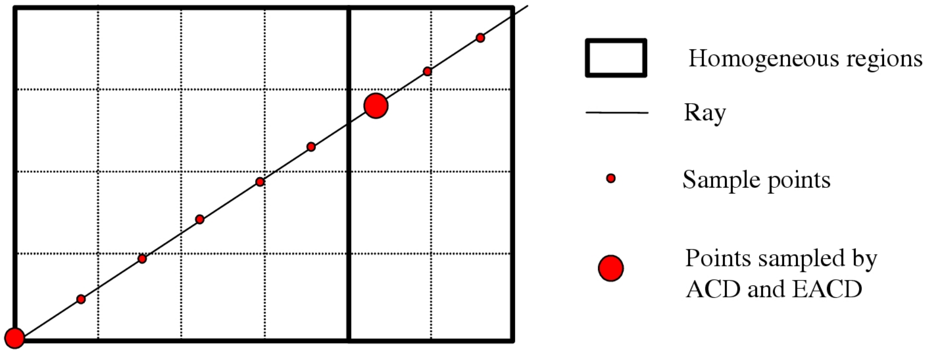


Figure 4.1: ACD and EACD sample points are aligned with the primitive sample points of a primitive ray caster. In the shown case, homogeneous region leaping requires just 2 samples as opposed to 9 samples needed by a primitive ray caster.

is calculated as the ceiling of the ratio of the distance value to the constant step size. For each method, both the volume data and the distance field are represented by 3D textures located on the GPU memory. Inside a homogeneous region, a ray can step forward as long as the distance value of the current voxel.

4.3 EACD Ray Casting

Note that, although eight distance values form an anisotropic shape around the voxel in ACD, the homogeneous regions defined by each of these values is cubic (assuming unit voxel dimensions). EACD extends this structure in such a way that it allows the definition of non-cubic homogeneous regions. We observe that non-cubic regions may reduce number of traversal steps considerably as shown in Figure 4.1.

The original traversal algorithm we developed in [25] and detailed in Chapter 3 essentially addresses ray tracing. The ray traversal algorithm is adapted for volume ray casting. In ray tracing, rays skip empty regions fully through the border of the next region. In order to find next position, the ray is intersected with the border planes defined by the homogeneous region. The intersection point giving the minimum parametric distance is selected as the next ray position.

On the other hand, volume ray casting with homogeneous region leaping require some adjustments to this algorithm. The image rendered with EACD should be no different than the image of primitive ray caster. For that reason, contrary to [25], the next position of a ray during the traversal is aligned with the sample points of the primitive ray caster s shown in Figure 4.1. The aligned position can be computed by dividing the homogeneous region distance to the constant ray step size. The ceiling of this division gives how many samples a ray can safely skip. The ray is advanced by the number of ray step times the constant step size. This scheme works even if the homogeneous region is only one voxel. Note that, we use point sampling (nearest neighbor) instead of trilinear filtering. In case of trilinear filtering, opacity and normal values may change in the border voxels of a region due to the interpolation. Since the optical properties of inner voxels are very similar, fetching one sample from this region is still sufficient for the ray integral factorization.

4.3.1 Implementation

The algorithm is relized using OpenGL 2.0 and Cg 1.4 toolkit [60] with FP40 profile. Our GPU allows for dynamic branching and long kernels. Therefore, the whole ray casting operation can be executed within a single fragment program.

The volume data contains a normal and opacity value for each voxel. Data is stored as a 4-component 16-bit floating point formatted 3D texture. Similarly, the distance field is stored in another 3D texture. In order to reduce memory requirements, 16- bit packed color format (3×5 bits RGB, 1 bit alpha) is used for EACD texture. Since EACD and ACD utilize different distance values for each direction octant, the distance field texture is enlarged by a factor of 2 along each direction. As a result, for each data voxel, there are 8 corresponding ACD/EACD voxels.

Prior to ray casting, a ray generator program is run to create and clip rays to the bounding box of the scene. As for the output, ray generator creates ray origin and direction textures. Rays intersecting the scene are then transformed into the volume coordinate space (integer coordinate values define voxel corners). By this way, the voxel indices of a point can be easily computed by taking the floor of its coordinates.

The Cg like pseudo-code for the EACD volume ray casting kernel is given in Figure 4.2. Variable types are explicitly given in order to reveal the vectorized operations in the algorithm. The code does not include any data dependent branching inside the main traversal loop. ACD traversal is almost identical to EACD. The only difference is that; instead of three distance values, one value is fetched from the distance texture. In the pseudo-code, `tex_direction`, `tex_origin` are 2D ray textures created by the ray generator. `tex_volume` is the 3D volume data texture which contains voxel normals and opacity values, while `tex_distance` is the 3D distance field texture as explained previously in this section. `ray_idx` is the ray index which is actually 2D texture coordinates to the ray textures. `t_step`

```
float4 RayCast( Sampler2D tex_direction, tex_origin,
                Sampler3D tex_volume, tex_distance,
                float2 ray_idx,
                float t_step,
                shadingParameters shading_param )
{
    //read ray data
    float3 D_ray = tex2D(tex_direction,ray_idx);
    float3 O_ray = tex2D(tex_origin,ray_idx);
    //initialize final color
    float3 C_final = 0;
    float A_final = 0;
    //loop while the ray is not fully opqaue or inside the volume
    while(A_final<1 && isInVolume(O_ray))
    {
        //read voxel data
        int3 vox_coords = floor(O_ray);
        float3 normal = tex3D(tex_volume,toTexCoords(vox_coords)).rgb;
        float A_sample = tex3D(tex_volume,toTexCoords(vox_coords)).a;
        //read the appropriate distance value based on the ray direction
        int3 distance = tex3D(tex_distance,getDistanceVoxelCoords(O_ray,D_ray));
        int3 region_borders = computeRegionBorders(vox_coords,distance);
        //compute ray/region border intersections
        float3 t_borders = intersectRayWithRegionBorders(O_ray,D_ray,region_borders);
        //find the minimum parametric distance
        float t_distance = min(t_borders.x, min(t_borders.y,t_borders.z));
        //compute the number of ray steps
        int n = ceil(t_distance/t_step);
        //compute the integrated region color and alpha
        float A_region = (1-pow(1-A_sample,n));
        float3 C_region = A_region* shade(O_ray,D_ray,normal,shading_param);
        //accumulate color & alpha
        C_final += C_region*(1-A_final);
        A_final += A_region*(1-A_final);
        //advance ray
        O_ray += D_ray*(n*t_step);
    }
    return float4(C_final, A_final );
}
```

Figure 4.2: Cg like pseudo-code for the EACD volume ray casting. Note that most of the operations work on vectors.

is the parametric distance for the constant ray step. A primitive ray caster always moves rays by this distance in each step. Finally `shading_param` is the structure keeping the shading parameters such as light position, diffuse and specular colors as well as the material colors. The output of the kernel is the accumulated color of the ray.

This code needs some minor adjustments for trilinear filtering. In that case, the calculation of the number of ray steps should be altered in such a way that, the ray should take constant step in border voxels (voxels neighboring to a different region). Border voxels can be easily determined by looking at the distance value (i.e. if any component of the distance vector is 0, it is considered as the border voxel). The resulting code should be like below:

$n = IsBorderVoxel(distance)\ ?\ 1 : floor(\ t_{distance}\ /\ t_{step}\ )$

4.3.2 Construction of Homogeneous EACD Regions

Construction of homogeneous EACD regions is almost identical to the one described in Section 3.3. Therefore, we only briefly describe the operation and give the differences in this section.

A heuristic with a simple greedy search is used in order to create EACD grid. The heuristic is based on finding the largest homogeneous region per voxel basis. We rely on the ACD grid and extend regions along the main axes to construct the homogeneous EACD regions. As a result, building the acceleration structure involves two phases. In the first phase ACD homogeneous regions are created. Firstly, the distance values of the cells are initialized to infinity. Then, ACD kernels are applied as described in Figure 3.9. In this procedure, the distance value of the overlaid element is computed using the following algorithm:

$$\mathrm{Dist}(\ \mathrm{V}_{\mathrm{i,j,k}} \text{ due to } \mathrm{V}_{\mathrm{x,y,z}}\ ) = \begin{cases} \mathrm{Dist}(\mathrm{V}_{\mathrm{i,j,k}}) & \text{,if inSameGroup}(\mathrm{V}_{\mathrm{x,y,z}},\mathrm{V}_{\mathrm{i,j,k}},\mathrm{T}_{\mathrm{o}},\mathrm{T}_{\mathrm{a}}) \\ 0 & \text{,otherwise} \end{cases}$$

Where $\mathrm{V}_{x,y,z}$ are the voxel coordinates of the current center voxel and $\mathrm{V}_{i,j,k}$ are the voxel coordinates for the overlaid element. Function *inSameGroup* reads the opacities and the surface gradients of $\mathrm{V}_{x,y,z}$ and $\mathrm{V}_{i,j,k}$. $\mathrm{V}_{i,j,k}$ is classified as belonging to a different region than $\mathrm{V}_{x,y,z}$, if the opacity difference is greater than the opacity threshold, T_o, or the angle difference is greater than the angle threshold, T_a. We give a different id to each separate region. As a result, we do not classify regions as empty or non-empty, as in case of ray tracing of geometric data. Instead, regions are arranged as the groups of voxels with similar optical properties (i.e., voxels with the same region id).

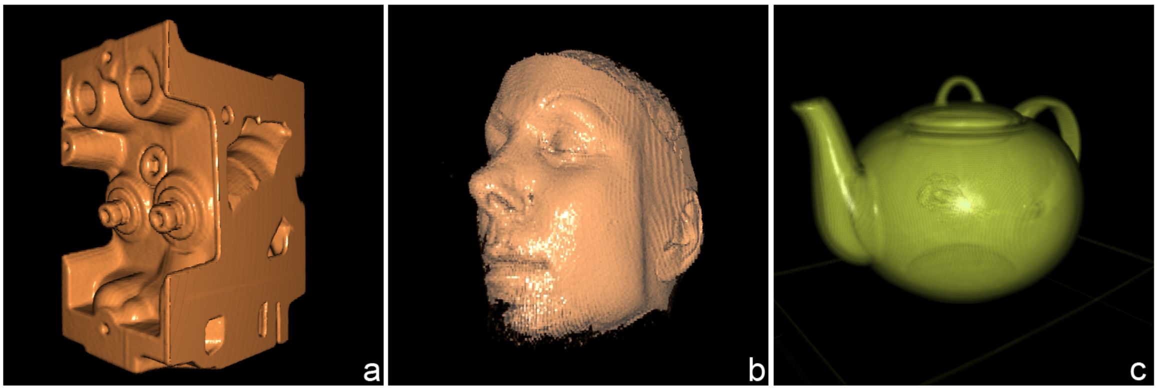


Figure 4.3: Tested volume datasets (a)engine, (b) mrbrain, (c) teapot

In the second phase, we first create six axial distance grids representing the range of voxels belonging to the same group in line along the $\pm x$, $\pm y$ and $\pm z$ axis directions. In order to define EACD homogeneous regions, we determine how much the cubic homogeneous regions can be extended by using the auxiliary axial distance grids and ACD field in a greedy manner. This operation is done for each voxel. The extension procedure is carried as follows: Border voxels of the homogeneous ACD regions are scanned and maximum possible extensions along the main axes are computed. The cubic region is then extended along the axis giving the maximum volume. This step is repeated once more, for one of the remaining two axes which is giving the maximum volume.

If tri-linear interpolation is to be be used during rendering, region grouping operation requires a minor modification. In this case, border voxels are marked prior to the creation of ACD grid. During the region classification, voxels marked as the border voxels are always considered as belonging to a different region.

4.4 Results And Discussion

A number of well known volume datasets were used for testing the algorithm. The rendered images are shown in Figure 4.3. Tests were run on a 512MB GeForce7800 GTX graphics board. The frame size for the rendered images is 512×512.

As seen in Table 4.1, the speedup compared to ACD is around 25%, while it is as much as 700% compared to the primitive ray caster. EACD is especially advantageous over ACD if the volume is composed of many non cubic homogeneous regions. Since both ACD and EACD use the same algorithm essentially, their performances should be equal in the worst case (assuming memory bandwidth is not the bottleneck). On the other hand, for EACD,

Table 4.1: Performance results of the ray casting methods. Results are in milliseconds. Last column is the speedup achieved by EACD compared to primitive ray casting.

Data	R.Cast	PC	ACD	EACD	Speedup
engine	183	83	42	33	554%
mrbrain	206	101	51	41	502%
teapot	346	121	62	49	706%

maximum distance limit of 32 (due 5 bits imposed by the low precision distance texture format) may cause some performance penalty in very large homogeneous regions. But none of the volume datasets we experimented on revealed such a problem. As an empty space skipping technique, PC traversal does not perform as fast as EACD or ACD. This is largely the result of shorter ray steps caused by the isotropic regions. The loop body of the primitive ray caster is fairly tight and efficient. However, it performs the worst compared to region leaping methods, since it needs to sample many more points .

Figure 4.4 illustrates the average number of loops performed per ray to complete the rendering. Brighter regions indicate higher loop counts. It is clearly seen from the image that ACD and EACD require considerably lower loop counts than PC. Among all methods, EACD can pass through the volume with the least average number of loops. EACD traversal step counts are generally 10% to 30% lower than ACD.

The opacity and angle thresholds T_o and T_a are used to control the size of homogeneous regions. As threshold values increase, larger homogeneous regions are formed and larger optical variation within the regions is allowed. Although the rendering speed increases due to larger regions, it causes more artifacts on the rendered image. The T_o and T_a parameters can be adjusted to compromise between the image quality and rendering time. Note that if the volume data has many empty regions, huge speedup may be achieved even for the low values of the parameters. Figure 4.5 shows the images rendered with several different T_o and T_a threshold values.

The problem of ACD and EACD is, the acceleration structure is eight times as big as it is for PC and for a primitive ray caster. Since the maximum allowed 3D texture resolution is 512^3 in our test hardware, the largest dimensions of the volume data can be 256^3. As the size of the local graphics memory enlarge and hardware capabilities enhance, this will be less of a concern. As of date, the latest GPUs can utilize 3D textures with the dimensions of as big as 2048^3. This is 64 times greater than the testing hardware. Still, a better solution to

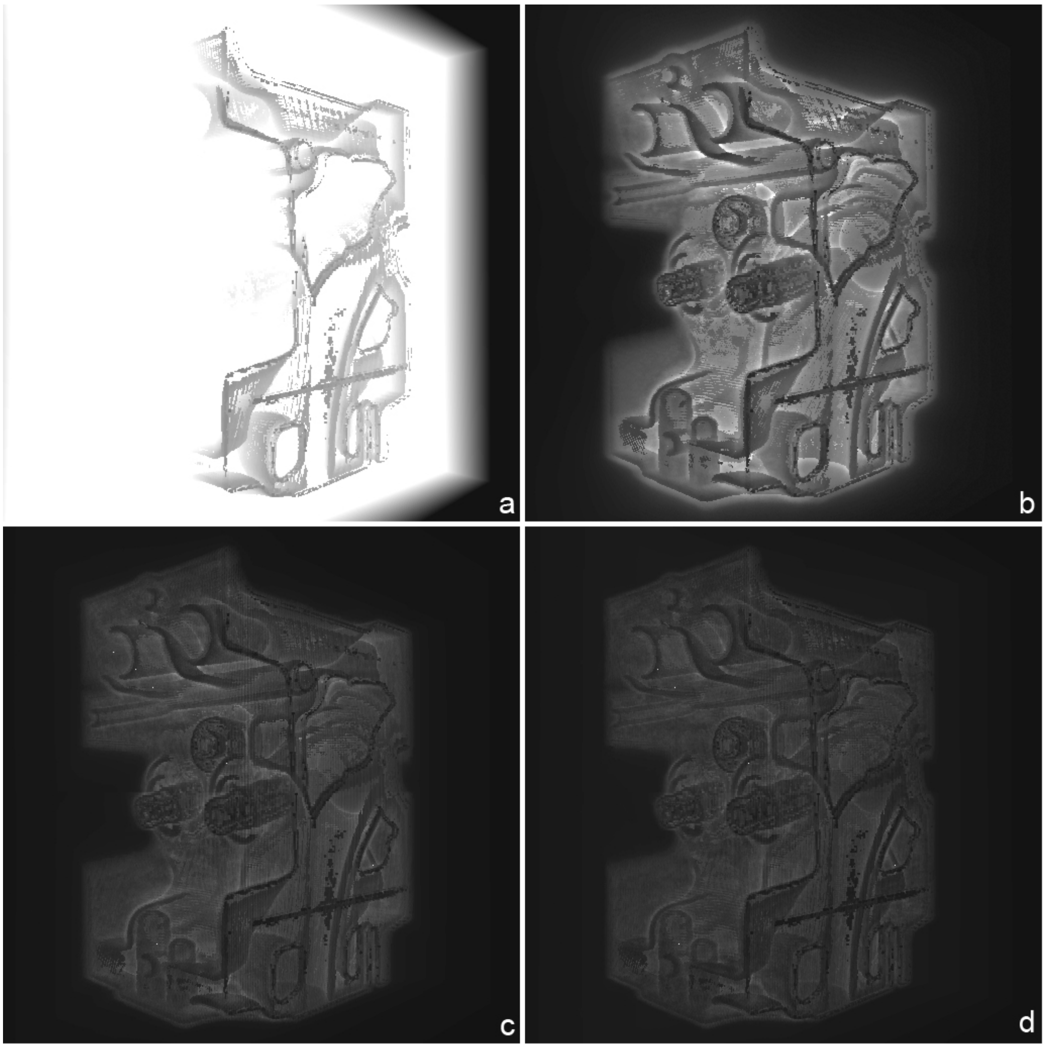


Figure 4.4: Illustration of loop counts for (a)Ray caster, (b)PC, (c)ACD, (d)EACD. Brightness and contrast is adjusted for visual clarity. Darker regions indicate lower loop counts.

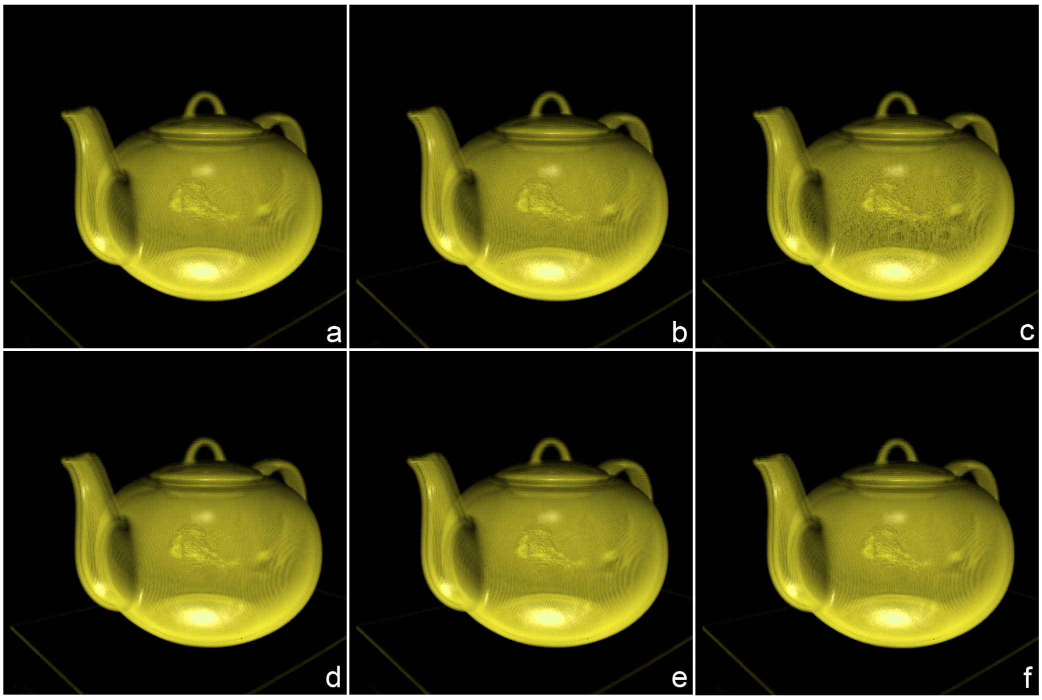


Figure 4.5: Teapot images rendered with ACD using different segmentation thresholds. (a)T_o:0.01, T_a:3, (b)T_o:0.02, T_a:3, (c)T_o:0.1, T_a:3, (d)T_o:0.01, T_a:1, (e)T_o:0.01, T_a:10, (f)T_o:0.01, T_a:20

this problem may be to explore hybrid partitioning structures or using some GPU friendly data compression method.

CHAPTER 5

GPU ACCELERATED STEREOSCOPIC RAYTRACING

Giving the sense of depth from two dimensional pictures is a well studied problem in virtual reality. Among the solutions, using a pair of images prepared for each eye (stereoscopic) is the most widely used technique due to the simplicity and effectiveness. Today, majority of the virtual reality applications depends on stereoscopic images. In real life a pair of cameras, one for left eye and one for right eye, are used for capturing stereo images. Analogously, in virtual reality two virtual cameras are used and the scene is rendered from their viewpoints for the stereoscopic vision.

Rendering a scene twice, and thus doubling the frame time is the major drawback of stereoscopic rendering. If the scene is complex or expensive rendering methods are used, doubling the frame time might not be feasible. Especially in virtual reality, where convincing images should be rendered in interactive rates, slow frame rates may destroy the sense of immersion experienced by the user. Similarly, offline rendering applications may suffer from the same problem, where time required to render stereoscopic frames separately may be infeasible due to time or budget constraints.

In this chapter we focused on efficient stereoscopic ray tracing on the GPU. In order to demonstrate the technique, we developed a Whitted style brute force recursive ray tracer that fully runs on the GPU. The utilized stereoscopic reprojection method is based on Adelson and Hodges' work [1]. However, although their reprojection algorithm is suitable for CPUs; it includes several difficulties and inefficiencies for GPU rendering. In this chapter, we show how the reprojection technique can be mapped to the parallel streaming model of the GPU efficiently. Note that although it is possible to use the introduced method for offline rendering, our main concern is GPU based real-time stereoscopic ray tracing.

The rest of the chapter is organized as follows: In the second section, related previous work is given. In the third section, we briefly describe the GPU based ray tracing. Image reprojection is explained in the fourth section. The following section describes how the problem is efficiently mapped to GPU programming model. Additionally, several variations of the technique are described. Test results and discussion are given in the sixth section.

5.1 Previous Work

Image reprojection is a kind of image based rendering technique. Essentially, a rendered or captured image from a certain viewpoint is used as the reference image; and the pixels of the reference image are reprojected onto the image plane of another viewpoint to generate the image from that viewpoint. By utilizing the image space temporal coherence, reprojection can be used in conjunction with ray tracing to accelerate the rendering the frames of an animation sequence. Stereoscopic rendering may be thought as a special kind of a two-frame animation sequence; in which only the viewpoint is shifted slightly for the second eye while the rest of the scene remains still. Therefore reprojection is well suited to the rendering of stereoscopic images. Since reprojection time is constant and does not depend on the scene complexity, proper reprojections can save huge amount of time during the rendering of the second image.

In this chapter, we limit our scope to the ray tracing of stereo image pairs. Badt was the first to use pixel reprojection to accelerate ray tracing of animation sequence [48]. In his work, first hit positions of the eye rays are preserved and projected to the viewpoint of the next frame. Then, the projected image is searched by using a 5×5 box filter for possible problems, and only the areas considered to be suspicious are ray traced. His method can be used to render the stereoscopic images. Similarly, Adelson and Hodges [1] used reprojection to build warped image of the right eye from the left eye image. However, instead of blocks, they search possible reprojection errors by scan line order. Their method guarantees to detect problematic reprojections. Only those problematic spans are ray traced for the right eye. They reported that, up to 93% of the left eye pixels can be successfully reprojected to the right eye without further need to ray trace. Later on, they proposed another algorithm that can accelerate the ray tracing not only of stereoscopic images but also of generic animation sequences by using reprojections [2]. On the other hand, the algorithm requires a space partitioning structure or bounding volumes, and restricted to convex objects. In a more recent work, Havran et al. exploited temporal coherence to speed

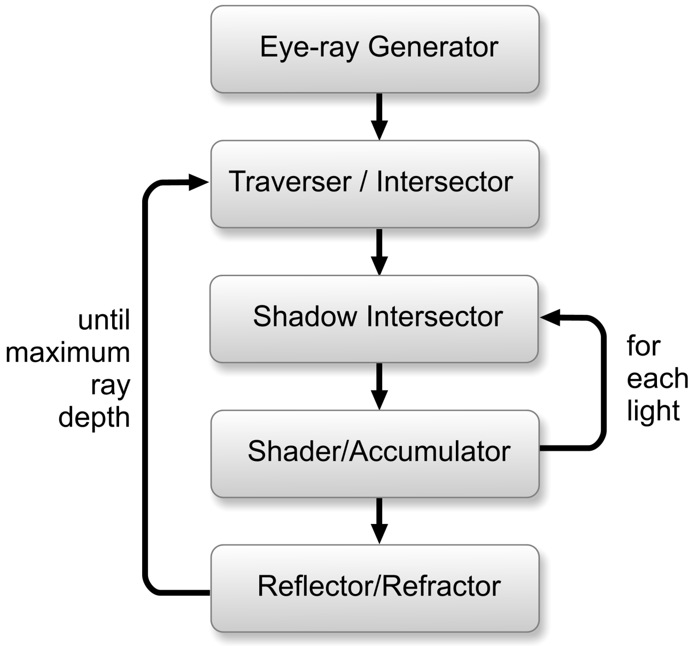


Figure 5.1: Kernels for the ray tracer.

up rendering of ray casted walkthroughs [40]. Their work can handle generic animation frames with different viewpoints. They classify possible reprojection errors and propose methods for verification of each kind of error.

5.2 The Ray Tracer

In order to test stereoscopic image generation, we implemented a GPU based ray tracer. Beacuse the aim of this chapter is to devise efficient GPU based reprojection methods; no acceleration structures or advanced shading methods are implemented. The ray tracer only supports spheres, infinite planes, point light sources and phong shading. However extending it to support other types of primitives, light models and shading methods is straightforward.

The kernel flow of the ray tracer is given in Figure 5.1, which is very similar to the kernels of our EACD ray tracer (see Chapter 3). However, since no acceleration structure is employed, traverser and intersector kernels are unified to one kernel. As seen in the figure there are five main kernels. All of these kernels are executed by the fragment processors. All kernels write their output to screen sized textures. By this way, the output of a kernel can be used by subsequent kernels. Each pixel of the texture represents a ray and keeps information of the ray passing through that pixel. The kernels operate on every pixel (i.e. every ray) of the screen. In order to execute a fragment program on all pixels (rays) a screen sized quad is sent to the GPU. The rasterizer scan converts the quad, and generates a fragment to be processed by the fragment processors, for each screen pixel.

Kernels are very similar to the ones described in Chapter 3. The first kernel is *Eye-ray generator*. It generates eye rays passing through the pixel centers by using the view projection matrix of the camera. As the output, ray origin and ray direction textures are generated. These textures are used by the subsequent kernels to read the corresponding ray information. The second kernel is *Traverser/Intersector*. It is responsible for ray-object intersection testing. Scene database is sent in a scene database texture. This texture keeps sphere positions, radii and corresponding material information. Since no acceleration structure is employed, all rays are tested against all spheres. The output of this kernel is written to intersection position and normal textures. The third kernel, *Shadow Intersector*, gets the intersection results of *Traverser/Intersector* along with a light position, and tests if the intersected position is in shadow with respect to the light source. The shadow test results are written to a shadow texture. Pixel value of the shadow texture is 1 if the intersected position is in shadow or 0 otherwise. The next kernel is *Shader/Accumulator*. It uses shadow texture, intersection position and normal textures, material information, and light parameters to illuminate pixels. The results are accumulated to a color buffer texture, which represents the final rendered image. Note that, *Shadow Intersector* and *Shader/Accumulator* kernels should be called once for each light source. The last kernel is *Reflector/Refractor*. Based on the material properties of the intersected positions, it reflects or refracts rays. All ray tracing operations are done over the whole screen in parallel. Therefore, during ray tracing a texture stack which stores previous results is used to create buffer trees instead of ray trees: Intersection and ray information textures are pushed to the stack before *Reflector/Refractor* kernel takes place. *Reflector/Refractor* writes the reflected or refracted ray information to the ray information textures, and the execution continues with *Traverser/Intersector*. Old textures are popped from the stack when the reflected rays are shaded and done.

5.3 Stereoscopic Ray Tracing by Reprojection

In order to obtain a stereo image pair, viewpoints are offset laterally from each other (shifted horizontally). For obtaining geometrically correct stereo pairs, view directions (camera-target vectors) should be parallel. For a balanced stereoscopic view, it is required to locate projection plane in zero parallax. That is, any geometry residing on the projection plane should project to the same screen location for both left and right eyes. As shown in Figure 5.2,

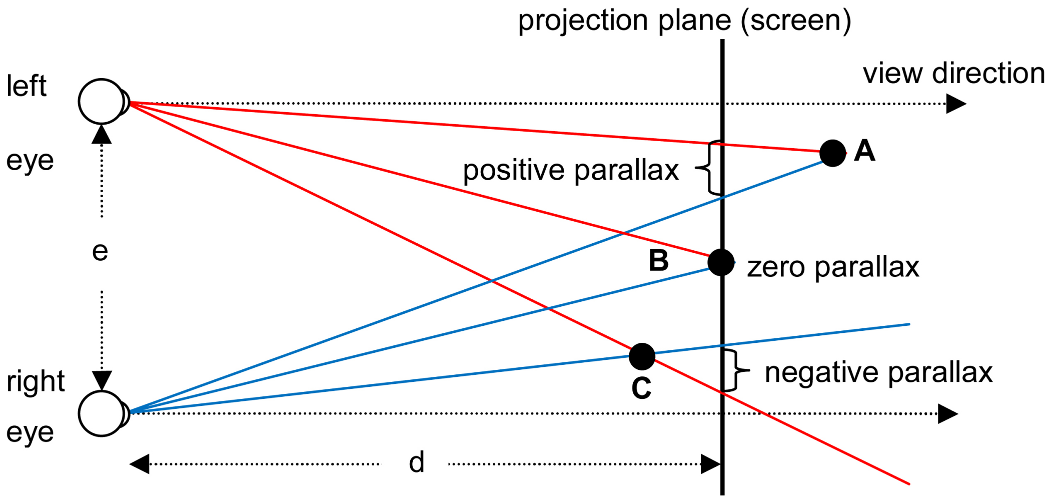


Figure 5.2: 2D illustration of stereoscopic perspective projection. e is eye separation distance. d is focal length. Point A is behind the projection plane, thus projects to positive parallax. C is in front of the projection plane which projects to negative parallax. Since B is on the projection plane, it is projected to zero parallax.

any geometry closer than the projection plane will project to negative parallax, while the geometry behind the projection plane will project to positive parallax. In order to create the view-projection matrices, right and left eye positions are offset laterally from each other and their view frustums are corrected asymmetrically so that viewports fully overlap each other at the focal length. This kind of stereoscopic camera setup is known as off-axis projection. Note that off-axis projection does not introduce vertical parallax.

An OpenGL implementation of the off-axis stereo projection is given in [9]. For the sake of brevity we omit the derivation of the matrices and assume that the image of the left eye is rendered by using M_{left}, while the right eye is rendered by using M_{right}, which are world-to-screen space transformation matrices for left and right eyes respectively. It is possible to find the reprojected right image location of a pixel, when the depth of the pixel (z distance from the viewpoint) is known. The reader can refer to [1] for the derivation of the reprojection equation. Similarly, a pixel can be reprojected onto right eye by using M_{right} if the 3D world coordinates of the point is known. Since world positions of the intersected pixels are already known in the left image, we simply transform these positions by M_{right} to calculate reprojected pixel locations on the right image.

On the other hand reprojection can cause some image artifacts. As described in [1] three types of pixel errors are possible. These are overlapped pixels, missing pixels and bad pixels.

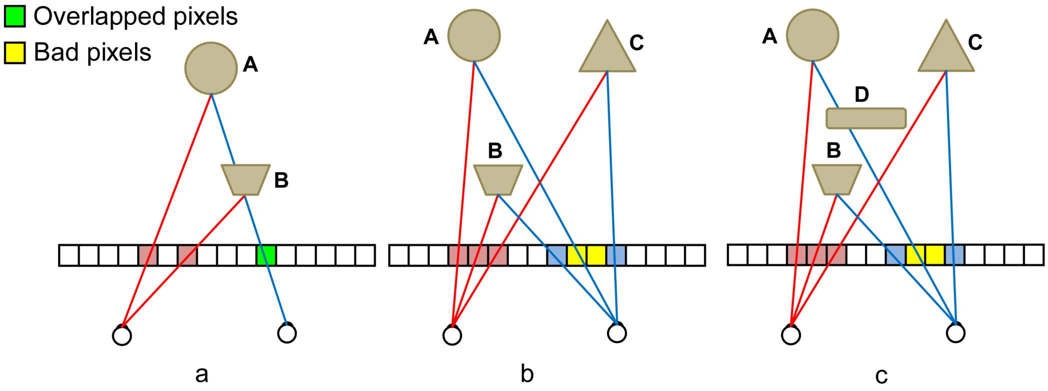


Figure 5.3: Reprojection problems. (a) Overlapped pixel and (b,c) bad pixel problems.

Overlapped pixel problem occurs when multiple reprojections are done onto the same pixel. The correct reprojection value should be resolved in this case. Adelson and Hodges show that this problem can be easily solved if reprojections are done beginning from the leftmost pixel to the right, and overwriting any previous reprojections. That is, only the last value is kept in case of multiple reprojections. The missing pixel problem happens when no reprojections are done onto some of the pixels. Since there is no information on those pixels, they should be ray traced to fill the missing values. Bad pixel problem occurs if two adjacent pixels on a scanline are reprojected to the screen locations more than one pixel away from each other. In this case, there is a gap of at least one pixel long between the reprojected locations. The validity of the values on these gaps are questionable, and therefore they should be ray traced. Overlapped and bad pixel problems are illustrated in Figure 5.3. In Figure 5.3-a, intersection positions which project to different locations on left eye are reprojected to the same pixel on right eye, and constitute overlapped pixel problem. Figures 5.3-b and 5.3-c depict bad pixel problem. Adjacent pixels of left eye are reprojected to non-adjacent positions leaving a gap. In Figure 5.3-b object A should be seen through this gap, while in Figure 5.3-c it is blocked by object D and should not be seen. However one can not determine if object A is seen or not through the gap just by looking at the reprojection. For that reason, the values on the gap are considered as invalid.

In order to rule out reprojection errors, left image is ray traced by scan line order, left to right fashion. Status of all right image pixels are set to `NO_HIT` initially. As left image pixels are ray traced, they are reprojected to the right image. Reprojected pixels in the right image are marked as `HIT`. If gaps are detected between any adjacent reprojected locations, the gap is marked as `NO_HIT`. After a scan line is done, pixels marked as `NO_HIT` are ray traced for the right image. The modified pseudo code of the algorithm is given in Figure 5.4. Note that, this algorithm can generate correct reprojections for visible surface ray tracing. However, it is not possible to reproject reflected/refracted pixels properly with this method.

5.4 GPU Based Stereoscopic Reprojection

Our GPU based stereoscopic reprojection takes place after *Shader* kernel is run. Intersection positions, normals and per-light shadow information are reprojected to be able to shade right pixels accurately. Especially for specular surfaces, slight variation of eye position may change the shading obviously. Therefore, instead of simply copying color values of the left image, new color values are calculated from scratch. In the current implementation only eye rays and the corresponding shadow rays are considered for reprojection. Reflection/refraction passes are not reprojected, but ray traced from scratch for both eyes. If the scene is not highly reflective this is not a big problem, since only a small and decreasing amount of rays are reflected. We propose an approximate solution to the reprojection of reflection/refractions in the discussion section at the end of this chapter.

Original stereoscopic reprojection algorithm given in Figure 5.4 includes some problems for GPU based ray tracing. Firstly, it requires strict ordering of rendered pixels (left to right fashion). However, GPUs do not guarantee the processing order of the rasterized pixels if they belong to the same primitive. Therefore, the algorithm should be modified for parallel rendering of rays. Secondly, the screen location of the pixels should be altered dynamically during reprojection, which is a dynamic scattering problem. Moreover in case of bad pixels, more than one reprojected pixels should be marked as `NO_HIT`. However, the fragment processors of GPUs can only write to a single screen location. Eventually, stereoscopic reprojection maps to one-to-many dynamic scattering problem.

In order to solve the first problem, reprojection phase is separated from the ray tracing phase: Whole screen is ray traced in the first pass, and then reprojected in a second pass. Left to right ordering of reprojections are guaranteed in the second pass by sending a point or line primitive for each pixel from left to right.

In order to solve the reprojection problem, we transformed scattering operation (writing to multiple computed addresses) to gather operation (reading from multiple computed addresses). Note that, GPUs can read from multiple arbitrary texture locations without any limitation. To facilitate scatter-to-gather transformation, dynamic scattering operation is decomposed into to three phases. In the first phase, screen locations of reprojections are

```
for(scan_line=0; scan_line<screen_height; ++scan_line )
{
    //clear hit status
    for{pixel=0; pixel<screen_width; ++pixel}
        hitStatus[pixel] = NO_HIT;

    //ray trace left eye & reproject to right eye
    prev_right_pixel = -1;
    for(pixel=0; pixel<screen_width; ++pixel)
    {
        //ray trace left image pixels
        ray = fireRay(left_eye_position,scan_line,pixel);
        traceResult = traceRay(ray);
        int_pos = getIntersectionPosition(traceResult);
        int_normal = getIntersectionNormal(traceResult);
        setLeftColor(scan_line,pixel, shade(int_pos,int_normal,left_eye_pos,
            light_params) );

        //compute the reprojected pixel position on the right image
        right_pixel = projectToRightEye(getIntersectionPosition(traceResult),
            M_right);
        if (right_pixel < screen_width)
        {
            //handle reprojection problems
            if (prev_right_pixel-right_pixel > 1)
                for (invalid=prev_right_pixel; invalid<right_pixel; ++invalid)
                    hitStatus[invalid] = NO_HIT;

            hitStatus[right_pixel] = HIT;
            //shade the reporjected right pixel
            setRightColor(scan_line,right_pixel, shade(int_pos,int_normal,
                right_eye_pos,light_params) );
            prev_right_pixel = right_pixel;
        }
        else
            prev_right_pixel = screen_width-1;
    }

    //ray trace the remaining right image pixels
    for{pixel=0; pixel<screen_width; ++pixel}
        if (hitStatus[pixel] == NO_HIT)
        {
            ray = fireRay(right_eye_position,scan_line,pixel);
            traceResult = traceRay(ray);
            int_pos = getIntersectionPosition(traceResult);
            int_normal = getIntersectionNormal(traceResult);
            setRightColor(scan_line,pixel, shade(int_pos,int_normal,
                right_eye_pos,light_params) );
        }
}
```

Figure 5.4: Stereoscopic reprojection algorithm

calculated and written to a scatter table. In the second pass, a point or a zero length line is sent to pipeline for each pixel and repositioned according to the scatter table. As a result of this phase a gather table is created. Gather table keeps the original pixel locations (prior to the reprojection) so that the correct left eye information can be copied to right; and the information of whether the reprojection is valid (`HIT`) or not (`NO_HIT`). In the third phase, intersection positions, normals and shadow results are reprojected to right eye by using the created gather table. Scatter and gather tables are actually 2D textures. The third phase is always performed on GPU. However, in order to create the gather table we devised three methods. One of them uses CPU to create the table, while other two create it fully on GPU. As demonstrated in the section, full GPU approach performs an order of magnitude faster than CPU.

The third phase is always performed on GPU. However, in order to create the gather table, we followed three approaches. One of the approaches uses CPU to create the table, while two others create the table fully on GPU. As demonstrated in the test results, full GPU approach performs much faster than CPU method.

5.4.1 CPU Created Gather Table

A naive way to attack the scattering problem is to transfer left image data from the GPU memory to CPU memory and to perform reprojections on the CPU. Since CPU can write to arbitrary memory locations without any problem, multiple dynamic scattering is trivial to implement. The reprojected values can then be sent back to GPU memory. However, this approach is sub-optimal. Transferring data between the GPU and CPU memory is (relatively) slow, and should be minimized. Time required for frequent transfers may pose a bottleneck and can increase frame time considerably. Reprojection of the intersection positions, normals and shadow results requires all of these data to be transferred to CPU memory and then sent back to the GPU memory after reprojection.

So as to reduce memory transfers, CPU only reads intersection positions from the GPU memory, creates gather table and sends the table back to the GPU. That is, in case of CPU based reprojections; first phase (scatter table creation) is skipped, and second phase (gather table creation) is carried out by the CPU.

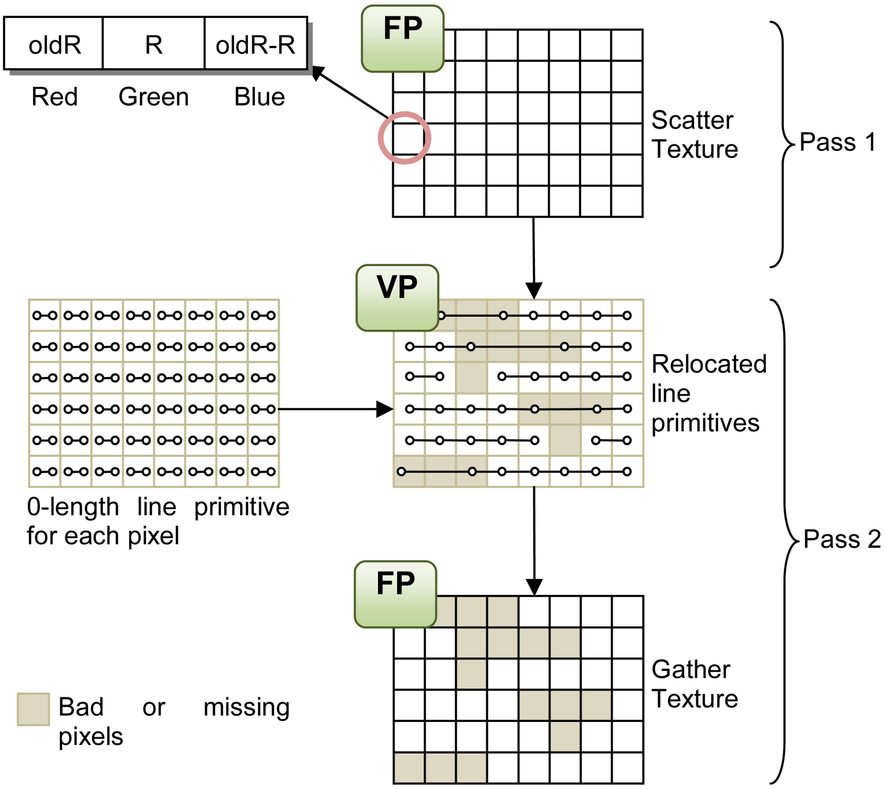


Figure 5.5: Two-Pass GPU based gather table creation. VP denotes data processed by vertex processors, while FP denotes data processed by fragment processors. *oldR* and *R* holds for `prev_right_pixel` and `right_pixel` respectively as calculated in Figure 5.4.

5.4.2 Two-Pass GPU Gather Table

Two-pass GPU based gather table creation consists of the first two phases of scatter-to-gather transformation as described above. In the first pass, a scatter texture is generated by means of a fragment program. A screen sized quad is sent to the GPU so that all image pixels are processed in this pass. As shown in Figure 5.5, scatter texture is an RGB texture; where *red* keeps the x component of the reprojected screen coordinates of the previous pixel, *green* keeps the x component of the reprojected screen coordinates of the current pixel and *blue* keeps the difference between the two. In the second pass, zero-length line primitives are sent to the pipeline for each screen pixel (from left to right fashion). A vertex program processes these lines by relocating line start and end positions according to the scatter texture. At the same, time a fragment program writes the gather texture according to the information coming from the relocated line stream. Contrary to the CPU based approach, this method fully runs on the GPU, and does not require expensive CPU-GPU data transfers. Note that,

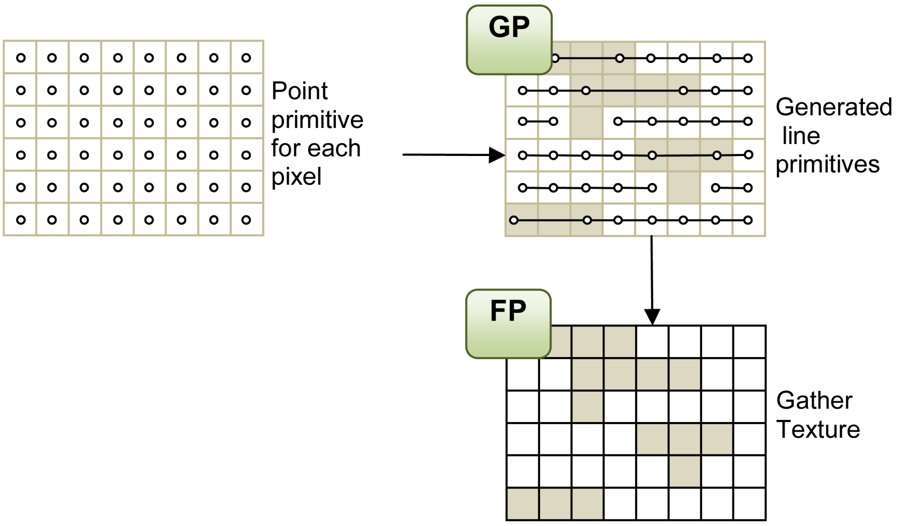


Figure 5.6: One-Pass GPU based gather table creation. GP denotes parts of data processed by geometry processors, while FP refers to fragment processing.

used vertex program should be kept simple (it relocates vertices just by looking up the scatter table in our implementation), since it is executed for both vertices of the line primitives (i.e. run twice for each pixel). It is the main reason why relocations are calculated in a previous pass by the fragment processors (i.e., for processing each pixel once). Moreover, fragment processors can run much faster than vertex processors on the majority of GPUs.

5.4.3 One-Pass GPU Gather Table

This method also runs fully on the GPU. This time, geometry processors found on the latest GPU generation are utilized. By using geometry processors, first phase (scatter table generation) can be skipped. Since geometry processors can emit new geometry (line strips in our case) to the pipeline with dynamically computed positions, scatter texture is not necessary. Namely, geometry processors can directly generate relocated line primitives. As illustrated in Figure 5.6, geometry program gets a point primitive instead of a line, and spawns a new, relocated line for each point.

Similar to the two-pass gather table generation, vertex processors could be used instead of geometry processors in this method. However, since vertex processors need a line primitive for each pixel, this would double the relocation calculation time. Especially for the previous GPU generations, where vertex processors are not as powerful as fragment processors and there are no geometry processors, two-pass approach should be used.

5.5 Results and Discussion

We tested the reprojection techniques on our GPU based interactive ray tracer. Since the reprojection accelerates visible surface ray tracing (eye rays and the corresponding shadow rays), we tested the scene with and without reflections. There are 100 animated sphere primitives and four light sources in the test scene. 46 of the objects are reflective. The scene database is written into a 2D texture and updated each frame. Stereoscopic anaglyph rendering of the test scene is shown in Figure 5.7. The test platform consists of an Intel Core2Duo 2.4 GHz CPU and nVidia 8800GTX GPU with release 162.18 graphics drivers. OpenGL is used for the implementation. Vertex and fragment programs are implemented by using Cg shading language, while geometry programs are implemented with OpenGL shading language. The scene was tested at three different frame sizes.

The separate and reprojected stereoscopic rendering results are given in Table 5.1. Note that reprojection saves significant amount of time on the right image. Especially in low resolutions, *RGPU1* and *RGPU2* results are very close. The gather table creation times for reprojection methods are listed in Table 5.2. Obviously, GPU based gather table creation is more than 10 times faster than CPU. GPU based one-pass and two-pass approaches performed almost identically. However, at high frame resolutions, one-pass GPU gather table method is faster than the two-pass method. That is because of the fact that, at high resolutions, large number of points are processed twice by GPU in two-pass method (line primitive instead of point primitive for each pixel) which may overwhelm the vertex processors.

5.5.1 Missing Object Problem in Reprojections

The stereoscopic reprojection algorithm guarantees the correct image if all points seen by the right eye are also seen by the left eye. However, there may be some cases such that an object, or a part of it, is only visible to right eye. In this case, since there is no information in the left image about the object, it cannot be reprojected to the right eye. A GPU friendly approach to solve this problem is to determine objects which are intersected by the viewing frustum of the right eye but not by the viewing frustum of the left eye. If such objects are found, they are drawn on the screen by traditional rasterization based rendering after the reprojection in order to mark their screen area as `NO_HIT`. Rasterization is very fast on GPUs (much faster than ray casting), and a simple fragment program to mark pixels is enough during the drawing operation. The bounding volumes or convex hull of the objects can be used instead of the original model if the object is complex.

Table 5.1: Frame times of Monoscopic (MONO), Separate stereoscopic (SS), Reprojected with CPU based gather table (RCPU), Reprojected with two pass GPU based gather table (RGPU2), and Reprojected with one-pass GPU based gather table (RGPU1). Visible: Visible surface ray tracing, FullRT: Full ray tracing with maximum ray depth of 3. Times are in milliseconds.

Frame Size		**MONO**	**SS**	**RCPU**	**RGPU1**	**RGPU2**
1024×768	Visible	147	294	250	175	186
	Full RT	194	388	345	271	282
800×600	Visible	91	181	154	109	108
	Full RT	120	242	216	172	171
640×480	Visible	58	116	99	70	70
	Full RT	79	161	145	116	115

Table 5.2: Gather table creation times.

Frame Size	**RCPU**	**RGPU1**	**RGPU2**
1024×768	60	15.5	5.1
800×600	36	1.87	3.2
640×480	22	1.2	2

5.5.2 Reprojections of Reflections and Refractions

Current method cannot handle reprojections of reflected and refracted pixels. If the scene is mostly diffuse and open environment this may not be a big problem, since most of the rays can terminate or escape from the scene immediately. On the other hand it is possible to generate approximate reprojections for these secondary rays by employing some heuristics. Reprojection of reflected/refracted pixels can be done right after the reflection/refraction pass finishes. Along with other information, reflected/refracted ray directions can also be reprojected and compared to real reflection/refraction directions. If the angle between the real and reprojected directions is less than a user defined threshold, reprojected color value may be used; otherwise the pixel is marked as `NO_HIT`. It is also possible to alter the threshold value adaptively, such that rays closer to the eye are given tighter threshold. Similarly, as the ray depth increases the threshold can be relaxed.

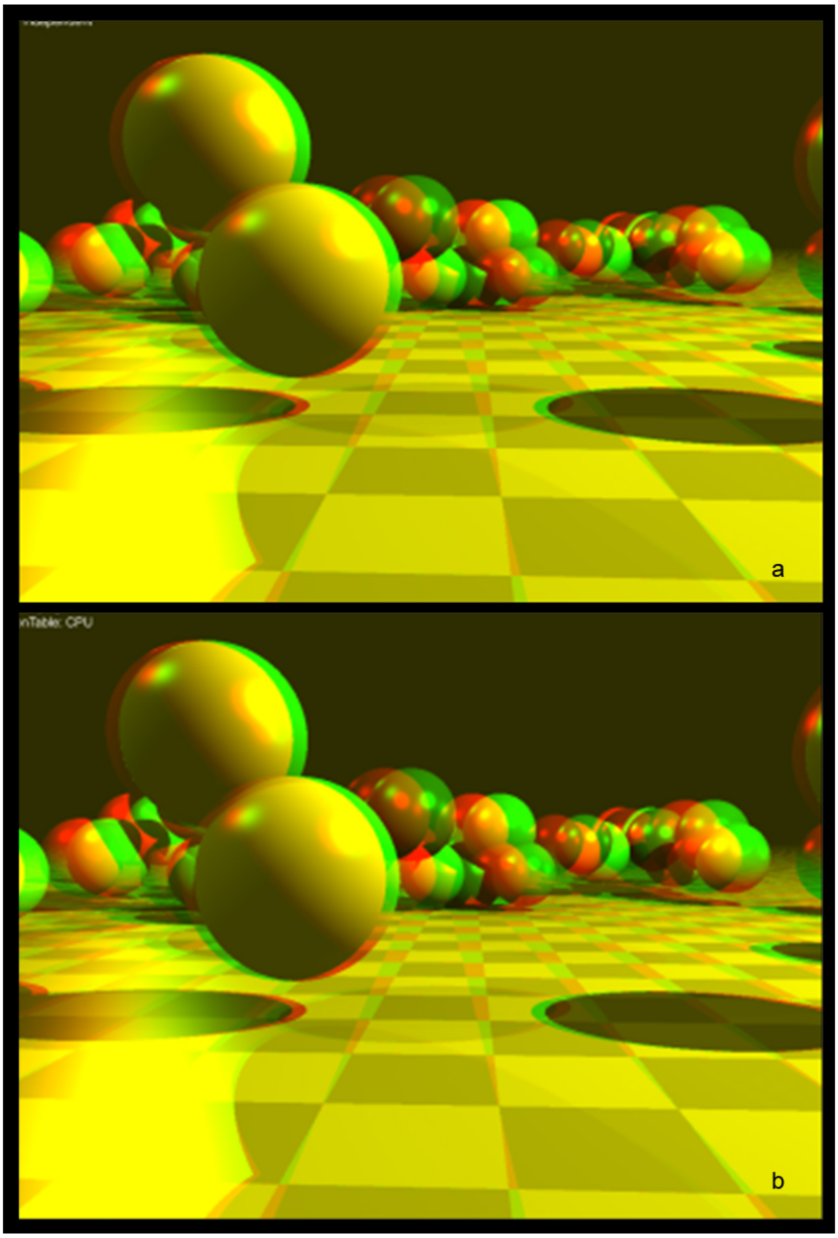


Figure 5.7: Anaglyph rendering of a dynamically animated scene (a) both eyes separately rendered, (b) right eye reprojected from the left eye's image.

CHAPTER 6

CONCLUSION

Real-time photorealistic rendering has broad application areas and an active research area with many problems remains to be solved. There are several approaches for the solution including processor specific optimizations, using rasterization based methods, employing better acceleration structures, building custom hardware and so on. Rasterization rendering with fixed function lighting and shading methods fall short for photorealism in most cases. Real-time photorealism requires fast algorithms for shadows, reflections, refractions, illumination and means to combine all of these effects efficiently. Particle based rendering methods such as ray tracing, path tracing or photon mapping provide an extensive solution to the global illumination problem and can generate realistic images. However, these methods require huge processing power. Considering the enormous arithmetical power of GPUs, we focused on efficiently utilizing the computational resources of GPU to accelerate ray tracing for real-time photorealistic rendering purposes.

It is crucial to explore efficient data structures and algorithms conforming to the parallel stream processing model to make best possible use of the graphics hardware. In this thesis, we used GPUs found on the commodity graphics cards to accelerate ray tracing. We devised and developed GPU specific solutions for fast ray tracing, volume rendering and stereoscopic image generation. The proposed EACD grids based acceleration of ray tracing and stereoscopic rendering can be used in virtual reality and other real-time applications.

Firstly, we studied regular grid based traversal techniques and introduced a GPU based traversal algorithm based on our extended anisotropic chessboard distance transformations. In order to compare the performance, efficient GPU implementations of some of the previously known traversal techniques have also been given. It is shown that the introduced traversal algorithm is several times faster than DDA, and considerably faster than other regular grid based empty space skipping methods. In addition, our algorithm suits well to the modern CPU architectures which support streaming parallel instructions. Therefore, presented methods can be ported to SIMD capable CPUs with some minor modifications. Our EACD based traversal algorithm requires minimum number of dynamic branching operations making it very efficient for the modern GPU architectures. As demonstrated by the test results, the ray traversal part is not the bottleneck in most cases when EACD is used. In general, especially for EACD and ACD, the main determining factor of the performance is time spent for the intersection tests. Time required for the intersection tests can be reduced by using finer grid subdivisions. Size of the graphics memory defines the limit on the maximum grid dimensions. A possible way to overcome this problem is to use hierarchical EACD grids. Hierarchical structure can eliminate the two major problematic points of the method including long preprocessing times and large memory consumption. We consider exploring hybrid or hierarchical EACD acceleration methods as a valuable future work.

Our EACD ray tracer uses multiple kernel passes for traversal and intersections. However, with the increased capabilities of the latest graphics hardware and development tools, it is possible to unify intersection and traversal kernels into a single kernel. This is expected to increase the performance because of the fact that; unnecessary I/O operations to send intermediate values between kernels, ray counting operations, and excessive API overheads due to multiple kernel passes will be eliminated. We left the implementation of this approach as a possible future work.

EACD acceleration structure is also suitable for direct volume visualization; since there are no triangle intersection tests, and ray traversal is one of most time consuming part of the rendering. Consequently in the second work, we used EACD based homogeneous region skipping to accelerate volume ray casting. In this technique, we exploited the coherency existing in most of the scientific volume data. Ray traversal through the empty regions and homogeneous regions is achieved efficiently with our method. Within a homogeneous region, ray integral is calculated in a single step using a factorization method. In order to compare the performance of our method, efficient GPU versions of some of the previously known ray casting techniques were also implemented. It is shown that the EACD based traversal algorithm is several times faster than a primitive volume ray caster, and considerably faster than other distance grids based homogeneous region leaping methods.

As the third work, we devised a GPU friendly solution to stereoscopic ray tracing. Real-time visual realism and real-time stereoscopic image generation are two important requirements of interactive virtual reality applications. In the method, the ray traced image of the left eye is reprojected to right eye and only suspicious areas are re-rendered for the right eye. The technique is based on a previous work of Adelson and Hodges [1]. However efficient mapping of the stereo reprojections to GPU programming model requires attention. We proposed pure GPU and hybrid CPU-GPU solutions to the reprojection problem. As shown experimentally, pure GPU methods can generate stereo images more efficiently than the hybrid one. In the context, we also discussed possible extensions for the reprojection technique to handle reflections and refractions, and proposed a GPU based method to solve missing object problem that may occur during the reprojections.

There are many possible research paths to further accelerate our GPU based ray tracing. For instance, coherence is one of the key elements for high performance rendering. We think that it is worthwhile to research on increasing the level of ray coherency in GPU ray tracers. Rendering a scene as small tiles instead of a whole buffer, or compacting/reordering rays for better coherency may help to increase the performance. Similarly, processing ray packets instead of single rays may utilize SIMD units better, while increasing compute to bandwidth ratio. Our work relies on the pipelined architecture on GPUs. On the other hand, some newly emerged programming interfaces and technologies such as CUDA and CTM have different approach. These new technologies depart from pipelines and favor unified processing model with huge amounts of threads. Employing these new technologies can make more efficient use of the GPU resources and thus worths great deal of attention.

GPUs are becoming more parallel, powerful and flexible processors. We strongly believe that with increasing flexibility, programmability and processing power, GPUs have great potential for achieving the goal of real-time photorealistic ray tracing.

REFERENCES

[1] Stephen J. Adelson and Larry F. Hodges. Visible surface ray-tracing of stereoscopic images. In *ACM-SE 30: Proceedings of the 30th annual Southeast regional conference*, pages 148–156, New York, NY, USA, 1992. ACM Press.

[2] Stephen J. Adelson and Larry F. Hodges. Generating Exact Ray-Traced Animation Frames by Reprojection. *IEEE Comput. Graph. Appl.*, 15(3):43–52, 1995.

[3] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering (Second Edition).* AK Peters, Ltd., 2002.

[4] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10, Amsterdam, North-Holland, 1987. Elsevier Science Publishers.

[5] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, Washington, D.C, 1968. Thompson Books.

[6] William Bilodeau, Michael Songy (Inventors), and Creative Tech Ltd (Applicant). US6384822: Method for rendering shadows using a shadow volume and a stencil buffer. *US Patent*, 1999.

[7] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547, 1976.

[8] Gunilla Borgefors. Distance transformations in digital images. *Comput. Vision Graph. Image Process.*, 34(3):344–371, 1986.

[9] Paul Bourke. Calculating Stereo Pairs. *http://local.wasp.uwa.edu.au/ pbourke/projection/stereorender/*, Last visited January 2008.

[10] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM.

[11] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[12] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006*, pages 203–209, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[13] Daniel Cohen and Zvi Sheffer. Proximity clouds - an acceleration technique for 3D grid traversal. *The Visual Computer*, 11(1):27–38, 1994.

[14] Microsoft Corp. DirectX Resource Center. *http://msdn2.microsoft.com/en-us/directx/default.aspx*, Last visited January 2008.

[15] Franklin C. Crow. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248, New York, NY, USA, 1977. ACM Press.

[16] Timothy J. Cullip and Ulrich Neumann. Accelerating Volume Reconstruction With 3D Texture Hardware. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1994.

[17] John Danskin and Pat Hanrahan. Fast algorithms for volume ray tracing. In *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, pages 91–98, New York, NY, USA, 1992. ACM Press.

[18] Advanced Micro Devices. ATI CTM Guide Technical Reference Manual. *http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf*, Last visited January 2008.

[19] Olivier Devillers. The Macro-Regions: An Efficient Space Subdivision Structure for Ray Tracing. In *Eurographics '89*, pages 27–38. Elsevier / North-Holland, 1989.

[20] Mike Dickheiser. *Game Programming Gems 6.* Delmar Cengage Learning, 2006.

[21] Paul J. Diefenbach. *Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering.* PhD thesis, The University of Pennsylvania., 1996.

[22] Philip Dutre, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination.* AK Peters, Ltd., 2003.

[23] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, New York, NY, USA, 2001. ACM.

[24] Alphan Es, Hacer Yalım Keleş, and Veysi İşler. Accelerated Volume Rendering with Homogeneous Region Encoding using Extended Anisotropic Chessboard Distance on GPU. In *EGPGV '06 Proceedings*, pages 67–73, 2006.

[25] Alphan Es and Veysi İşler. Acceleration of Regular Grid Traversals Using Extended Chessboard Distance Transformation on GPU. In *CAD-CG '05: Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05)*, pages 434–441, Washington, DC, USA, 2005. IEEE Computer Society.

[26] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[27] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C (2nd Edition).* Addison-Wesley Professional, 1995.

[28] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[29] A. Fujimoto, Takayuki Tanaka, and K. Iwata. ARTS: accelerated ray-tracing system. In *Tutorial: computer graphics; image synthesis*, pages 148–159, New York, NY, USA, 1988. Computer Science Press, Inc.

[30] Ping-Fu Fung and Pheng-Ann Heng. Efficient Volume Rendering by IsoRegion Leaping Acceleration. In *Proc. Of The Sixth International Conference in Central Europe on Computer Graphics and Visualization'98*, 1998.

[31] Allen Van Gelder and Kwansik Kim. Direct volume rendering with shading via three-dimensional textures. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 23–ff, Piscataway, NJ, USA, 1996. IEEE Press.

[32] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Real-time Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, 2007.

[33] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 213–222, New York, NY, USA, 1984. ACM Press.

[34] GPGPU. GPGPU General-Purpose Computation Using Graphics Hardware. *http://www.gpgpu.org*, Last visited January 2008.

[35] Silicon Graphics. OpenGL: Home Page. *http://www.sgi.com/products/software/opengl/*, Last visited January 2008.

[36] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361, New York, NY, USA, 2002. ACM.

[37] Eric Haines. The standard procedural databases (SPD). *http://tog.acm.org/resources/SPD/*, Last visited January 2008.

[38] Daniel Hall. The AR350: Today's ray trace rendering processor. In *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware - Hot 3D Presentations*, 2001.

[39] Vlastimil Havran. *Heuristic ray shooting algorithms.* PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000.

[40] Vlastimil Havran, J. Bittner, and Hans-Peter Seidel. Exploiting temporal coherence in ray casted walkthroughs. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 149–155, New York, NY, USA, 2003. ACM Press.

[41] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph.*, 24(4):145–154, 1990.

[42] Tim Heidmann. Real Shadows, Real Time. *Iris Universe*, 18:23–31, 1991.

[43] Karl Hillesland and Anselmo Lastra. GPU floating-point paranoia. In *Proceedings of GP2*, 2004.

[44] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[45] Jim Hurley. Ray tracing goes mainstream. *Intel Technology Journal*, 9(2):99–108, 2005.

[46] RapidMind Inc. Sh: A high-level metaprogramming language for modern GPUs. *http://www.libsh.org/*, Last visited January 2008.

[47] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping.* A. K. Peters, Ltd., Natick, MA, USA, 2001.

[48] Sig Badt Jr. Two Algorithms for Taking Advantage of Temporal Coherence In Ray Tracing. *The Visual Computer*, 4(3):123–132, 1998.

[49] James T. Kajiya. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, New York, NY, USA, 1986. ACM Press.

[50] Filip Karlsson and Carl Johan Ljungstedt. Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Chalmers Univ. of Technology, 2004.

[51] Hacer Yal?m Kele?, Alphan Es, and Veysi ??ler. Acceleration of direct volume rendering with programmable graphics hardware. *The Visual Computer*, 23(1):15–24, 2007.

[52] J. Kruger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 287–292, Washington, DC, USA, 2003. IEEE Computer Society.

[53] Stanford Computer Graphics Laboratory. BrookGPU. *http://graphics.stanford.edu/projects/brookgpu/*, Last visited January 2008.

[54] Stanford Computer Graphics Laboratory. GPU Bench. *http://graphics.stanford.edu/projects/gpubench/*, Last visited January 2008.

[55] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA, 1994. ACM Press.

[56] David Laur and Pat Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *SIGGRAPH Comput. Graph.*, 25(4):285–288, 1991.

[57] Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, 1988.

[58] Wei Li and Arie Kaufman. Texture Partitioning and Packing for Accelerating Texture-Based Volume Rendering. In *Graphics Interface*, pages 81–88. A K Peters, 2003.

[59] Wei Li, Klaus Mueller, and Arie Kaufman. Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 42, Washington, DC, USA, 2003. IEEE Computer Society.

[60] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM Press.

[61] Tom McReynolds and David Blythe. Advanced Graphics Programming Techniques Using OpenGL, (SIGGRAPH '99 Course Notes). *http://www.opengl.org/resources/code/samples/sig99/advanced99/notes/notes.html*, Last visited January 2008.

[62] Michael Meißner, Ulrich Hoffmann, and Wolfgang Straßer. Enabling classification and shading for 3D texture mapping based volume rendering using OpenGL and extensions. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 207–214, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[63] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, 1997.

[64] Kasper H. Nielsen. Real-time hardware-based photorealistic rendering. Master's thesis, Informatics and Mathematical Modelling, The Technical University of Denmark, 2000.

[65] nVidia. Floating point specials,. *http://download.nvidia.com/developer/Papers/2005/FP_Specials/FP_Specials.pdf*, Last visited January 2008.

[66] nVidia. GeForce 8 Series. *http://www.nvidia.com/page/geforce8.html*, Last visited January 2008.

[67] nVidia. NVIDIA CUDA. *http://developer.nvidia.com/object/cuda.html*, Last visited January 2008.

[68] nVidia. NVPerfKit. *http://developer.nvidia.com/object/nvperfkit_home.html*, Last visited January 2008.

[69] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1998. ACM Press.

[70] John D. Owens. *Streaming architectures and technology trends.* Addison-Wesley, 2005.

[71] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems).* Addison-Wesley Professional, 2005.

[72] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum (Proceedings of Eurographics)*, 26(3):415–424, 2007.

[73] Timothy J. Purcell. *Ray tracing on a stream processor.* PhD thesis, Stanford University, 2004.

[74] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA, 2002. ACM Press.

[75] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 258, New York, NY, USA, 2005. ACM Press.

[76] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 283–291, New York, NY, USA, 1987. ACM Press.

[77] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM Press.

[78] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118, New York, NY, USA, 2000. ACM.

[79] David Roger, Ulf Assarsson, and Nicolas Holzschuch. Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. In *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, pages 99–110, 2007.

[80] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[81] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR: a hardware architecture for ray tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[82] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime ray tracing of dynamic scenes on an FPGA chip. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 95–106, New York, NY, USA, 2004. ACM Press.

[83] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *SIGGRAPH Comput. Graph.*, 26(2):249–252, 1992.

[84] Milos Sramek and Arie Kaufman. Fast ray-tracing of rectilinear volume data using distance transforms. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):236–252, 2000.

[85] Marc Stamminger and George Drettakis. Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 557–562, New York, NY, USA, 2002. ACM Press.

[86] K. R. Subramanian and Donald S. Fussell. Applying space subdivision techniques to volume rendering. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 150–159, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[87] Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for GPU. Master's thesis, University of Aarhus, Denmark, 2005.

[88] Jayaram K. Udupa and Dewey Odhner. Shell rendering. *IEEE Comput. Graph. Appl.*, 13(6):58–67, 1993.

[89] Stanford University. The Stanford 3D Scanning Repository. *http://graphics.stanford.edu/data/3Dscanrep/*, Last visited January 2008.

[90] Ingo Wald, Carsten Benthin, and Philipp Slusallek. A simple and practical method for interactive ray tracing of dynamic scenes. Technical report, Saarland University, 2002.

[91] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1):6, 2007.

[92] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006.

[93] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Comput. Graph. Forum (EG 2001 Proceedings)*, 20(3):153–164, 2001.

[94] Daniel Weiskopf, Tobias Schafhitzel, and Thomas Ertl. GPU-based nonlinear ray tracing. *Comput. Graph. Forum*, 23:625–634, 2004.

[95] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference*

on Computer graphics and interactive techniques, pages 169–177, New York, NY, USA, 1998. ACM.

[96] Lee A. Westover. *Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm.* PhD thesis, Univ. of North Carolina at Chapel Hill, Chapel Hill, N.C., 1991.

[97] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[98] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, New York, NY, USA, 1978. ACM Press.

[99] Andrew Woo. The shadow depth map revisited. In *Graphics Gems III*, pages 338–342. Academic Press Professional, Inc., San Diego, CA, USA, 1992.

[100] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.

[101] K.J. Zuiderveld, A.H.J. Koning, and M.A. Viergever. Acceleration of ray-casting using 3D distance transforms. In *Visualization in Biomedical Computing II, Proc. SPIE 1808*, pages 324–335, 1992.

APPENDIX A

SHADER SOURCE CODES

A.1 Grid based ray traversal source codes

```
#define O_EPS                     0.0001f
#define T_INT_EPS                 0.0001f
#define STATIC_TRAVERSE_STEPS     0
#define INTERSECT_RAY             0
#define OUT_RAY                   -1
#define EMPTY_VOX                 -2
```

A.1.1 Cg source code for branching DDA ray traversal

```
// Continue traversal until a non-empty voxel is found
void stepTraverse(
            float2 rayIdx    : WPOS,          // ray index
    out     float4 oCellIdxT : COLOR,         // results
    #ifdef DEPTH_OUT
    out     float  oDepth    : DEPTH,         // depth buffer
    uniform float  outCode,                   // depth value for ouf of scene rays
    uniform float  intersectCode,             // depth value for rays to be tested
    #endif
    uniform sampler3D          texGrid,       // accel. grid
    uniform samplerRECT        texCellIdx,    // last ray positions (cell idx)
    uniform samplerRECT        texRayO,       // ray origins
    uniform samplerRECT        texRayD,       // ray directions
    uniform float3             cellSize,      // size of a cell
    uniform float3             invNumCells    // inv. Number of cells: 1/(W,H,D)
)
{
    // setup
    float3 d = texRECT(texRayD, rayIdx).xyz;
    float3 sign_d = sign(d);
    float3 cellStep = sign_d;
    float3 tStep = sign_d*(cellSize/d);

    float4 tttCell = texRECT(texCellIdx, rayIdx);
    float4 cellStat = unpack_4ubyte( tttCell.w );
    float3 ttt = tttCell.xyz;
    float3 cell = round(cellStat.xyz*255);

    float4 gridSample;
    float3 incr;
    // loop body
    do
    {
        float tmin = min( ttt.x, min(ttt.y,ttt.z));
        incr.x = (ttt.x == tmin);
        incr.y = (ttt.y == tmin);
        incr.z = (ttt.z == tmin);

        ttt += tStep*incr;
        cell += cellStep*incr;

        gridSample = tex3D(texGrid, cell*invNumCells);
    } while( gridSample.w==0 );     // loop until a non-empty voxel found
                                    // or the ray is out of scene
    // write results
    float fPacked = pack_4ubyte( float4(cell/255.0) );
    #ifdef DEPTH_OUT
        oDepth = gridSample.w == 1 ? outCode : intersectCode;
    #else
        if (gridSample.w == 1) ttt.x = -1;  // depth will be modified
                                            // accordingly by the next kernel
    #endif
    oCellIdxT = float4( ttt, fPacked  );
}
```

A.1.2 Cg source code for multi-pass DDA ray traversal

```
// Perform one or a predefined number of traversal steps
void
stepTraverseMP(
            float2 rayIdx       : WPOS,     // ray index
    out     float4 oCellIdxT    : COLOR,    // results

    uniform sampler3D       texGrid,        // accel. grid
    uniform samplerRECT     texCellIdx,     // last ray positions (cell idx)
    uniform samplerRECT     texRayO,        // ray origins
    uniform samplerRECT     texRayD,        // ray directions

    uniform float3          cellSize,       // size of a cell
    uniform float3          invNumCells     // inv. Number of cells: 1/(W,H,D)
)
{
    // setup
    float3 d = texRECT(texRayD, rayIdx).xyz;
    float3 sign_d = sign(d);
    float3 cellStep = sign_d;
    float3 tStep = sign_d*(cellSize/d);

    float4 tttCell = texRECT(texCellIdx, rayIdx);
    float4 cellStat = unpack_4ubyte( tttCell.w );
    float3 ttt = tttCell.xyz;
    float3 cell = round(cellStat.xyz*255);

    float4 gridSample;
    float3 incr;

    // 1 traversal step
    float tmin = min( ttt.x, min(ttt.y,ttt.z));
    incr.x = (ttt.x == tmin);
    incr.y = (ttt.y == tmin);
    incr.z = (ttt.z == tmin);
    ttt += tStep*incr;
    cell += cellStep*incr;
    gridSample = tex3D(texGrid, cell*invNumCells);

    //additional traversal steps if desired (static loop unrolled by compiler)
    for (int loops=0;loops<STATIC_TRAVERSE_STEPS-1;++loops)
    if (gridSample.w == 0)
    {
        float tmin = min( ttt.x, min(ttt.y,ttt.z));
        incr.x = (ttt.x == tmin);
        incr.y = (ttt.y == tmin);
        incr.z = (ttt.z == tmin);

        ttt += tStep*incr;
        cell += cellStep*incr;

        gridSample = tex3D(texGrid, cell*invNumCells);
    }

    float fCode = INTERSECT_RAY;                     // output value for intersection
    if (gridSample.w == 1) fCode = OUT_RAY;          // ,or for out of scene
    else if (gridSample.w == 0) fCode = EMPTY_VOX;   // ,or for empty voxel
    float fPacked = pack_4ubyte( float4(cell/255,fCode/255.0f) );

    if (gridSample.w == 1) ttt.x = -1;
    oCellIdxT = float4( ttt, fPacked  );
}
```

A.1.3 Cg source code for branching PC ray traversal

```
// Continue traversal until a non-empty voxel is found
void
stepTraverse(
            float2 rayIdx       : WPOS,     // ray index
    out     float4 oCellIdxT    : COLOR,    // results
    #ifdef DEPTH_OUT
    out float  oDepth       : DEPTH,        // depth buffer
    uniform float           outCode,        // depth value for ouf of scene rays
    uniform float           intersectCode,  // depth value for rays to be tested
    #endif
    uniform sampler3D       texGrid,        // accel. grid
    uniform samplerRECT     texCellIdx,     // last ray positions (cell idx)
    uniform samplerRECT     texRayO,        // ray origins
    uniform samplerRECT     texRayD,        // ray directions

    uniform float3          invNumCells     // inv. Number of cells: 1/(W,H,D)
)
{
    // setup phase
    float3 o = texRECT(texRayO, rayIdx).xyz;
    float3 d = texRECT(texRayD, rayIdx).xyz;
    float3 invD = 1/d;
    float3 sign_d = sign(d);
    float3 sign_d_epsilon = sign_d*O_EPS;

    // read previous traversal status
    float4 tttCell = texRECT(texCellIdx, rayIdx);
    float4 cellStat = unpack_4ubyte( tttCell.w );
    float3 cell = round(cellStat.xyz*255);

    // dda related
    float3 cellStep = sign_d;
    float3 tStep = sign_d*invD;
    float3 pos = min(tttCell.x, min(tttCell.y, tttCell.z))*d+o;

    // proximity clouds related
    float3 abs_d = abs(d);
    float3 tttBegin = ( saturate(sign_d+sign_d_epsilon+float3(1,1,1)) -o  ) * invD;
    float D = abs_d.x+abs_d.y+abs_d.z;
    float3 C = d/D;

    // temporaries
    float pd;
    float3 ttt = tttCell.xyz;

    float4 gridSample = tex3D(texGrid, cell*invNumCells);

    // loop body
    do
    {
        if ( gridSample.x == 0 )// close to a non-empty voxel -> dda
        {
            float tmin = min(ttt.x, min(ttt.y,ttt.z));
            float3 incr;
            incr.x = (ttt.x == tmin);
            incr.y = (ttt.y == tmin);
            incr.z = (ttt.z == tmin);
            ttt += tStep*incr;
            cell += cellStep*incr;
            pos = o+tmin*d;
        }
        else                    // otherwise -> proximity clouds
        {
            pd = round( gridSample.x * 255 );
            pos += C*pd;
            cell = floor(pos);
            ttt = tttBegin + cell*invD;
        }
        gridSample = tex3D(texGrid, cell*invNumCells);

    } while( gridSample.w==0 ); // loop until a non-empty voxel found
                                // or the ray is out of scene

    // write results
    float fPacked = pack_4ubyte( float4((cell)/255.0) );
    #ifdef DEPTH_OUT
        oDepth = gridSample.w == 1 ? outCode : intersectCode;
    #else
        ttt.x = (gridSample.w == 1) ? -1 : ttt.x; // depth will be modified
                                                  // accordingly by the next kernel
#endif
    oCellIdxT = float4( ttt, fPacked  );
}
```

A.1.4 Cg source code for multi-pass PC ray traversal

```
// Perform one or a predefined number of traversal steps
void
stepTraverseMP(
            float2 rayIdx       : WPOS,     // ray index
    out     float4 oCellIdxT    : COLOR,    // results

    uniform sampler3D           texGrid,    // accel. grid
    uniform samplerRECT         texCellIdx, // last ray positions (cell idx)
    uniform samplerRECT         texRayO,    // ray origins
    uniform samplerRECT         texRayD,    // ray directions

    uniform float3              cellSize,   // size of a cell
    uniform float3              invNumCells // inv. Number of cells: 1/(W,H,D)
)
{
    // setup phase
    float3 o = texRECT(texRayO, rayIdx).xyz;
    float3 d = texRECT(texRayD, rayIdx).xyz;
    float3 invD = 1/d;
    float3 sign_d = sign(d);
    float3 sign_d_epsilon = sign_d*O_EPS;

    // read previous traversal status
    float4 tttCell = texRECT(texCellIdx, rayIdx);
    float4 cellStat = unpack_4ubyte( tttCell.w );
    float3 cell = round(cellStat.xyz*255);

    // dda related
    float3 cellStep = sign_d;
    float3 tStep = sign_d*invD;
    float3 pos = min(tttCell.x, min(tttCell.y, tttCell.z))*d+o;

    // proximity clouds related
    float3 abs_d = abs(d);
    float3 tttBegin = ( saturate(sign_d+sign_d_epsilon+float3(1,1,1)) -o  ) * invD;
    float D = abs_d.x+abs_d.y+abs_d.z;
    float3 C = d/D;

    // temporaries
    float pd;
    float3 ttt = tttCell.xyz;

    float4 gridSample = tex3D(texGrid, cell*invNumCells);

    // 1 traversal step
    if ( gridSample.x == 0 )    // close to a non-empty voxel -> dda
    {
        float tmin = min(ttt.x, min(ttt.y,ttt.z));
        float3 incr;
        incr.x = (ttt.x == tmin);
        incr.y = (ttt.y == tmin);
        incr.z = (ttt.z == tmin);
        ttt += tStep*incr;
        cell += cellStep*incr;
        pos = o+tmin*d;
    }
    else                        // otherwise -> proximity clouds
    {
        pd = round( gridSample.x * 255 );
        pos += C*pd;
        cell = floor(pos);
        ttt = tttBegin + cell*invD;
    }
    gridSample = tex3D(texGrid, cell*invNumCells);

    // additional traversal steps if desired (static loop unrolled by compiler)
    for (int loops=0;loops<STATIC_TRAVERSE_STEPS-1;++loops)
    if( gridSample.w == 0 )
    {
        if ( gridSample.x == 0 )// close to a non-empty voxel -> dda
        {
            float tmin = min(ttt.x, min(ttt.y,ttt.z));
            float3 incr;
            incr.x = (ttt.x == tmin);
            incr.y = (ttt.y == tmin);
            incr.z = (ttt.z == tmin);
            ttt += tStep*incr;
            cell += cellStep*incr;
            pos = o+tmin*d;
        }
        else                    // otherwise -> proximity clouds
        {
            pd = round( gridSample.x * 255 );
            pos += C*pd;
            cell = floor(pos);
            ttt = tttBegin + cell*invD;
        }
        gridSample = tex3D(texGrid, cell*invNumCells);
    }

    float fCode = INTERSECT_RAY;                        // output value for intersection
    if (gridSample.w == 1) fCode = OUT_RAY;             // ,or for out of scene
    else if (gridSample.w == 0) fCode = EMPTY_VOX;      // ,or for empty voxel
    float fPacked = pack_4ubyte( float4(cell/255,fCode/255.0f) );
    oCellIdxT = float4( ttt, fPacked  );
}
```

A.1.5 Cg source code for branching ACD/EACD ray traversal

```
// Continue traversal until a non-empty voxel is found
void
stepTraverse(
            float2 rayIdx       : WPOS,     // ray index
    out     float4 oCellIdxT    : COLOR,    // results
    #ifdef DEPTH_OUT
    out float  oDepth       : DEPTH,        // depth buffer
    uniform float           outCode,        // depth value for ouf of scene rays
    uniform float           intersectCode,  // depth value for rays to be tested
    #endif
    uniform sampler3D       texGrid,        // accel. grid
    uniform samplerRECT     texCellIdx,     // last ray positions (cell idx)
    uniform samplerRECT     texRayO,        // ray origins
    uniform samplerRECT     texRayD,        // ray directions

    uniform float3          invNumCells     // inv. Number of cells: 1/(W,H,D)
    uniform float3          halfInvNumCells // 0.5 * invNumCells
    uniform float3          rangeScale,     // scale required to expand (0,,1)
                                            // distance to integer
{
    // setup phase ------------------------------
    float3 o = texRECT(texRayO, rayIdx).xyz;
    float3 d = texRECT(texRayD, rayIdx);

    // read previous traversal status
    float4 cellw = texRECT(texCellIdx, rayIdx);

    // compute constants
    float3  invD        = (1/d);
    float3  sgn         = sign(fixed3(d));
    float3  sgnScaled   = rangeScale*sgn;
    float3  offs        = saturate(-sgn)*halfInvNumCells;
    float3  oEps        = o + sgn*O_EPS;
    float3  tttBegin    = ( saturate(sgn) -o ) *invD;
    // advance to the next voxel
    float3 cell = floor(oEps+(cellw.w  + T_INT_EPS)*d);

    // read voxel
    float4 gridSample = tex3D(texGrid, cell*invNumCells+offs);

    // loop body
    float4 ttt;
    while( gridSample.w==0 )// loop until non-empty voxel or out of scene
                            // is reached
    {
        // find border cell (apex voxel)
        #ifdef  EACD_MODE   // -> EACD
            float3 range = round(sgnScaled*gridSample.xyz);
        #else               // -> ACD
            float3 range = round(sgnScaled*gridSample.x);
        #endif
        float3 borderCell = (cell+range);

        // compute parameteric ray distance to the apex voxel borders
        ttt.xyz = tttBegin + borderCell*invD;
        ttt.w = min(ttt.x, min(ttt.y,ttt.z));

        // compute indices fo the next voxel
        float3 oNew = oEps+ttt.w*d;
        cell = floor(oNew);

        // read voxel
        gridSample = tex3D(texGrid, cell*invNumCells+offs);
    }

    // parametric ray distance to the nearest voxel border (for intersection tests)
    ttt.xyz = tttBegin + cell*invD;
    ttt.w = min(ttt.x, min(ttt.y,ttt.z));

    // write results
#ifdef DEPTH_OUT
    oCellIdxT = float4(cell, ttt.w);
    oDepth = gridSample.w < 1 ? outCode : intersectCode;
#else
    oCellIdxT = float4(cell, (gridSample.w < 1) ? -1 : ttt.w);// depth will be set
                                                              // accordingly
                                                              // by the next kernel
#endif
}
```

A.1.6 Cg source code for multi-pass ACD/EACD ray traversal

```
// Perform one or a predefined number of traversal steps
void
stepTraverseMP(
            float2 rayIdx       : WPOS,         // ray index
    out     float4 oCellIdxT    : COLOR,        // results

    uniform sampler3D           texGrid,        // accel. grid
    uniform samplerRECT         texCellIdx,     // last ray positions (cell idx)
    uniform samplerRECT         texRayO,        // ray origins
    uniform samplerRECT         texRayD,        // ray directions

    uniform float3              invNumCells     // inv. Number of cells: 1/(W,H,D)
    uniform float3              halfInvNumCells // 0.5 * invNumCells
    uniform float3              rangeScale,     // scale required to expand (0,,1)
)
{
    // setup phase
    float3 o = texRECT(texRayO, rayIdx).xyz;
    float3 d = texRECT(texRayD, rayIdx);

    // read previous traversal status
    float4 cellw = texRECT(texCellIdx, rayIdx);

    // compute constants
    float3 invD = (1/d);
    fixed3 sgn = sign(fixed3(d));
    half3  sgnScaled = rangeScale*sgn;
    half3  offs = saturate(-sgn)*halfInvNumCells;
    float3 oEps = o + sgn*O_EPS;
    float3 tttBegin = ( saturate(sgn) -o ) *invD;

    // advance to the next voxel
    half3 cell;
    if (cellw.w>0)
        cell = floor(oEps+(cellw.w  + T_INT_EPS)*d);
    else
        cell = cellw.xyz;

    // read voxel
    fixed4 gridSample = tex3D(texGrid, cell*invNumCells+offs);
    float4 ttt;

    // One or more traversal steps if desired (static loop unrolled by compiler)
    for (int loops=0;loops<STATIC_TRAVERSE_STEPS;++loops)
    if( gridSample.w==0 )
    {
        // find border cell (apex voxel)
        #ifdef  EACD_MODE// -> EACD
            float3 range = round(sgnScaled*gridSample.xyz);
        #else           // -> ACD
            float3 range = round(sgnScaled*gridSample.x);
        #endif
        half3 borderCell = (cell+range);

        // compute parametric ray distance to the apex voxel borders
        ttt.xyz = tttBegin + borderCell*invD;
        ttt.w = min(ttt.x, min(ttt.y,ttt.z));

        // advance to the next voxel
        float3 cell = floor(oEps+(cellw.w  + T_INT_EPS)*d);

        // read voxel
        float4 gridSample = tex3D(texGrid, cell*invNumCells+offs);
    }

    // parametric ray distance to the nearest voxel border (for intersection tests)
    ttt.xyz = tttBegin + cell*invD;
    ttt.w = min(ttt.x, min(ttt.y,ttt.z));

    // write results
    oCellIdxT = float4(cell,ttt.w );

    if ( gridSample.w == 0 ) oCellIdxT.w = EMPTY_VOX;         // mark as empty voxel,
        else if ( gridSample.w < 0.9 ) oCellIdxT.w = OUT_RAY; // or as the ray is out
}
```

A.1.7 Cg source code for intersector kernel (for ACD/EACD traversal)

```
// Triangle list - ray intersection tests
void
intersectTriList(
            float2 rayIdx       : WPOS,       // ray index
    out     float4 oCellIdxT    : COLOR,      // results
    #ifdef DEPTH_OUT
    out float  oDepth       : DEPTH,        // depth buffer
    uniform float           traverseCode,   // depth value for ray miss
    uniform float           intersectedCode,// depth value for ray hit
    #endif
    uniform sampler3D       texGrid,        // accel. grid
    uniform samplerRECT     texCellIdx,     // last ray positions (cell idx)
    uniform samplerRECT     texRayO,        // ray origins
    uniform samplerRECT     texRayD,        // ray directions

    uniform samplerRECT     texTriList,     // triangle list
    uniform samplerRECT     texTriV0,       // 1st triangle vertices
    uniform samplerRECT     texTriV1,       // 2nd triangle vertices
    uniform samplerRECT     texTriV2,       // 3rd triangle vertices

    uniform float3          invNumCells     // inv. Number of cells: 1/(W,H,D)
)
{
    // setup phase
    float3 rayD = texRECT( texRayD, rayTexIdx ).xyz;
    float3 rayO = texRECT( texRayO, rayTexIdx ).xyz;

    // read ray traversal status (voxel indices, max allowed parametric distance)
    float4 traverseSample = texRECT(texCellIdx,rayTexIdx);

    // compute voxel index to triangle list texture (scene database)
    float3 voxTexCoord = traverseSample.xyz;
    float2 triListIdx = tex3D( texGrid, voxTexCoord*invNumCells).xw;
    float2 triListIdxRECT = triListIdx * 65535 + float2(0.5f, 0.5f);

    // init other variables
    float4 baryTMin = traverseSample.wwww + float4(0,0,0.00001f,0);
    float4 baryT;
    float2 nearestTriIdx = float2(-1,-1);
    float3 v1,v2,v3;
    float2 triangleIdx;
    float3 EPSILON = float3(0.00001f, -0.00001f, 1.00001f );
    float3 qvec, pvec, tvec;
    float  det;
    float3 edge1, edge2;
    half2 triListIdxRECT_H = triListIdxRECT;
    float fCnt = texRECT( texTriList, triListIdxRECT_H ).x;
    fCnt = round(fCnt*65535.0f);
    half fCntH = fCnt + triListIdxRECT.x;

    // ray - triangle list intersection test loop
    do
    {
        // read next triangle
        triListIdxRECT_H.x++;
        triangleIdx = texRECT( texTriList, triListIdxRECT_H ).xw;

        float2 triangleIdxRECT = triangleIdx * 65536.0f;

        // read vertex data
        v1 = texRECT( texTriV0, triangleIdxRECT.xy ).xyz;
        v2 = texRECT( texTriV1, triangleIdxRECT.xy ).xyz;
        v3 = texRECT( texTriV2, triangleIdxRECT.xy ).xyz;

        // Möller–Trumbore intersection test
        edge1  = v2-v1;
        edge2  = v3-v1;
        pvec = cross(rayD,edge2);
        det = dot(edge1,pvec);
        tvec = ray0 -  v1;
        qvec = cross(tvec,edge1);
        baryT.xyz = float3( dot(tvec, pvec), dot(rayD, qvec), dot(edge2,qvec) )/det;

        // test intersection result
        if (    baryT.z≤baryTMin.z &&
                baryT.z>EPSILON.x &&
                baryT.x>=0 && baryT.y>=0 &&
                baryT.x+baryT.y≤EPSILON.z )
        {
            baryTMin = baryT;               // a valid intersection is found ->
            nearestTriIdx = triangleIdx;    // update intersection result
        }
    } while (triListIdxRECT_H.x<fCntH);     // loop until all triangles tested

    // write intersection results
    oBaryI = float4( baryTMin.xy, nearestTriIdx.xy*65535 );

    #ifdef DEPTH_OUT
        oDepth = nearestTriIdx.y < 0 ? traverseCode : intersectedCode;
    #endif
}
```

A.2 Direct volume rendering source code

```
struct BBox
{
    float3  min;            // in World Coordinate System (WCS)
    float3  max;            // in WCS
    float3  size;           // in WCS
    float3  invSize;        // in WCS

    float3  cellSize;       // in WCS
    float3  invCellSize ;   // in WCS
    float3  numCells;       // in WCS
    float3  invNumCells;    // in WCS

    float3  worldToTexCoord( float3 posWorld )
    {
        return (posWorld-min)*invSize;
    }
};

// light information
struct Light
{
    float3  position;       // in WCS
    float3  positionGrid;   // in grid coords (0,.wxHxD)
    float3  ambient;        // light ambient color
    float3  diffuse;        // light diffuse color
    float3  specular;       // light specular color
};
```

```
// volume material properties
struct Material
{
    float3  ambient;        // surface ambient color
    float3  diffuse;        // surface diffuse color
    float3  specular;       // surface specular color
    float   glossiness;     // surface glossiness
};

// phong illumination function
float3  phong(
    Material mat,
    Light    light,
    float3   N,
    float3   L,
    float3   E
    )
{
    if (dot(N,E)<0) N = -N;
    float fD = dot(N,L);
    float3 A = mat.ambient * light.ambient;
    float3 C = A;
    if (fD>0)
    {
        float3 D = mat.diffuse*light.diffuse*dot(N,L);
        float3 R = 2*dot(L,N)*N-L;
        float3 S = mat.specular *
                   light,specular*pow(saturate(dot(E,R)),mat.glossiness );
        C += D+S;
    }
    return C;
}

// output registers
struct FragOut
{
    float4  color:  COLOR0;
    #ifdef  STATS
        float4  stats:  COLOR1;
    #endif
};
```

A.2.1 Volume rendering with ACD/EACD

```
FragOut
render(
    uniform     samplerRECT texRay0,            // ray origins
    uniform     samplerRECT texRayD,            // ray directions
    uniform     sampler3D   texDist,            // EACD grid
    uniform     sampler3D   texVol,             // volume data
    varying     float2      texRayID: TEXCOORD0,// ray index

    uniform     float       stepDelta,          // min ray step distance
    uniform     float       invStepDelta,       // 1/stepDelta
    uniform     float3      rangeScale,         // scale to expand 0..1 to
                                                // integer
    uniform     BBox        box,                // volume bounding box coords
    uniform     float3      eyePosGrid,         // eye position
    uniform     Material    mat,                // volume material
    uniform     Light       light,              // light information
    uniform     float       maxLoopCnt
```

```
)
{
    FragOut returnVal;      // define output registers

    // read ray origin
    float4 Ow = texRECT( texRayO, texRayID ).xyzw;

    // init final color value
    float4 C = float4(0,0,0,0);

    // Continue if the ray is inside the BBox
    if (Ow.w != 0)
    {
        // Find ray voxel & read ray direction
        float3 O  = (Ow.xyz-box.min)*box.invCellSize;
        float4 Dw  = texRECT( texRayD, texRayID ).xyzw;
        float3 D  = Dw.xyz*box.invCellSize;
        float  fMaxDist = Dw.w;

        // compute constants
        float3  invD        = (1/D);
        half3   sgn         = sign(half3(D));
        half3   sgnScaled   = rangeScale*sgn;
        float3  offs        = saturate(-sgn)*box.invNumCells*0.5f;
        float3  oEps        = O + sgn*0.00001f;
        float3  tttBegin    = ( saturate(sgn) - O ) *invD;
        half3   cell        = floor(oEps);
        float4  ttt;
        float   tNow        = 0;

        // read the distance from EACD grid
        half4 gridSample = tex3D(texDist, cell*box.invNumCells+offs);

        // loop until opacity of the final color is >= 1
        while( C,a < 1 )
        {
            half4 C2 = tex3D(texVol, (O)*box.invNumCells ); // real volume data

            float numSteps = 1;
            {
                #ifdef  EACD_MODE   // -> EACD
                    half3   range = round(sgnScaled*gridSample.xyz);
                #else               // -> ACD
                    half3   range = round(sgnScaled*gridSample,a);
                #endif

                // find border cell (apex voxel)
                half3 borderCell = (cell+range);

                // compute parameteric ray distance to the nearest apex border
                ttt.xyz = tttBegin + borderCell*invD;
                ttt.w = min(ttt.x, min(ttt.y,ttt.z));

                // compute the number of ray steps to skip the distance
                numSteps = ceil((ttt.w - tNow)*invStepDelta);
            }

            tNow += numSteps*stepDelta;
            float numSamples = numSteps;


            float B = 1-C2,a;
            half3  N = normalize(C2.rgb);
            half3  L = normalize( light.positionGrid - O );
            half3  E = normalize( eyePosGrid - O );
            float rgnAlpha = (1-pow(B,numSamples));
            float3 rgnColor = (phong(mat, light, N, L, E ))*rgnAlpha;

            C.rgb = C.rgb + rgnColor*(1-C.a);
            C.a   = C.a +   rgnAlpha*(1-C.a);

            // skip distance and anvance to the next voxel
            O = oEps + tNow*D;
            if (! (all(O>=float3(0,0,0)) && all(O<box.numCells)) ) break;

            // read next voxel
            cell = floor(O);
            gridSample = tex3D(texDist, cell*box.invNumCells+offs);
        }
    }

    // write results
    returnVal.color = C;
    return returnVal;
}
```

A.3 Stereo reprojection source codes

A.3.1 CG source code for the stereo reprojection with two-pass gather table generation

```
// PASS1:
// Scatter phase of two-pass stereo reprojection:
// fragment program to create relocation texture
void
stereoScatter(
            float2      texCoord    : TEXCOORD0,    // texture coords of this pixel
    out     float4      oColor      : COLOR,        // scatter info buffer
    uniform sampler2D   texPos,                     // pixel world coords
    uniform float       screenWidth,                // width ..
    uniform float       halfScreenWidth,            // and half width of the screen
    uniform float2      invTexSize,                 // 1/(bufferSize.x,bufferSize.y)
    uniform float4x4    matRight                    // right eye reprojection matrix
)
{
    float3 point = tex2D( texPos, texCoord ).xyz;       // world coords of the pixel
    float  mask = tex2D( texPos, texCoord ).a;          // 0 : invalid point
    float4 transPos = mul(matRight, float4(point,1) );  // reproject to right eye
    float newPos = transPos.x/transPos.w;               // project to clip coords
    newPos = (newPos+1) * halfScreenWidth;              // convert to screen coords
    newPos = int(newPos);

    // world coords of the previous pixel
    float3 pointPrev =  tex2D( texPos, texCoord-float2(invTexSize.x.0) ).xyz;
    // is it valid , our out of scene point ?
    float  maskPrev = tex2D( texPos, texCoord-float2(invTexSize.x.0) ).a;

    // reproject to right eye
    float4 transPosPrev = mul(matRight, float4(pointPrev,1) );
    // convert to screen coords
    float newPosPrev = transPosPrev.x/transPosPrev.w;
    newPosPrev = (newPosPrev+1) * halfScreenWidth;
    newPosPrev = int(newPosPrev);


    float2 lineInfo;
    float len;
    bool prevValid = (maskPrev>=0);
    if ( prevValid && newPos>newPosPrev+1)
    {
        lineInfo = float2(newPosPrev, newPos+1);  // bad pixel range
        len = newPosPrev-newPos;
    }
    else
    {
         lineInfo = float2(newPos, newPos+1);     // valid reprojection
         len = 1;
    }

    // remove out–of–screen lines (make zero–length)
    if ( newPos<0 || newPos>screenWidth ) lineInfo = float2(0,0);

    // write scatter info to buffer
    oColor = float4(lineInfo.x,lineInfo.y, lineInfo.y-lineInfo.x,0);
}



// PASS2:
// Gather phase of two–pass stereo reprojection:
// Vertex program to create relocation texture
// Gets line primitives and transforms them to reprojected spans
// by using the results of stereoScatter().
void
stereoGatherVP(
            float2      pixelPos: TEXCOORD0,    // texture coords of this pixel
            float3      pos     : POSITION,     // clip coords of the pixel
    uniform float       invHalfScreenWidth,     // half screen width
    uniform sampler2D   texReloc,               // scatter info texture
    out     float4      oPos    : POSITION,     // scattered clip coords
                                                // (to stereoGatherFP)
    out     float3      oTex0   : TEXCOORD0     // relocation info (to stereoGatherFP)
)
{
    float4 reloc = tex2D( texReloc, pixelPos ); // read relocation info

    oPos.zw = float2(0,1);                      //
    oPos.xy = pos.xy;                           // init output clip coords
    oPos.x  = (pos.z)==0 ? reloc.x : reloc.y;   // if the first point of the line,
                                                // set span start to x
                                                // otherwise set span end to x
    oPos.x = (oPos.x)*invHalfScreenWidth - 1.0f;// convert to clip coords

    oTex0.xy = pixelPos;                        // address of the source pixel
    oTex0.z  = reloc.z;                         // length of the span (in pixels)
}

// Gather phase of two–pass stereo reprojection:
// Fragment program to create relocation texture
void
stereoGatherFP(
            float3 texCoord0    : TEXCOORD0,// (from stereoGatherVP)
        out float4 oRelocInfo   : COLOR )   // relocation info buffer
{
    oRelocInfo = float4( texCoord0, 1 );    // get relocation info
                                            // from stereoGatherVP
    oRelocInfo.z = texCoord0.z>1.01f ? 0    // keep old values
                                            // (possible reprojection problem)
                                     : 1;   // reproject (reprojection valid)
}


// PASS3:
// Pixel peprojection by using relocation texture
// (same for one–pass and two–pass methods)
void
stereoRelocate(
            float2      wpos    : TEXCOORD0,// texture coords of the pixel
    out     float4      oColor  : COLOR,    // reprojected pixels

    uniform sampler2D   texRelocInfo,       // relocation info (from previos pass)
    uniform sampler2D   texBuffer,          // source pixels (to be reprojected)
    uniform float2      invScreenSize       // 1/(screenSize.x,screenSize.y)
)
{
    float4 texCoord = tex2D( texRelocInfo, wpos );  // read relocation info
    if (texCoord.z > 0)                             // if reprojection is valid,
        oColor = tex2D( texBuffer, texCoord.xy );   // reprojected pixel =
                                                    // source pixel
    else                                            // otherwise,
        discard;                                    // do not reproject
}
```

A.3.2 GLSL source code for the reprojection with one-pass gather table generation

```
// PASS1:
// Vertex program to create relocation texture
void main()
{
    gl_Position = gl_Vertex;  // pass thru the vertex
}


// Geometry program to create relocation texture
// Works on point primitives instead of line primitives
#version 120
#extension GL_EXT_geometry_shader4 : enable

uniform vec2        invScreenSize;      // 1/(screenSize.x,screenSize.y)
uniform float       invHalfScreenWidth; // 2/(screenSize.x,screenSize.y)
uniform vec2        invTexSize;         // 1/(bufferSize.x,bufferSize.y)
uniform float       screenWidth;        // width of the screen (screenSize.x)
uniform float       halfScreenWidth;    // half width of the screen (0.5*screenSize.x)
uniform mat4        matRight;           // right eye reprojection matrix
uniform sampler2D   texPos;             // pixel world coordinates

void main(void)
{
    // last two coords keep the tex coords of the current pixel
    vec2    texCoord    = gl_PositionIn[0].zw;
    // read world coords of the pixel
    vec3    point       = texture2D( texPos, texCoord ).xyz;
    // convert to screen coords
    vec4    transPos    = matRight * vec4(point,1);
    float   newPos      = transPos.x/transPos.w;
    newPos = (newPos+1) * halfScreenWidth;
    newPos = int(newPos);

    // read world coords of the previous pixel
    vec3    pointPrev   = texture2D( texPos, texCoord-vec2(invTexSize.x.0) ).xyz;
    // valid or out of scene point ?
    float   maskPrev    = texture2D( texPos, texCoord-vec2(invTexSize.x.0) ).a;
    // reproject previous pixel
    vec4    transPosPrev= matRight * vec4(pointPrev,1);
    // convert to screen coords
    float   newPosPrev  = transPosPrev.x/transPosPrev.w;
    newPosPrev = (newPosPrev+1) * halfScreenWidth;
    newPosPrev = int(newPosPrev);


    vec2 lineInfo;
    bool prevValid = (maskPrev>=0);
    float len;
    if ( prevValid && newPos>newPosPrev+1) // bad pixel range
    {
        lineInfo = vec2(newPosPrev, newPos+1);
        len = newPos-newPosPrev;
    }
    else                                   // valid reprojection
    {
        lineInfo = vec2(newPos, newPos+1);
        len = 1;
    }

    // generate line primitive
    lineInfo = lineInfo*invHalfScreenWidth;
    gl_TexCoord[0]  = vec4(texCoord.xy,len,1);
        gl_Position = vec4( lineInfo.x-1, gl_PositionIn[0].y, 0,1);
        EmitVertex();

        gl_Position = vec4( lineInfo.y-1, gl_PositionIn[0].y, 0,1);
        EmitVertex();

        EndPrimitive();

}

// Fragment program to create relocation texture
// same as stereoGatherFP()
void main()
{
    // get relocation info (coming from the above geometry program)
    gl_FragColor = vec4(gl_TexCoord[0].xyz,1);
    gl_FragColor.z = gl_TexCoord[0].z>1.01 ? 0   // keep (invalid)
                                           : 1.0;// reproject
}

// PASS2:
// Pixel peprojection by using relocation texture
// ( exactly same as stereoRelocate() )
```

VITA

PERSONAL INFORMATION

Surname, Name:	Es, Ş. Alphan
Nationality:	Turkish (TC)
Date and Place of Birth:	February 14, 1975, Ankara
Marital Status:	Single
Phone:	+90 312 2101050 - 1186
Fax:	+90 312 2101315
email:	alphan.es@gmail.com

EDUCATION

Degree	**Institution**	**Year of Graduation**
MS	METU,Computer Engineering	2000
BS	Ege University, Computer Engineering	1996
High School	Bursa Erkek	1991

WORK EXPERIENCE

Year	**Place**	**Enrollment**
1997- Present	TÜBİTAK UZAY	Chief Researcher
1995 July	Ege University	Intern Engineering Student
1994 July	BİSAŞ	Intern Engineering Student

FOREIGN LANGUAGES

English

PUBLICATIONS

Journals

1. Alphan Es and Veysi İşler, "Accelerated Regular Grid Traversals Using Extended Anisotropic Chessboard Distance Fields on a Parallel Stream Processor ", Journal of Parallel and Distributed Computing, 67(11):1201-1217, 2007.
2. Hacer Yalım Keleş, Alphan Es, and Veysi İşler, "Acceleration of Direct Volume Rendering with Programmable Graphics Hardware", The Visual Computer, 23(1):15-24, 2007.
3. Tülin Taner, Semra Ciger, Hakan El, Derya Germeç, and Alphan Es , "Evaluation of dental arch width and form changes after orthodontic treatment and retention with a new computerized method", American Journal of Orthodontics and Dentofacial Orthopedics, 126(4):464-475, 2004.

International Conferences

1. Alphan Es and Veysi İşler, "GPU Based Stereoscopic Ray Tracing", Proc. of ISCIS 07, 2007. *(best paper award).*
2. Alphan Es, Hacer Yalım Keleş, and Veysi İşler, "Accelerated Volume Rendering with Homogeneous Region Encoding using Extended Anisotropic Chessboard Distance on GPU", Proc. of EGPGV'06, pp.67-73, 2006.
3. Ali Telli and Alphan Es, "Link Analysis For BILSAT-1", Proc. of IEEE Aerospace Conference, pp. 6-, 2006.
4. Ali Telli and Alphan Es, "Visualization of Global BILSAT-1 VHF Frequency Usage", Proc. of RAST 2005, pp. 456-460, 2005.
5. Alphan Es and Veysi İşler, "Acceleration of Regular Grid Traversals Using Extended Chessboard Distance Transformation on GPU", Proc. of CAD/CG 2005, pp. 434-441, 2005.
6. Ali Telli and Alphan Es, "Global BILSAT-1 VHF Frequency Usage Visualization", Proc. of ICSSC 2005, 2005.
7. Alphan Es and Veysi İşler, "Three Dimensional Computer Animation For Presenting Weather Forecast", Proc. of WSCG'99, 1999.

8. Alphan Es and Veysi İşler, "Simplification of Triangular Meshes Using Iterative Edge Contractions", Proc. of ISCIS-99, 1999.

National Conferences

1. Alphan Es and Veysi İşler, "Üç Boyutlu Bilgisayar Animasyonu ile Hava Tahmininin Sunulması", (in Turkish), Proc. of Bilisim-98, 1998.
2. Alphan Es, Erdem Ayvaz and Veysi İşler "Internette 3. Boyut Teknolojisi", (in Turkish), Proc. of INET-TR 97, 1997.

HOBBIES

Playing music, painting, 3D modeling and digital sculpting, architecture, movies, tennis.