DYNAMIC VOXELIZATION TO AID ILLUMINATION OF REAL-TIME SCENES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

BORA YALÇINER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF GAME TECHNOLOGIES

AUGUST 2016

Submitted by Bora Yalçıner in parital fulfillment of the requirements of the degree of **Master of Science in the Department of Modelling and Simulation Middle East Technical University** by,

Prof. Dr. Nazife Baykal
Director, **Graduate School of Informatics** ________________

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Head of Department, **Modelling and Simulation** ________________

Assist. Prof. Dr. Yusuf Sahillioğlu
Supervisor, **Computer Engineering** ________________

Examining Committee Members

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Modelling and Simulation, Middle East Technical University ____________

Assist. Prof. Dr. Yusuf Sahillioğlu
Computer Engineering, Middle East Technical University ____________

Prof. Dr. Bülent Özgüç
Computer Engineering, Bilkent University ____________

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

Assoc. Prof. Dr. Sinan Kalkan
Computer Engineering, Middle East Technical University ____________

Date: 31.08.2016

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: BORA YALÇINER

Signature :

ABSTRACT

DYNAMIC VOXELIZATION TO AID ILLUMINATION OF REAL-TIME SCENES

Yalçıner, Bora
M.Sc., Department of Modelling and Simulation
Supervisor : Assist. Prof. Dr. Yusuf Sahillioğlu

August 2016, 79 pages

In this thesis, we focus on approximating indirect illumination on real-time applications to visualize realistic scenes. In order to approximate indirect illumination we provide a fast sparse voxel tree structure for highly dynamic scenes. Our system tries to cover traditional real-time animation methods including dynamic non-deforming objects and objects that deform with bone transformations. The voxel scene data structure is designed for fully dynamic objects and eliminates the voxelization of the dynamic objects per frame which in turn facilitates efficient realistic rendering. We combine this new scene information structure with the widely used real-time rendering techniques and these techniques' data structures such as shadow mapping and deferred rendering to provide an efficient cone ray-casting algorithm that achieves global illumination in real-time.

Keywords: computer graphics, global illumination, cone-tracing

ÖZ

GERÇEK ZAMANLI SAHNELERIN ISIKLANDIRILMASINA YARDIMCI, DINAMIK VOXELLESTIRME TEKNIKLERI

Yalçıner, Bora
Yüksek Lisans, Oyun Teknolojileri
Tez Yoneticisi : Yrd. Doç. Dr. Yusuf Sahillioğlu

August 2016, 79 sayfa

Bu tezde, gerçek zamanlı sahnelerdeki dolaylı ışıklandırmayı yaklaşık olarak hesaplama üzerine bir çalışma gerçekleşmiştir. Dolaylı ışiklandırmayı gerçek zamanda hesaplamak için hızlı hesaplanabilien ve çok sayıda dinamik objeleri destekleyen bir voxel veri yapısı tekniği sunuyoruz. Bizim sunduğumuz bu sistem hali hazırda gerçek zamanlı bilgisayar grafiği animasyon metodlarını; örnek olarak, dinamik defrome olmayan objeleri ve iskelet deformasyonuna mağruz kalan objeleri kapsamaktadır. Bizim ortaya koyduğumuz voxel veri yapısı baştan sona dinamik objeler göz önünde tutularak tasarlanmıştır ve her karede dinamik objelerin voxelleştimesine gerek yoktur. Biz yukarıda bahsettiğimiz bu yeni veri yapısını şuanda kullanılmakta olan gerçek zamanlı sahne gerçekleme teknikleri ile beraber elverişli bir koni şeklinde bir ışın fırlatma algoritması ile dolaylı işiklandırma hesaplamısını gerçek zamanda gerçekleştirmeyi amaçlıyoruz.

Anahtar Kelimeler: bilgisayar grafiği, sahne aydınlatma, koni şeklinde ışın izleme

dedicated to all game developers who create dreams while giving up on so many

ACKNOWLEDGMENTS

First and foremost i would like to thank to my thesis advisor Assist. Prof. Dr. Yusuf Sahillioğlu for his supervision. He accepted me as his student while i was a dire situation one year ago and him letting me to take the wheel of this research eventually lead to success.

I also would like to thank Prof. Dr. Bülent Özgüç for his last minute consultation and his care about my thesis. In addition i would like thank Prof. Dr. Uğur Güdükbay who 'indirectly' motivated me to pursue computer graphics. I took his Computer Graphics course while i was on undergraduate education. Eventually he was a jury member in my thesis defense.

I would like to thank all people, which i consider myself lucky to meet and know, during the course of this thesis in my workplace, Coastal Engineering Research Laboratory.

This includes;

my neighbours Nilay Doğulu, Ayşe Duha Metin, for all the 'coffe' that we grab thorough the duration of this thesis. Without them, this course of this research will be much more dull and distant. In addition to that i also thank Nilay Doğulu to show me how to be a high-class PhD student. Her insight about anything related to academic career definitely inspired me. Even if I train myself to be at least half of her professionally, I deem myself suitable for any PhD program and eventually for an academic position if i choose to pursue that career path. Finally, I thank Ayşe Duha Metin for her lively nature which made me revitalized when I was on a bad day. Seeing her smile every day probably made me finish this thesis couple of days early. However, like every rose, she has her thorns. Her constant tease was itchy, yet enjoyable;

little 'elder' sister Naeimeh Sharghivand for making this voyage emotionally bear-

able. She always pull me out of the depressive mood that I constantly get into.

'friend' (unfortunately only friend) Cağıl Kirezci and Ebru Demirci (soon to be Kirezci) to show me how a couple converges towards happiness;

Duygu Tüfekci for making my latter lunchtimes more enjoyable;

Gökhan Güler for showing me different places to eat throughout the city;

my father Prof. Dr. Ahmet Cevdet Yalçıner, who initiated my work in the lab, for his constant aid thought this thesis and more importantly supporting my major change from Civil Engineering to Computer Science seven years ago;

workplace room-mate Deniz Can Aydın, co-chef Merve Bilgin, Gözde Güney Doğan, Ezgi Çınar, lab dance queen Nuray Sefa and all other members of the Coastal Engineering Research Lab for teaching me how a research lab/institute should be like, socially.

I also would like to thank to my friend Nail Akıncı for his visual insight and all my other friends for their support.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

DEDICATON vi

ACKNOWLEDGMENTS vii

TABLE OF CONTENTS ix

LIST OF TABLES xii

LIST OF FIGURES xiii

CHAPTERS . . . .

1 INTRODUCTION 1

1.1 Motivation 1

1.2 Objective of the Study and Contribution 2

1.3 Outline 3

2 BACKGROUND AND PREVIOUS WORK 5

2.1 Rendering Equation 5

2.2 Off-line Illumination Methods 7

2.2.1 Monte Carlo Path Tracing 7

2.2.2 Radiosity and Virtual Point Lights 8

2.2.3 Photon Mapping 8

2.3 Real-time Illumination Methods 9

2.3.1 Global Illumination 10

2.3.2 Deferred Lighting 12

2.3.3 Ambient Occlusion 13

2.3.4 Reflection 14
2.4 Computer Animation Methods 15
2.4.1 Non-Deformed Objects with Transformations . . 16
2.4.2 Bone Deformed Objects 16
2.4.3 Vertex Morph Targets 17
2.5 Importance of Screen Space in Real-Time Graphics 17
3 VOXEL CONE TRACING 19
3.1 Overview 19
3.2 Voxel Information 19
3.2.1 Anisotropic Voxels 20
3.3 Cone Tracing 20
3.3.1 Ambient Occlusion 21
3.3.2 Smooth Shadows 22
3.3.3 Global Illumination 22
4 PROPOSED METHOD 25
4.1 Overview and Differences over Voxel Cone Tracing 25
4.2 Voxelization 26
4.3 Cascaded Voxel Grid 28
4.4 Sparse Voxel Octree (SVO) 31
4.5 SVO Reconstruction 33
4.6 Cone Trace on the SVO 35
4.7 Voxel Transformations & Applicability of the Common Animation Methods 37
4.8 Memory Usage 40
4.9 Method Limitations 40
5 RESULTS 43
5.1 Scene Results 43
5.1.1 Sponza Atrium 44
5.1.2 Sibernik Cathedral 50
5.1.3 Cornell Box 54

5.2 Dynamic Object Scalability 57
5.3 Real-time Applicability 57
6 CONCLUSIONS 59
6.1 Conclusion 59
6.2 Future Work 59
REFERENCES 63
APPENDICES 69
A Thesis Runtime Information 71
B GPU Friendly Graphics (GFG) File Format 77

LIST OF TABLES

Table 5.1 Sponza Timings 45

Table 5.2 Sibernik Timings 51

Table 5.3 Cornell Box Timings 55

LIST OF FIGURES

Figure 2.1 Monte Carlo Rendering Illustration 7

Figure 2.2 Photon Mapping Illustration 9

Figure 2.3 LPV propogation step 11

Figure 2.4 Deferred Rendering G-Buffers 12

Figure 2.5 Screen Space Ambient Occlusion 13

Figure 2.6 Schoastic Screen Space Reflections 15

Figure 2.7 Joint Deformed Mesh 16

Figure 4.1 Voxelization Illustration 27

Figure 4.2 Voxel Cache Data Structure 28

Figure 4.3 Cascaded Voxel Grids 29

Figure 4.4 Voxel Grid Page System 29

Figure 4.5 Inside Page System 30

Figure 4.6 SVO Data Structure 32

Figure 4.7 SVO Array View 32

Figure 4.8 Semi-Sparse SVO View 33

Figure 4.9 Cascaded SVO Example View 34

Figure 4.10 SVO node location definition 35

Figure 4.11 Point and Linear Sampling Differences 36

Figure 4.12 SVO centered node sampling 37

Figure 4.13 Cone Tracing 38

Figure 4.14 Nyra Voxel Animation 39

Figure 4.15 Voxel Snap Problem 42

Figure 4.16 Voxel Holes 42

Figure 5.1 Deferred Rendering Scaling 44
Figure 5.2 Sponza Coverage 45
Figure 5.3 Sponza Scaling 46
Figure 5.4 Sponza Ambient Occlusion 47
Figure 5.5 Sponza Real-Time Reflections 48
Figure 5.6 Sponza Sphere Reflection 49
Figure 5.7 Sibernik Coverage 51
Figure 5.8 Sibernik Scaling 52
Figure 5.9 Sibernik Ambient Occlusion 52
Figure 5.10 Sibernik Real-Time Reflections 53
Figure 5.11 Cornell Box Coverage 55
Figure 5.12 Cornell Box Scaling 56
Figure 5.13 Cornell Ambient Occlusion 56
Figure 5.14 Dynamic Object Scaling 58
Figure 5.15 Cone Tracing Timings with respect to Cone Aperture 58

Figure 6.1 Voxel Weighted Grid Splat 60

Figure A.1 Working Directory Layout 71
Figure A.2 Light Bar 73
Figure A.3 Thesis Bar 75
Figure A.4 Cone Bar 75

Figure B.1 GFG Maya Export Options 78

CHAPTER 1

INTRODUCTION

1.1 Motivation

Main motivation of this thesis is to provide potential solution to the global illumination problem in current games. Recently available graphics hardware is extremely powerful and capable of rendering photo-realistic scenes with ease. However, simulating indirect illumination in real-time is still a challenging task, especially for dynamic lights and animated objects.

Most of the modern games use static light maps generated using off-line rendering techniques. The light map quality is visually pleasing however it is nowhere near fast enough to calculate in real-time, which makes the calculation static for both the light and the objects. This kind of approach is beneficial if the scene light is static and it can only be applied to static objects in the scene. For dynamic objects different approaches can be applied and those will be discussed in Chapter 2.

Solving the rendering equation in real-time is plausible; anyhow, resolution and polygon compromises will make the consumer shy away from playing the game. Modern games rasterizes nearly millions of triangles and determine color of millions of pixels in mere milliseconds, which makes applying polygon or screen space bound algorithms challenging task. Recently, 4K (3840 x 2160) resolution displays have become available; thus this technological advancement indirectly created demand to support it in respectable frame rates. Most of the today's games are pixel bound since most of scene rendering calculation done on screen space. Deferred Rendering, ambient occlusion and reflections are all done in screen space as a post processing effect in recent games. This sudden increase in pixels made respectable drops of frame rates on games.

In this thesis we consider approximating the scene with voxels which introduces voxel count bounded algorithm. Since the voxels are only used for indirect illumination and reflection it can be adjusted to achieve the desired frame rates. However, sampling the voxel structure is pixel bound because each pixel should trace series of cones and combine the results of those cones. Still, reduction of resolution will not be as noticeable because voxel structure is used to calculate only the indirect lighting.

1.2 Objective of the Study and Contribution

In this thesis we propose a more dynamic object firendly methodology on cone-tracing method proposed by Cyril Crassin [1]. Crassin's method relies on voxelizing dynamic objects in real-time, however our method introduce application of traditional animation methods to the voxelized objects per-voxel basis. These traditional animation methods include transformations and joint based deformers. Our method is more huge scene friendly with its camera bound dynamic voxel grids. In addition to that, supports multiple nested voxel grids which enables higher fidelity near the camera and fidelity decreases with respect to the distance of the camera. All of these multiple voxel cascades are included in the single sparse voxel octree (SVO).

Proposed approach supports fully dynamic objects and lights as long as the light has its shadow maps [2] generated. Using these shadow map info and voxel grid structure, it traces series of cones per pixel in order to approximate a solution to the rendering equation [3]. Cone tracing method is used for the SVO sampling since it provides efficient memory access when cone ray is away from the shot surface.

This methodology uses state of the art rendering techniques used by the modern games which includes shadow mapping, deferred rendering and traditional joint (bone) based deformers with animations. This enables fast integration with the modern game engines since most of the data is already available. This solution is designed to complement the traditional direct rendering pipeline which lacks dynamic indirect illumination support. In current generation of hardware it is not fast enough to entirely use this approach to calculate entire lighting of the scene with plausible results, at least in commercial games.

In this study, provided method tries to define real-time scene approximation using voxels. This voxel approximation, can be used to calculate first(or multiple if computation unit is powerful) bounces of the global illumination. However, direct illumination will still be calculated using traditional deferred lightning technique. SVO structure will be used for ambient occlusion and real-time reflections. This thesis mainly focuses on the scene approximation rather than usage of the structure. In addition, implementations and discussions of the both global illumination and ambient occlusion are provided.

This study aims to improve on the solutions or define new solutions to the voxel-cone tracing technique and sparse voxel octree structure [1] in following ways:

- Approximation of the rendering equation in real time
- Scalable support for dynamically movable objects including,
  - Rigid dynamic objects
  - Deforming objects with joint hierarchies (skeleton)
  - Deforming objects with vertex morph targets
- Open world or big scene usable method

- Scalable for multiple dynamic lights
- Real-time dynamic reflection support
- Should incorporate with already well established real-time rendering methods such as,
  - Shadow mapping [2], [4]
  - Deferred rendering GBuffer [5]
  - Physically based shading [6], [7]
  - Screen space reflections [8]

Most of the objectives stated above are covered in this thesis however, deforming objects with vertex morph targets, physically based voxels are not covered in this thesis. This voxel hierarcy only supports dynamically transformed objects and objects that are animated with joint hierarchies. Specular illumination is handled by the Phong illumination model [9]. Other improvements which are not covered by this thesis are stated as future work in the conclusion chapter.

1.3 Outline

Outline of this thesis is as follows:

Chapter 1 explains the motivation behind this research and briefly touches on what improvements have been established over the existing cone tracing method.

Chapter 2 briefly reviews the literature and specifies the methods used in computer graphics literature. Topics of the background work include global illumination methods, and computer animation methods. Illumination methods cover both real-time and off-line approaches.

Chapter 3 throughly remarks the voxel cone tracing method [1] that this study is based on.

Chapter 4 introduces the proposed method and discusses improvements and limitations over the voxel cone tracing method.

Chapter 5 explains the results obtained by using the proposed method. Sponza Atrium, Cornell Box and Sibernik cathedral scenes are chosen as test scenes. Timing graphs are provided for each parameter such as resolution, voxel count, cone angle and cone count per pixel.

Chapter 6 summarizes the proposed methodology and presents recommendations for further research.

Appendix A is the manual of the thesis source code, actual source can be accessed from here [10].

Appendix B discusses GPU Friendly Graphics (GFG), the file format used in the thesis, which is developed along with the thesis concurrently. GFG file

format supports skeletal animations, multi material meshes, mesh object space axis aligned bounding boxes and pre-formatted meshes vertex data wich can directly be used by the GPU. Source code; which includes Autodesk® Maya importer exporter and header libraries, can be found in [11].

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

In this chapter, research about interactive global illumination is provided. Methods that are discussed split into off-line and on-line solutions. Real-time solutions are only covered if they are used by an actual commercial game. The reasoning behind this is that many methodologies are considered real-time when they achieve certain calculation time threshold. However, these methodologies are actually real-time capable for simple test scenes but actual visual quality and geometric complexity required by current potential games are much more higher and competitive nature of the gaming industry market pushes the required visual fidelity to the limit.

Global Illumination is a heavily focused research area and most of the topics discussed here are only a small portion of the entire illumination field. Recently, Ritschel et al. [12] did a comprehensive research about interactive global illumination and compiled research on a single paper. It is a very compact form of information about interactive, real-time global-illumination. Most of the background research discussed in this chapter can be considered as pillar methodologies of indirect illumination.

2.1 Rendering Equation

Kajiya presented the rendering equation [3] which unified different rendering phenomena and merged them into a single equation. Rendering equation simulates the light which scatters from different surfaces in the scene. It is defined for per surface patch in the entire scene since each object contributes to the illumination of other objects with either having emissive surface properties and applying light directly or having reflective properties and indirectly transmitting light from other potential luminance points in the scene.

The rendering equation is the current norm of physically simulating the light in a scene. Most of the games and off-line applications approximates rendering equation ultimately. Since it simulates light in an accurate way, most of the rendering related research backtrack to this equation.

The rendering equation, in its original form, defined in the paper is as follows:

$$I(x, x') = g(x, x') \left[\varepsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right] \qquad \text{(Equation 2.1)}$$

Where:

- $I(x, x')$ is total intensity of light passing from point x to point x'
- $g(x, x')$ is geometry term.
- $\varepsilon(x, x')$ is light emitted x from to x'
- $\rho(x, x', x'')$ is intensity of the light scattered from x'' to x indirectly from x'.

Rendering equation can be written in terms of incoming radiance on a hemispherical integral like this;

$$L_0(p, w_0) = L_e(p, w_0) + \int_\Omega f_r(p, w_i, w_0) L_i(p, w_i) \cos\theta_i dw_i \qquad \text{(Equation 2.2)}$$

Where:

- $L_0(p, w_0)$ is total outgoing radiance from surface point p outward along the direction w_0
- $L_e(p, w_0)$ is total emitted radiance from surface point p outward along the direction w_0
- $f_r(p, w_i, w_0)$ is the bidirectional reflectance distribution function which defines the surface property of how much of the light incoming from the direction w_i is transferred towards w_0 after surface reflection, refraction and absorption.
- Ł$_i(p, w_i)$ is same as L_0 but its direction is w_i
- Ω term represents the positive hemisphere centred at surface point p.

In this thesis, the hemispherical form of the equation will be used since its definition is applicable to cone tracing method.

A can be see from the equation, its computation of is costly, it is a recursive function in which number of calls that will happen recursively is determined by the integral equation. In addition to that, function is defined for per visible surface point initially and after first bounce of the light, equation will include many other invisible surface points and will keep increasing exponentially up until desired visuals are achieved [3].

Along with the equation, Kajiya introduced solution to integral part of the equation using a Neumann series method. Integral is approximated by summing multiple rays of light for arbitrary number of w_i for some dw_i. Calculation is stopped until desired amount of irradiance is gathered.

Rendering equation can be approximated by using the Monte Carlo methodology. Monte Carlo method requires importance sampling methods in order to be practical. Kajiya introduces methods for sampling rays towards interesting parts of the domain and sparse sample areas which have minimal contribution. Further-

more, k-d space partitioning scheme is proposed to apply to the solution of the integrand. The techniques that are provided in the section "Off-line Illumination Methods" tackles the problem of solving the rendering equation in time intervals that are not considered real-time.

2.2 Off-line Illumination Methods

In this section, we will explain the off-line solutions that are considered core solutions to the rendering equation. Those techniques include Mote Carlo Path Tracing, Radiosity and Virtual Point Lights, Photon Mapping.

2.2.1 Monte Carlo Path Tracing

Monte Carlo Path tracing solves rendering equation by launching rays arbitrarily and bounces them on the scene up until enough illumination is gathered for each each surface (Figure 2.1). Illumination is generated by reaching to a light source. In order to achieve high quality results, a high number of rays needs to be sampled. As discussed above, instead of arbitrarily shooting rays, rays can be send with a logic by looking where its integrand has high values. This is called importance sampling. Importance sampling can be applied by looking to the bidirectional reflectance distribution function (BRDF) f_r or by looking at the incoming light term L_i and launch outgoing rays according. However, doing this is not a trivial task since it is not computationally easy to determine where the largest contribution of irradiance is coming from.

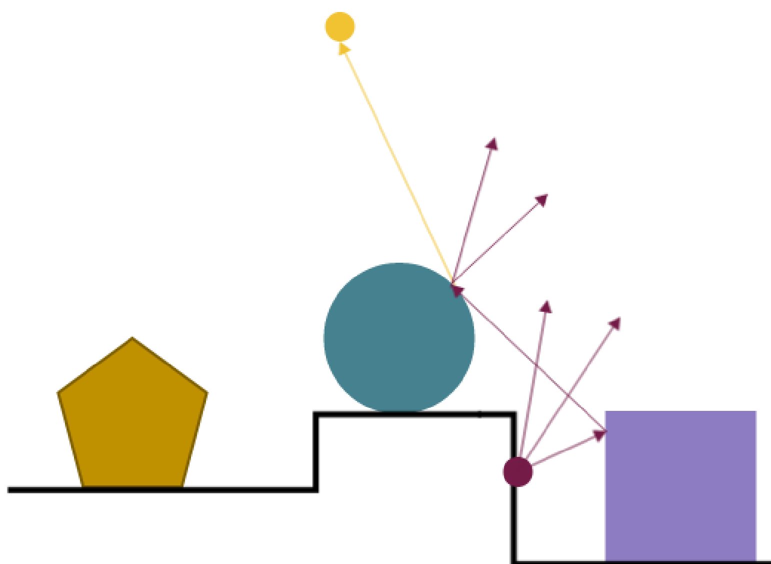

Figure 2.1: *Monte Carlo method illustration for solving rendering equation. Dark red surface point starts to shoot rays (coloured with dark red) until it finds the light source. Yellow coloured ray is light ray. Notice hight amount of rays require to render a single surface if the rays launched arbitrarily.*

That being said, to reduce time of calculation; in adition to the improtance sampling, other methods can be used. Bi-directional path tracing method launches rays from light sources as well as surfaces to make the Monte Carlo solution to converge faster [13]. Metropolis light transport algorithm [14], introduces a metropolis sampling method; which is initially proposed for handling sampling problems in computational physics, to sample the paths contributions to the image by tying to define a function f, which represents and records paths towards the image plane from light sources.

Even though Monte Carlo path tracing method provides photo-realistic imagery, itself and its improved forms are not applicable to the real-time solutions.

2.2.2 Radiosity and Virtual Point Lights

Firstly introduced by Goral et al. [15], radiosity method defines surfaces which have uniform reflectance distribution function, makes its BRDF output to be uniformly distributed over the hemisphere. Although radiosity research was conducted before the rendering equation paper, it can be considered as an approximative solution of the Rendering Equation. Perfectly diffuse functions have constant hemispherical BRDF, which simplifies some portion of the rendering equation. Moreover, solution consists of dividing scene as surfaces and determining the light interraction between those surfaces via links, and computing the interaction between the surfaces. Furthermore, radiosity solution applied to non-diffuse environments [16].

Common usage of radiosity is by deploying many points that served as virtual point lights (VPL) throught the scene [17]. Then, each surface point accumulates lighting in its vicinity by using generated points. Modern approaches utilize shadow mapping to sample VPL towards the scene [18]. Ritschel uses imperfect shadow maps [19] to use sample instant radiosity in real time speeds. Furthermore, Ritschel improves his own solution by making indirect shadows more accurate by viewpoint occluder adaptation [20]. Recently, Sun et al. [21] combined many light approach with sparse voxel octree structure and accelerated virtual point light solution SVO traversal.

2.2.3 Photon Mapping

Photon mapping [22] approaches the light transport problem differently. Photon mapping firstly launches series of photons from the lights sources of the scene and stores the bounced photons in a photon map. After photon generation, surface irradiance is calculated from those photons by either determining photon density near the surface area or doing a final gathering which illumination data gathered from each visible surface point by tracing the length (Figure 2.2).

Photon mapping relates to bi-directional path tracing when we consider the final gathering version. Instead of sending light rays towards the scene from light sources, for each iteration it sends photons and stores them in a map. Difference here is that the photons can also bounce; in contrast, bi-directional path tracing

is calculated from light source and the initial contact point. The photon map information can be cached and does not required to be calculated every frame, until either light or one of the objects of the scene is moved. Moving anything on the scene invalidates the cached data. Cached data is valid as long as every object and light is stationary.

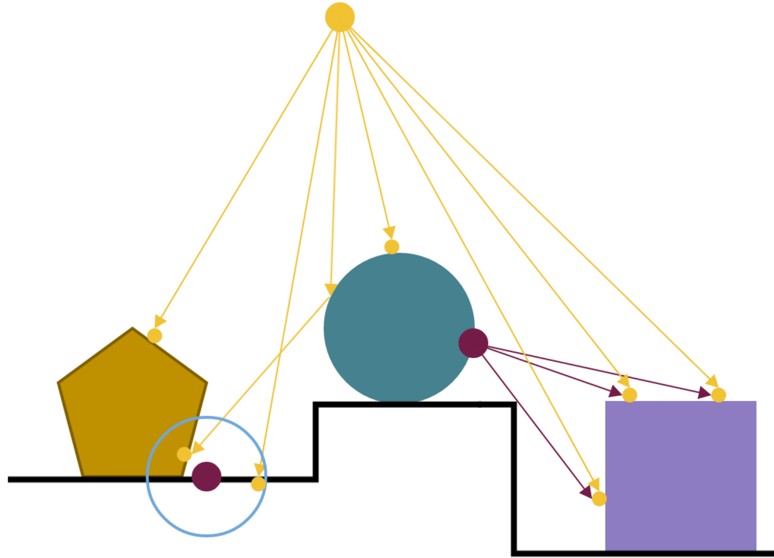

Figure 2.2: *Photon Mapping illustration for solving rendering equation. First step is the splat series of photons through the entire scene (yellow rays). Second part is to either doing a proximity check to determine total irradiance (left surface) or to sending rays to neighbouring photons (right surface).*

Photon mapping initial photon splat can be computed on the GPU using rasterization [23]–[25]. But these approaches only use for initial transfer from light to the surface. Yao et al. [26] incorporates secondary bounces using special environment maps from the scene rendered with position, normal and color data. After photon splatting, indirect illumination is calculated using deferred shading and shadow mapping.

2.3 Real-time Illumination Methods

Rendering equation represents most of the physical phenomena such as reflections, ambient occlusion and shadows; on the other hand, real-time rendering methods have separate algorithms for each phenomenon. Because of the real-time requirement of the approach, most of these algorithms are pre-calculated and stored in a cache; thus, making the solutions to work only for static objects and lights. However with additional cost, most of these algorithms are applicable to the dynamic object and dynamic light. Some of these algorithms complement each other with respect to scene dynamism. In short, we will explain the techniques that modern games use to partially solve the rendering equation with respect to reflections, illumination and shadowing.

2.3.1 Global Illumination

In this section, indirect illumination methods; which are used in modern games, will be explained. Section also covers the most used direct lighting methodology in today's games which is deferred lighting. Most of the remaining algorithms cover only diffuse indirect illumination however, glossy reflections can also be approximated. Off-line methods that are explained below can cover both direct and indirect illumination.

2.3.1.1 Light Maps

Earlier games fully relied on pre-calculated lighting, which were stored in form of a light map [27]. Still, light mapping is used for static objects and static lights because light mapping has high quality results for both static objects and lights. Only drawback is that light maps have memory requirements which can be a limitation for low memory GPUs. Most of the memory requirements are not as important as the performance and quality gain over the real-time methods unless the scene that is being rendered is large.

Lightmaps have heavy memory constraints and it has become harder to incorporate it with open world scenes. Stefanov explained the solution used in the Far Cry 3 game [28] which defines a method to work on large dynamic scenes. Based on the Sloan et al. [29] and Kirstensen et al.'s approach [30], they use pre-computed light probes which hold radiance transfer and using this data to update camera centric volume texture dynamically. They also update light probe values depending on time of day change. Their light probe approach holds only couple of megabytes of memory on the CPU. Even though entire map of the Far Cry is vast, their approach achieved minimal memory requirements. However, like other light map approaches; dynamic objects only receive light and do not contribute global illumination.

2.3.1.2 Light Propagation Volumes

Light propogation volumes (LPV) approximates the scene with a volumetric structure [31]. Direct light is sampled to this volumetric structure using Reflective Shadow Maps (RSM) [18]. This provides simple light transport calculation between grids of the volumetric structure, which makes the algorithm to be parallel for each volume grid. Moreover, it makes the algorithm to be able to work on the GPU pretty well. Furthermore, Kaplanyan et al. introduces the cascaded approach, in this approach scene is divided in to more fine grids near the camera and coarser grids away from the camera [31]. Cascaded voxel structure which will be explained in this thesis was inspired from this paper. Multiple cascades concentrates the quality near the camera, where players attention will be focused on most of the time.

Light propagation volumes have two steps. First scene geometry density is calculated for each volumetric location, using depth peeling from the camera and

reflective shadow maps. In the propagation section of the algorithm, light propagates based on its initial reflection direction (Figure 2.3).

Light propagation volume method is designed to compute low-frequency lighting in a scene like indirect illumination. It mostly covers diffuse indirect illumination, in addition to that, Kaplanyan et al. [31] discusses visually acceptable approximations on specular indirect illumination.

Computationally LPV algorithm is designed for fully dynamic lights and scenes. RSM generation is more or less like shadow-map generation with additional data. Today's modern games have very efficient shadow map rendering pipelines and thus those optimized pipelines can be used to generate RSM as well. Only data difference is here that RSM has normal information as well as depth information.

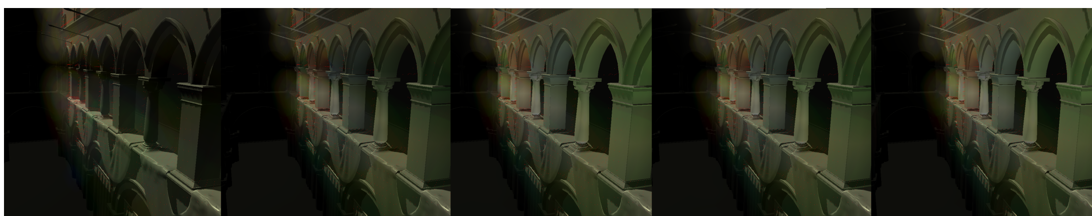

Figure 2.3: *First 4 propogation step and initial light injection step from reflective shadow maps. Images generated using ligh propogation volumes demo by Benjamin Thaut [32].*

Limitation of this approach is that LPV suffers from light bleeding since it has limited information about the geometric density of each grid. Since it has used already generated data; i.e. RSMs and Camera Depth buffer, it may not fully cover the scene with this limited geometric density data.

This approach is computationally fast for both dynamic lights and scenes. Light propagation volumes are camera centric and dynamically updated every frame and works fast on previous generation consoles (Sony Playstation® 3, Microsoft® Xbox360).

2.3.1.3 Voxel Cone Tracing

Since this research is heavliy based on this approach, voxel cone tracing method will thoroughly be explained in upcoming chapter. In short voxel cone tracing method approximates the scene geometry using voxels and storing voxel in the sparse voxel octree(SVO) structure [1], [33]. Then initial light information is splattered using hardware rasterizer using a shadow-map. Then radiance information is averaged through the top levels for higher cone angles. Finally for each screen pixels, couple of cones traverse the scene and gather illumination data for each pixel.

2.3.1.4 Light Probes

Inb real-time engines dynamic objects indirect illumination is often captured in a low-frequency solutions. Many game engines use a form of pre-calculated light

data in cache. Unity®; up until Unity® 5, used light probes [34], [35] for illuminating the dynamic objects. Unreal Engine® uses similar approach called Indirect Lighting Cache [36]. Although named differently, both implementations sample pre-calculated data on places near the scene. Cache data generated near the surfaces is finer and on other regions of the scene generation is done coarsely.

Middleware indiect illumination solution company, Enlighten, is also using light probes for dynamic objects [37]. Enlighten is integrated with most of the modern real-time rendering engines such as Dice® Frostbite and Unity®. Enlighten also uses light probes and store the outgoing radiance in frequency domain using spherical harmonics [38].

2.3.2 Deferred Lighting

Deferred rendering is not a recent discovery [5]. But only in the recent years it is started to be utilized by the game engines. One of the first implementation of the deferred shading is on Kill Zone 2 [39]. Up until late 2000s, hardware limitations prevented implementation on the GPU up until OpenGL and DirectX exposed frame buffer objects. This enabled rendering into multiple render targets, each render target holds a required information about lighting calculation. Moreover, memory cost can be quite memory intensive if illuminated material is required to have various material information. Packing screen space data increases performance since each light will fetch entire G-Buffer data for that pixel and bandwidth of that texture fetch is the main bottleneck in the deferred rendering algorithm.

This completely eliminates unnecessary calculations since you only illuminate the pixels that are already rasterized. Normal, position, color or any other value required by the lighting calculation are stored in the textures. Combination of this texture cache is called Geometry Buffer, in short G-Buffer.

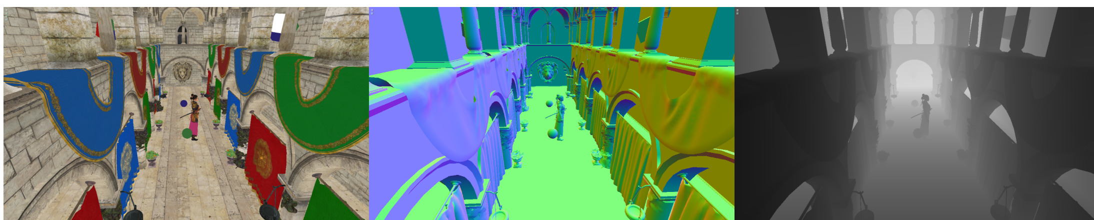

Figure 2.4: *Sponza scene G-Buffer. G-Buffer holds world space normals, unlit color and depth information in screen space. From left to right, color normal and depth values of the G-Buffer is shown.*

G-Buffer can change depending on the illumination method. Physically based material may require more data than simple illumination methods like Phong Illumination Model. In addition, all rendered pixels are required to hold normals; which can be either in screen space or in world space, pixel position and pixel albedo (unilluminated color value) (Figure 2.4). Pixel position is stored as a depth buffer value since depth buffer value can be used to determine the screen space or world space position of the pixel by unprojecting the pixel.

G-Buffer enables many other rendering algorithms like screen-space reflections.

Since G-Buffer holds position and normal data. Those two can be used to start tracing algorithms directly from the hit surface since by definition camera rays will hit those location. This accelerates initial setup of cone tracing method and increases performance.

Just like any other screen space method lighting calculation overhead is decoupled from the scene complexity. In deferred shading algorithm, lighting calculation is only bounded by pixel count on the screen.

2.3.3 Ambient Occlusion

Ambient occlusion method is first proposed by Zhukov et al.[40]. It defines equation independent from the light source, which resolves the ambient illumination by looking obscurance of vision on the scene. Below, you can find some of the implementations of this phenomenon on screen space.

2.3.3.1 Screen Space Ambient Occlusion(SSAO)

Screen Space Ambient Occlusion (SSAO) is first introduced to gaming insdustry by Crytek [41]. Crytek's implementation uses depth buffer discrepancy between neighbouring pixels by sampling multiple points around the pixel thus, determining the pixel occlusion. Implementation requires high amount of samples per pixel in order to be visually acceptable. To increase performance, implementation randomly rotates the sampling kernel, then blurs the noisy result. Method generates shadows near the contact points of the neighbouring objects (Figure 2.5).

Figure 2.5: *SSAO example from Cry Engine 2. Notice the darkened pixels near hard edges [41].*

Limitation of this method comes from its screen space nature, only close hight-frequency ambient occlusion can be calculated since it checks neighbouring pixels for occlusion. For high-frequency occlusion more broader scene information is required and screen space information does not provide that.

2.3.3.2 Horizon-based Ambient Occlusion(HBAO)

Shanmugam and Arıkan [42] introduced the idea to incorporate normals to the ambient occlusion calculation on screen space. This requires normals of the screen-space pixels which is provided by the deferred rendering technique. First they treat each pixel as a surface defined by point and a normal hence the need of normal and position for each pixel. Then each pixel samples other pixels and try to estimate occlusion.

HBAO technique further improves on that by ray marching on horizons on a radius r sphere [43]. Sphere radius is proportional to the pixel size, and horizon ray is generated by looking at tangent plane. Horizon is incremented by an angle towards the surface normal and sampled multiple times.

Iteration times can still be quite high for current generation hardware. Instead of fully sampling each pixel, sampling is done sparsely then blur pass is incorporated. Since ambient occlusion is low-frequency illumination it does not necessarily degrade visual quality as much. HBAO technique is more accurate than SSAO technique since it also incorporates surface normals into the equation. Just like SSAO, HBAO can only calculate close distance occlusion since distant occlusion information may not be available since only screen space data is available.

2.3.4 Reflection

Another phenomenon that can be achieved by the rendering equation is the object reflections. Glossy or mirror-like objects reflect incoming light away from their surfaces with minimal absorption. This creates the reflection phenomenon and it is a product of rendering equation. However, real-time applications cover this phenomenon specifically by using screen space to sample reflection points or using static environment maps to specifically sample from pre-calculated and cached data.

2.3.4.1 Screen Space Reflection

Firstly introduced by CryEngine [8], screen space local reflections method utilizes pre-generated geometry buffer; which is mainly used by deferred lighting, to trace rays from the screen space mainly using geometry buffer position and normal data. Tracing direction is towards reflection vector and tracing is done by doing ray-marching algorithm (Figure 2.6.

Ray traversing can be accelerated by using hierarchical depth buffer which is mipmapped depth buffer and each parent holds the most close distance among its children [45]. After that ray traversing is not necessarily required to march in constant steps and rays can skip multiple depth sample points with one mipmapped depth buffer lookup. Moreover, neighbouring rays can be used to reduce number of rays required to be traced in order to get realistic results [44]. Cone tracing method can also be incorporated for sampling the reflection vector [45],

Figure 2.6: *Screen Space Reflection implementation by Stachowiak et al. [44]. Notice that camera is specifically put on a place to perfectly cover surfaces reflection points by the screen space.*

[46]. Geometry buffer color values are convolved in to lower mipmap chains of the texture then sampled from appropriate level of mipmap depending on the cone opening.

Most important fact about screen space reflection is that it supports dynamic geometry in high-frequency form. Because of that even mirror like objects can be approximated with this approach with minimal computational cost.

2.3.4.2 Environment Probes (Environment Map)

Environment mapping is a solution to the specular reflections on the scene. Most of the objects in the scenes require reflection effects in order to be visually compelling. Debevec's method proposed image based lighting method which is baseline of this approach [47]. Today's modern games render static objects in the scene into a cube map. There are multiple cube maps laid out on the scene which is used by the GPU to approximate reflections on the object. Limitation of this method is that only static object's reflection can be stored on the image since environment cube maps are pre-calculated. Secondly cube maps require high amounts of memory and in order to be visually consistent with the scene, environment maps are required to be probed to various places of the scene. This improves the memory cost.

Another limitation of this method is that it is not pixel perfect accurate since environment probe is sampled from a space that is not exactly used by the surface. This introduces inaccuracies however for low-frequency reflections method is usable.

2.4 Computer Animation Methods

In this section we will provide traditional animation methods that re used in today's modern games. These methods provided here are supported by the sparse voxel octree structure provided in this thesis.

2.4.1 Non-Deformed Objects with Transformations

This is the most basic animation technique. Single transformation is applied to the entire rigid body, transformation is changed per frame by a key-frame chain interpolated to the current time-step. In addition to key-frame chain, this kind of animation can be triggered by physics system.

Non-deformed animation is mostly used by rigid bodies, like bullets or scene solid moving objects like barrels and boxes.

2.4.2 Bone Deformed Objects

Used in mostly humanoid objects, bone deformed objects are animated by a bone or joint chain. Each object vertex has weights of adjacent bones. Each vertex holds constant amount of weights and all of the weights adds up to one. For each frame, weighted transformation occurs for each vertex per time frame. Skeleton structure holds key-frame chain for each of its joint rotation, in addition to that root translation key-frame chain is also stored.

Transforming vertices using multiple bone transformation is called skinning. Skinning results in organic transformations near the skeleton joints which makes the method suitable for organic objects (Figure 2.7).

Figure 2.7: *Nyra character; which is used in the thesis, doing her jumping animation. Coloured hierarchy formed by joints and drives the movement of the base character. All of the vertices of the object has four weights which is used to deform the object using the most influencing nearby joints.*

Just like non-deformed object animation, physics system can solve bone transformation and trigger animations in this system. This is called rag-doll physics. Most of the modern games uses deformed objects not only for humanoid but also rigid object with moving parts like cars, helicopters. It is more convenient to use single modelled object than constructing the scene hierarchy in the engine.

2.4.3 Vertex Morph Targets

Even though it can be applied to many different animation problems, vertex morph targets are mostly used for facial animation in real-time graphics. Instead of having third party helpers like joints, each vertex holds their position as a key-frame. Then interpolation occurs between those key-frames.

For real-time graphics its operation is costly and has high memory constraints if vertex size is high. Because of these reasons it is used for facial animations in games, more high-quality version of morph targets used in cinematic cut-scenes.

2.5 Importance of Screen Space in Real-Time Graphics

Screen space effects enabled visually complex scenes, since all of the screen-space effects are independent from the scene geometry. As long as program pushes triangles to the hardware efficiently, modern graphics hardware do not choke under high triangle loads. Screen space algorithmic complexity depends on the screen pixel counts because of that, sudden increase in pixel density; i.e. 4K displays, greatly effects performance of modern games. Other limitations of the screen space implementations are that they do have limited information about the scene. Only visible pixels hold information in screen space and with this limited information, algorithms cannot completely solve the rendering problems that it is designed for, thus making screen space effects reliable for close distance interactions between surfaces.

If we summarize;

Pros:

- Independent from scene complexity
- Inherently acceleratable by the GPU
- No redundant calculations needs to be made

Cons:

- Pixel bound
- only visible data is available for calculation

CHAPTER 3

VOXEL CONE TRACING

3.1 Overview

All of this chapter is about Crassin's PhD thesis [1] which is about the voxel octree structure and voxel cone tracing method. Cone tracing method is first introduced by Amantadies et al. [48]. In the original paper, intersection of the cones are done between objects and the cone and each intersection provides coverage ratio and intersection boolean result. Crassin's approach provides cone tracing method which is accelerated by sparse voxel octree (SVO) structure.

Other key points of the proposed method which will be discussed in this chapter are hardware linear interpolation support, anisotropic voxels and N^3-tree [49] utilization.

3.2 Voxel Information

Crassin's method splits voxel information into two separate parts. Brick maps and node pointers. Node pointers are in the form of N^3-tree. N^3-tree is generic form of octree. Octree is the special case of N^3-tree where $n = 2$. N^3-tree is very useful for holding voxel corner data in the brick structure since it prevents memory repetition.

Brick map portion of the data structure holds the actual surface parameters. In order to utilize hardware trilinear interpolation, bricks are stored in texture memory. Brick resolution is determined by the N value of the N^3-tree. For $N = 2$ each brick holds $2 \times 2 \times 2$ voxel data. Brick map resolution here plays a important role about data redundancy. In his thesis, Crassin analysed the memory cost of the different N values, and different voxel storage schemes. These schemes are about storing voxels at he center of the nodes or the corner of the nodes and storing these nodes with or without having border values. Border values are required to sample consistently in run-time without tracing the tree for the neighbouring values. Increasing the N value of the N^3-tree also reduces the data dependency between bricks in terms of averaging and sampling.

However, utilizing higher N counts increases the averaging complexity since instead of using box filter N covered Gaussian filter is required to be used. In

contrast, having higher N values reduces the construction complexity since voxels that are not at the border of the brick are not required to be exchanged between brick nodes in order to get correct linear interpolation results.

Crassins method constructs the tree off-line for static objects and utilizes fast dynamic voxelization method for dynamic objects [33]. Dynamic objects then are splattered into the static octree in run-time.

Choice of illumination method in the Crassins PhD thesis was Phong illumination [9]. In order to represent the surface normal over a voxel volume, normals are defined using a normal distribution function (NDF). Based on Toksvig's approach [50], normals are filtered as Gaussian normal distribution with a direction vector $\vec{D}$ and a standard deviation of σ.

3.2.1 Anisotropic Voxels

Crassin suggested the usage of anisotropic voxels for internal nodes of the tree. For thin objects; i. e. wall with a single voxel depth, averaging will result on same value for all the directions of the voxels but in the example above thin wall should still have same color value for the direction that the wall is facing. In order to accomplish that, voxel holds its information anisotropically by holding a value for all directions of the voxel and interpolates them according to the incoming direction.

Storing voxels anisotrophically means additional memory cost. To be precise, voxels will store six times more data than a isotropic version since each voxel has six faces. However this is required only for the intermediate levels because bottom level does not undergo averaging process.

Anisotropic voxels increase memory cost of each voxel by six. However for thin layer objects (i.e. walls, thin cloth) this approach greatly increases visual fidelity. In our method anisotropic voxel was not adopted since we construct the tree per frame.

Linear interpolation friendly tree requires different averaging process in order to be visually consistent. Each node is required to fetch 8 neighbouring voxels on the different bricks. This means 8x more tree traversal for each average process. In construction time Crassin uses neighbouring pointers for each brick to traverse to the neighbouring nodes fast. Still this means extra process if we consider per frame voxel octree construction. However tracing has to be done in every frame but SVO construction can be done per multi-frame. It will introduce delayed indirect lighting effects however, those effects only complement the main image hence results should be visually acceptable.

3.3 Cone Tracing

Another novel approach in Crassin's thesis is instead of launching rays through the scene, method launches cones instead of rays. Cones have useful properties

to take advantage of SVO structure. First, cones have increasing aperture when the length of the cone increases. This enables to sample lower levels of the SVO when the cone has higher heights which enables reduction of tree traversal. Since cones have actual volumes; unlike rays, minimal amount of cones will be required to cover most half sphere needed to be integrated per surface. Amount of cones required depends on the cone angle. Only couple of high angle cones are needed to cover entire half sphere.

Now, we will explain problems that can be solved by using voxel cone tracing technique. Voxel cone tracing technique accelerates light ray tracing because of that, it can be used to cover entire rendering equation phenomena including reflections, ambient light occlusion, indirect diffuse lighting.

3.3.1 Ambient Occlusion

First we start by examining ambient occlusion equation, because it is simpler case than rendering equation. We take the hemispherical version of the Zhukov's equation [40].

$$A(p) = \frac{1}{\pi} \int_{\Omega} V(p, w) \cos\theta dw \qquad \text{(Equation 3.1)}$$

Where:

- $A(p)$ is total occlusion at the surface point p
- $V(p, w)$ is visibility term which returns zero or one depending on the rays occlusion.
- θ is angle between the ray and the surface point normal

In order to compute integral efficiently, integral is split into smaller integrals where each resembles a cone [1].

$$A(p) = \frac{1}{N} \sum_{i=1}^{N} \int_{\Omega_i} V(p, \theta) \cos\theta d\theta \qquad \text{(Equation 3.2)}$$

In this equation each integration represents a cone with aperture of θ. In closer distances from the surface, we can consider the $\cos\theta$ term constant since angle difference is minimal[33]. Visibility function now takes θ instead of w. Resulting integral will return a value between zero and one. One meaning fully visible and zero meaning fully occluded. Even though this equation does not have distance factor incorporated to it Crassin suggested that enclosed scenes visibility is limited by the distance and should be incorporated into the equation. Therefore, each sample decays over the distance $\frac{1}{1+\lambda r}$ [33].

3.3.2 Smooth Shadows

Direct light shadowing can be achieved with cone tracing. For each surface point, Crassin expressed that sending a single cone towards to the light source from the surface will be enough for testing light occlusion. Just like ambient occlusion we trace the visibility by sampling voxel density of each cone throughout the scene. Result of the shadows will be smooth and seamless.

3.3.3 Global Illumination

Just like ambient occlusion evaluating the integrand with accumulation of multiple cones can be used for rendering equation. Rendering equation;

$$L_0(p, w_0) = L_e(p, w_0) + \int_{\Omega} f_r(p, w_i, w_0) L_i(p, w_i) \cos\theta dw_i \qquad \text{(Equation 3.3)}$$

can be expressed as series of summations just like ambient occlusion equation;

$$L(p, w_0) = L_e(p, w_0) + \frac{1}{\pi N} \sum_{i=1}^{N} \int_{\Omega_i} f_r(p, \theta, w_0) L_\theta(p, \theta) \cos\theta d\theta \qquad \text{(Equation 3.4)}$$

Emitting light is kept as a ray since implementation will only approximate the initial bounce from the surfaces. Outgoing ray will go towards the camera pixel. One of the limitations of the cone tracing is expressed here. It is good for low-frequency illumination, like indirect illumination. It is still usable for high-frequency illumination but resulting image will be blurry. In order to apply high-frequency illumination, i.e. specular lighting, more narrow cones are required to be launched which impacts performance. By definition, cone-tracing converges to ray tracing if limit of cone angle goes to zero. Because of that we apply most appropriate amount of cones and angles to achieve real-time frame rates.

Initial bounce from the camera can be done with standard tracing which is centred from each pixel. To improve that, Crassin preferred using deferred rendering G-buffer which can be used to start rays from the visible surface points directly [1].

In practice cone tracing method will not have a contact point of reflection / refraction since sampling point covers a volume. Reflecting from a volume can be a complex task since it is not clear to simulate how the volume interacts with the incoming light source.

Illumination also requires light injection pass in order to sample voxels faster in run-time. Light injection pass determines lit voxels by using calculating lighting for each voxel. This pass is only required to be done when the light moves since most of the voxels are static.

Light injection is done using a data structure similar to the reflective shadow maps (RSM) [18]. However, it only holds world position data. Using this world position, SVO is traversed and found leaf voxel is lit using light direction and stored normal and color value. Calculated luminance is then stored on the voxel tree then distributed through the higher levels of the tree structure. Theni the entire SVO is illuminated which makes it ready for sampling.

CHAPTER 4

PROPOSED METHOD

4.1 Overview and Differences over Voxel Cone Tracing

Our proposed method differs from Crassin's Method in multiple ways. Firstly, our method is designed to be dynamic object friendly from ground up. Crassin's method for dynamic objects is not scalable for many complex dynamic objects[33]. His algorithm relies on dynamically voxelized objects in runtime. Our solution perceives voxelization as a modelling tool and assumes all models come with their voxel model in addition to the traditional triangle model.

Secondly, voxel grid(s) follows the camera which requires octree reconstruction every frame. Since the system built to support extreme amounts of dynamic objects, anchoring octree to the camera was the probable choice. In addition, camera anchored octree enabled other advantages since the system can support an extremely large scene with a small octree depth by only tracing the near vicinity of the camera. Also, required voxel models can be streamed and used when they are about to be included in sparse voxel octree (SVO) as needed.

Thirdly, octree structure is dense up to a certain point (level 6, 64^3 in our case), memory constraints are rather low up to a certain point and we wanted to utilize hardware linear interpolation on lower levels of the tree. In addition to that, it simplifies SVO reconstruction since only the levels after the dense portion required to be constructed every frame.

Lastly, our voxel octree representation is quite different. In order to achieve real-time frame rates, our octree structure does not support fast hardware interpolation on its sparse nodes. The reason for that is during construction, entire voxel transform pool will be injected 8 times(one for each potential neighbouring corner) for interpolation to be consistent between nodes. this makes SVO construction rather slow. Interpolation will be employed differently on the cone tracing portion of the algorithm.

Cone-tracing algorithm is also different since our structure does not support fast linear interpolatable layout. In order to simulate hemispherical integral, multiple cones are needed to be sampled from the SVO and each cone will require 8 SVO fetches to interpolate linearly. This has heavy memory bandwidth requirements and not real-time applicable on current hardware. To achieve visually smooth results, we still relied on linear interpolation. In order to simulate faster, more

efficient sampling solutions are introduced. However accuracy of the trace is reduced since the sample count for each surface patch is reduced. In addition to interpolating in the level for smoother results, interpolation between levels should also be considered. To simulate interpolation between levels our approach uses previously sampled result to simulate the interpolation of between the depths of the tree.

4.2 Voxelization

Before runtime, we need to generate voxel representation of each object. Voxelization process is done by rendering each object into a series of 3D textures with a given voxel span. Voxel span is the width, height and depth of the voxel which will be used to sample the object. Because of the structure limitations, voxel representation cannot exceed 1024^3 (see fig. 4.2). Object's axis aligned bounding box (AABB) is used to determine if the object can be covered with at most 1024^3 dimensioned grid and the given span. If model's axis-aligned bounding box cannot fit into the voxel grid coverage; which is determined by span and voxel grid dimensions, that model is skipped for that span. However, this does not mean that this object will not contribute to the voxelized scene. Since our system utilizes multiple cascades, skipping is done only for that level of the cascade and potentially that object can appear on other cascades. If the object is large enough to be skipped by all of the voxel caches, that object will not contribute to the indirect illumination of the scene.

Our voxelization scheme is similar to the algorithm discussed by Crassin [51]. We choose to utilize surface voxelization algorithm to further reduce the amount of voxels on the scene since most of our algorithm is bound to the voxel size. However, surface voxelization creates sampling artefacts if cone tracing samples are distant from each other. The reason for that is, it may skip the thin surface layer of the object on the lover levels of the octree.

We purposely did not choose the sparse solid voxelzation technique discussed by Schwarz et al. [52], since our voxel transformation process cannot necessarily guarantee model-voxel consistency which may lead to voxel overlapping between bigger inner voxels and smaller outer voxels. However, combining more advanced transformation method which will be proposed as a future work this approach can be more applicable.

In voxelization process, all triangles are rasterized and fragments are generated as if we are rendering the triangle to the screen (Figure 4.1). Before rasterization however, triangle dominant axis is determined in order to prevent voxelization gaps in the triangle. Dominant axis is determined by looking up the vertex normals of the triangle and choosing direction that has the highest pixel coverage of that triangle. This is done in the geometry shader, since in geometry shader hardware exposes all the vertex values from a single invocation. After dominant axis selection, triangle is rotated towards that axis to increase pixel coverage of that triangle.

For edge cases, rasterization may still introduce voxel gaps on the triangle. This

can be prevented by using conservative rasterization [53] as discussed in [1]. In our approach, we used multi-sample anti-aliasing (MSAA) rasterization discussed in here [54]. MSAA is generally used for anti-aliasing but in our case it provides the necessary samples to fill the potential gaps around neighbouring voxels.

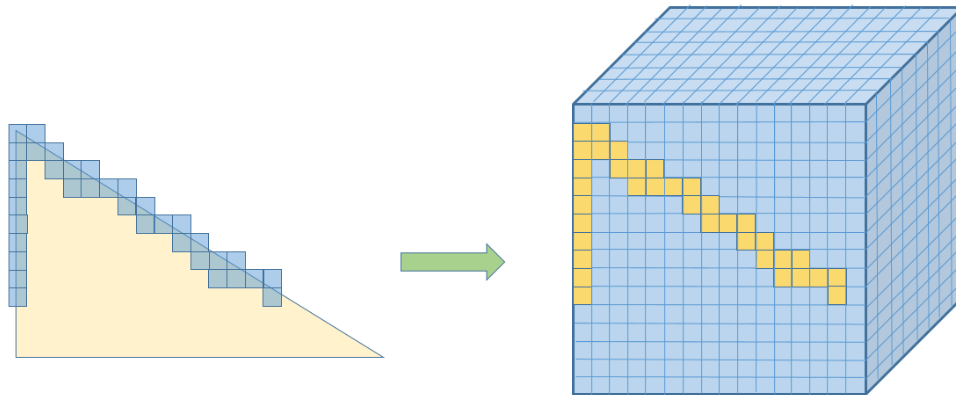

Figure 4.1: *Each triangle is rasterized as if it is being rendered in to a screen then from the fragment position the model space position of the pixel is determined. Then voxel position is determined.*

Our method requires two passes and we use accurate high precision data structures for voxelization. Since our voxelization method is done before runtime, limitation of the multi-pass and high memory bandwidth required voxelization costs were irrelevant. Voxelization occurs on a dense 3D texture which has a dimensions 256^3. We Sample the portion of the geometry on each pass in order to work for on small memory devices. Moreover, floating point precision of three dimensional memory requires high amount of memory and may not work on even high-end GPUs. Geometry voxelization for each node can require up to 64 passes ($1024^3/256^3 = 4^3 = 64$). However, these bottlenecks are trivial since all of the calculations are done before run-time.

Depending on the rasterization resolution, more than one pixel may sample each voxel. Our voxelization implementation atomically averages fragments generated by the shader in per voxel basis by adding each channel of color, normal and specular values and then dividing it when voxel packaging occurs. This requires higher byte requirements to prevent data loss because each voxel can be sampled by multiple fragments. We use three single precision floating point number for both normal and color values and a one floating point number for specular values.

After each sub-voxelization process, second pass on the three dimensional textures takes place. Each sampled voxel is packed and added to the voxel cache. Voxel cache structure holds multiple mesh's voxels in order provide batching on the CUDA kernel calls which will be explained in the later parts of the thesis. Packing converts floating point colors, normals and specular value into the 8 bit per-channel fixed point precision format.

4.3 Cascaded Voxel Grid

Now we will explain the run-time algorithm. Algorithm starts with pre-voxelized geometry explained on the previous part. Each mesh comes up with have its unique voxel representation for each cascade. We used struct of arrays approach in order to increase coalesced access on the GPU. We cache the pre-voxelized data on the GPU memory and transform from this data as needed when an object is included by the voxel grid system.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

X	Y	Z	Mip

NormalX	NormalY	NormalZ	Empty

ColorR	ColorG	ColorB	Specular

Weight0	Weight2	Weight3	Weight4
Weight Index0	Weight Index1	Weight Index2	Weight Index3

Figure 4.2: *Voxel cache data structure. Data is laid out as array of struct method thus each row is laid out on a different array in order to increase GPU memory access efficiency. Voxel weights and weight indices are only available for joint deformed objects.*

Last 8 bits of word that stores the normal is empty but when the voxel is transformed into world space position it holds density data of the voxel (Figure 2.2). In voxel cache, all voxels fully occlude by definition thus there is no reason to hold density information.

Our cache structure holds voxel position relative to the object axis-aligned bounding box (AABB). As stated above, according to the AABB size and bit limitations of the structure; which can only hold 1024^3 length grid, mesh may not have a voxel representation for that resolution. Because of that, last two bits of the word that holds the voxel indices is used to represent that whether this level of voxels are considered as a base level or not (Figure 4.2). This information is used to determine culling process handled if there is multiple cascades. Culling process is explained at the voxel transformation part algorithm explanation.

Our system may have multiple voxel grids that cover one level of the sparse voxel octree with reduced resolution. This idea is inspired from Kaplanyan et al.[31] approach. Each voxel structure implicitly positioned inside of the other cascade. Our implementation has 3 cascades with 512^3 nests. Even though voxel structure holds only 1024^3 voxels by bit limitations, each inner voxels' actual position can be derived by using voxel grid structure layout. Our implementation doubles the resolution while maintaining the cascade size. Figure4.3 shows our nested grid approach. Notice that each cascade relative position can be derived by function $x = (2^n - 1) * \frac{cascadeSize}{2^n}$ where n is the current cascade number, x is the starting location of the cascade.

Each voxel grid system is responsible for; including excluding voxels, transforming

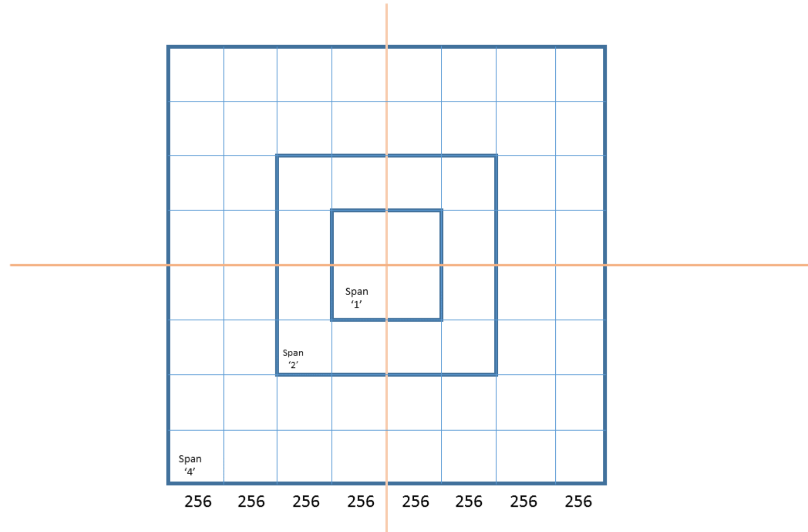

Figure 4.3: *Cascaded Voxel Grid Representation in 2D. Each cascade has same actual resolution* 512^2 *with different voxel spans. This enables virtual resolution of* 2048^2 *and only nearby objects use highest resolution.*

included voxels and culling voxels if the transformed voxel is out of bounds. Each cascade does these operations only for their own resolution of voxels. This operations are split into two parts, first part is about including and excluding voxels and second part is about transformation and culling of the voxels.

Before explaining those algorithms we need to explain how voxel system stores its data. Algorithm is designed to support as many dynamically input and output operations as possible. To achieve this requirement, memory is divided into two separate divisions called pages and segments (Figure 4.4). Segments are fixed size groups of voxels that are owned by the same object. Objects can have multiple segments and each segment will be full, unless it is the last segment. This segmented approach enables to reduce atomic operations per inclusion of the object.

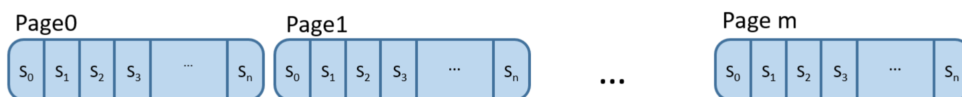

Figure 4.4: *Voxel grid page system uses memory system that removes or add pages. n is the segment count and m is the page count. Each segment is fix sized.*

The first iteration of the algorithm had single atomic add for each potential voxel that will be included in each frame that had a high performance cost. Current iteration of the algorithm has one atomic add operation per object segment, thus relieving atomic operation serialization on the GPU. Each object segment tries to find a location in the page system by starting from the first page and iterating through the last page. Each page has a stack of available segment locations and a stack index variable which points at the top of the stack. Each segment tries to atomically allocate location on the segment using the index variable. If it cant find a position, it moves to the next page. If all pages are full, all remaining

segments are dropped and memory allocator is signalled to issue a new page allocation.

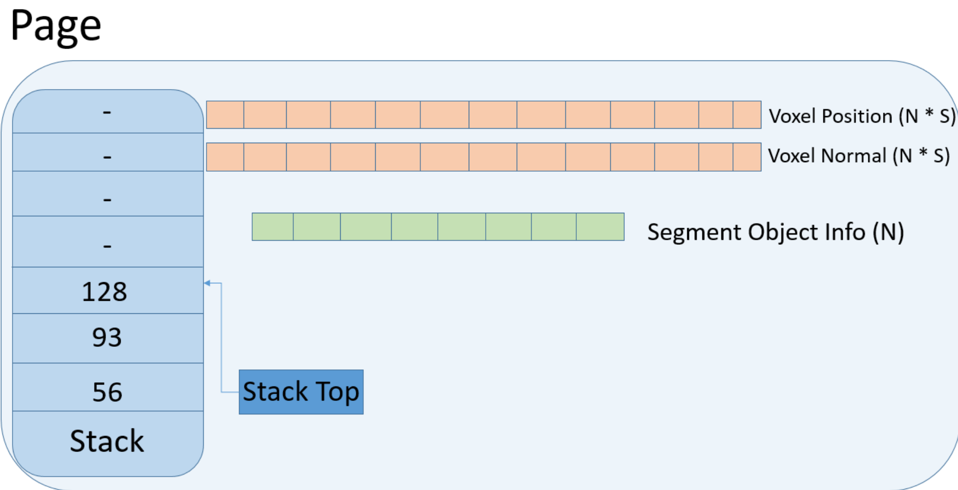

Figure 4.5: *Inside the page system, there is a stack which holds the empty locations, two arrays that hold normal and positions, and an array that holds object information of that segment. S is segment size, N is segment count per page.*

Portion of the algorithm that is responsible for voxel inclusion is only writes to or removes from the green array (see 4.5). Green array holds the information required to fetch and use the data from the voxel cache. Next algorithm, voxel include, will use this data structure to actually transform and cull objects from the voxel cache.

Before explaining the voxel include algorithm we will explain a implementation detail. In our implementation segment size chosen as 1024 and page size chosen as 65536. Each page can hold 64 segments. Because of that we choose our stack data structure to hold a single byte. Our implementation uses CUDA and we choose block size as 512 threads. The reason behind this number is that firstly it supports full occupancy and secondly this guarantees that each thread block in the streaming multiprocessor will be responsible for the single object. Thus this enables to fetch and store transformations required for that object to the shared memory without an occupancy penalty. This significantly improves performance and shared memory usage for the object that is required to have multiple transformations like bone transformed objects.

Voxel transformation portion of the algorithm is responsible for transforming objects that are included by the input/output portion of the algorithm. Currently, three types of objects are supported by the system; static objects, non-deformed dynamic objects and deformed objects with joint transformations. At each frame, transformation system re-transforms the entire page system according to object transformation and new grid position because for interactive applications such as games, camera movement occur frequently.

In detail, for static and non-deformed dynamic objects each thread block loads two matrices, one is for transformation and the other is for rotation. Rotation matrix is used by normals and transformation is used for voxel positions. Then,

all threads in the blocks load a voxel from the cache and include them to the system. For deformed objects with joint transformations, entire pool of voxels ask for their required final joint transformations. After that, threads load required joint transforms for this thread block. Then transformation occurs and voxels are now considered in world space.

After world space positions of the voxels are determined, their positions at the voxel grid will be determined. After that, collision checks takes place. If voxels collide with a inner grid, algorithm checks whether there is any lower level representation of this voxel. The reason behind is that if there is a higher resolution alternative managed by the other voxel cache, we omit inclusion of the voxel since lower level voxels will be averaged up to the higher level voxel anyway on the SVO average stage. This is where the last two bits of the first word is used. (see Fig. 4.2).

Now all required voxels by the SVO are on the page system and ready for reconstruction by the SVO sub-algorithm. This concludes input output and transform portions of the algorithm.

4.4 Sparse Voxel Octree (SVO)

Before explaining the octree reconstruction algorithm, first we will explain the data structure behind the octree. First of all, octree will be used both in the OpenGL section of the program and CUDA section of the program. CUDA section is responsible for updating and reconstructing the SVO and the OpenGL section is responsible for using the constructed SVO to cone trace and get results. Because of that, we could not use pointer system for SVO construction since OpenGL does not support pointers. To make the algorithm work on both CUDA and OpenGL, we used indexing relative to each level of the SVO. Each level preallocated by determining peak voxel counts for each scene. However, instead of relying on pre-allocation, algorithm can be adjusted to allocate its own memory in run-time by starting with an initial size then aggressively allocating memory when algorithm runs out of memory.

In our approach, we used two OpenGL buffers which are mapped to the CUDA context. From these two arrays, we allocate all sparse levels of the tree. For each level, arrays are divided into chunks and offsets of these are stored on a different buffer. Each level allocates relative to this position, thus, this further increases potential maximum node per level. OpenGL and CUDA also shares two 3D Textures, one is a mippmaped texture that holds material informations and the other is index texture that is used as a starting point of the SVO sparse portion.

One of the arrays is used for pointer mapping and other is used for holding required data for illumination and occlusion. Data structure is laid out to be as compact as possible and it is required to fit CUDA 64-bit atomic operations to get atomically averaged efficiently. After transformations, voxels may overlap on the grid(i.e. object overlapping, voxel model grid - voxel world grid mismatch). In order to resolve overlap, average operation needs to be conducted. Most ef-

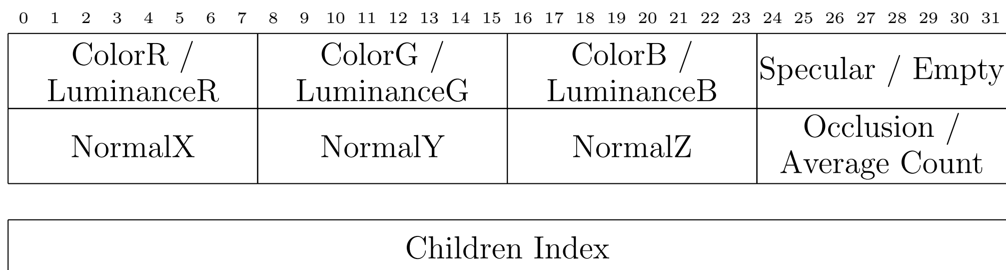

Figure 4.6: *SVO data structure. Color portion has different values depending on the light injection pass has happened or not.*

ficient and lock free way to implement this was by using atomic compare and swap operation; which is available on CUDA, on the entire material data set. If we had data structure that has values which uses more than a 64-bit word, we could have changed the algorithm to use a compare and swap atomic busy lock and average values that way. However, using compare and swap operation as a bush lock mechanism on a streaming multiprocessor will have heavy performance limitations.

Color and specular portions hold unilluminated data up until light injection occurs. After light injection, illuminated color values will be available on the material node instead of unilluminated data. Constant ambient light; which illuminates the scene, will also be applied at this stage. After illumination calculation, specular portion is irrelevant and will not be used. Implementation uses Phong illumination method because of that, we only require single value for object specularity, more sophisticated illumination methods may require more data and structure will be designed to incorporate that extra memory cost [6], [7].

Normal and occlusion portions hold world space normals which are used in the light injection. After injection normals are used to determine luminance direction when a cone samples the voxel. For leaf nodes, occlusion portion is used by atomic average operation as voxel count. Since leaf node voxels implicitly have full occlusion, there is no need to hold occlusion for the leaf nodes.

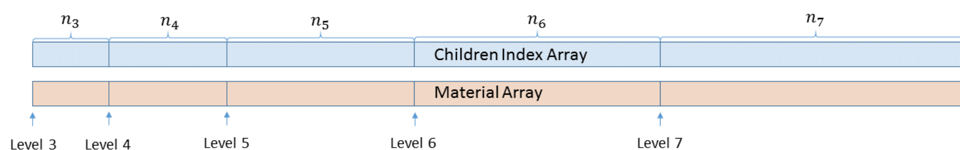

Figure 4.7: *2 dense, 5 sparse level, SVO Structure sparse portion array view. Two OpenGL buffers are used and each level has different pre-determined offsets. Dense portion uses single 3D mipmapped texture. One array holds index hierarchy, other holds material.*

Each pointer node holds single 32 bit integer, which is guaranteed to point all of the eight children. Each child may be empty, but algorithm requires to allocate all eight of the children, since each node individually allocates and does not have any information regarding how many children of this node are required to be allocated. This increases the memory requirements of the algorithm, but

performance increase gained by using this structure, which enables full atomic reconstruction, is worth the extra memory cost.

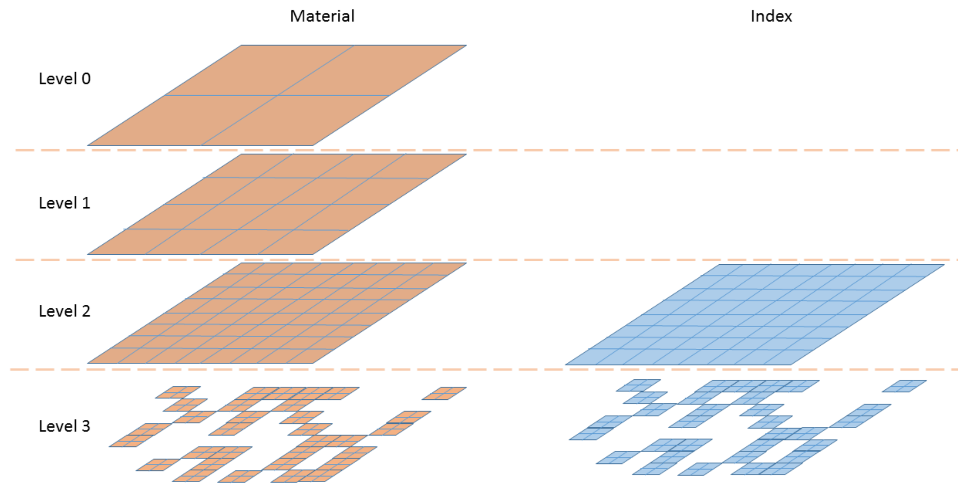

Figure 4.8: *Simple four level and one cascade SVO (actually quadtree) 2D representation. First three levels are densely stored and last layer is stored sparsely. This figure aligns with figure 4.7. Notice that all of the allocations are 2 by 2 (in octree version 2 by 2 by 2). In addition, last dense level is required to hold the 'root' index hierarchy which is shown as blue grid. Orange grids are for material information.*

Our SVO structure is semi-sparse. That means up to a certain level tree is allocated densely, which is 64^3 in our implementation. Seventh level is sparsely allocated. In order to combine the sparse and dense portions of the algorithm sixth level has densely allocated index map. Other levels of the dense data do not require pointer map since all of their children are densely allocated.

Before averaging, not only the last level but each cascade level holds leaf nodes. Even on inner cascades, algorithm is required to consider potential case that parent node and its children are both holding material information.

Only dense levels can utilize hardware 3D interpolation since those values are stored in the texture memory. Sparse values are stored in linear memory and do not utilize texture memory. The reasoning behind choosing linear memory over texture memory for sparse portions of the data structure will be explained in the SVO reconstruction section.

4.5 SVO Reconstruction

In this section we will explain SVO construction portion of the algorithm. After page layout has been set by include/exclude and transformation operations, SVO reconstruction operation takes place.

Entire SVO can be constructed with the single kernel call on the GPU. This approach works well on Nvidia® Maxwell architecture devices. For other devices

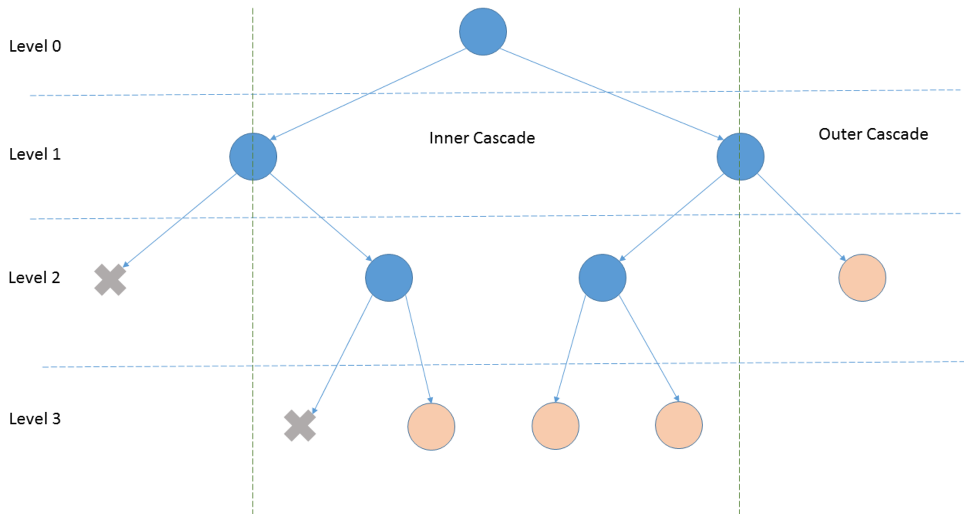

Figure 4.9: *Simple 4 layered, two cascaded SVO (actually binary tree) for one dimensional data before averaging. Orange nodes actually hold material information, blue ones are allocated and ready for averaging. X is for empty positions that are allocated but not used. Notice that last level has only data for only inner portion of the tree and outer layers of the tree do not allocate the last level.*

and architectures, level by level construction approach is issued. SVO reconstruction is required to be done in every frame since voxel grid(s) follows the camera. In addition to that, algorithm should scale well with dynamic objects.

Full atomic reconstruction algorithm relies on atomically allocating a node when it is required. On every frame SVO is reset, and each voxel that is transformed and stored in the voxel page system individually starts to allocate nodes in top down fashion. Starting from the first sparse level, atomic allocation occurs when a leaf voxel does not find its parent on a particular level. It atomically locks the node location required; which is always available since allocation always allocates the eight potential children, and allocates all of the children. And it continues to go down until it finds its position. Then it atomically averages its material values at that node.

Level by level method does not rely on atomic operations besides the atomic add operation. For each level, two sub operations occur. First each potential leaf node marks the required node on that level, then second pass looks up the the requested nodes and atomically allocates an index for that location on the children level. This algorithm requires to be called for each level individually with the entire voxel transform pool which is the bottleneck of the level by level approach implementation.

Both of those algorithms define the nodes at the center of the voxel location in order to make reconstruction algorithm fast. Like Crassin suggested [1], corner voxel definition can also be used since it makes the cone tracing with linear interpolation faster. However, we reconstruct octree every frame, our reconstruction scheme should also be faster. Corner voxel implementation increases work done for each voxel by eight(see Fig 4.10). Each node should also be incorporated to

the neighbouring nodes in order to interpolate consistently.

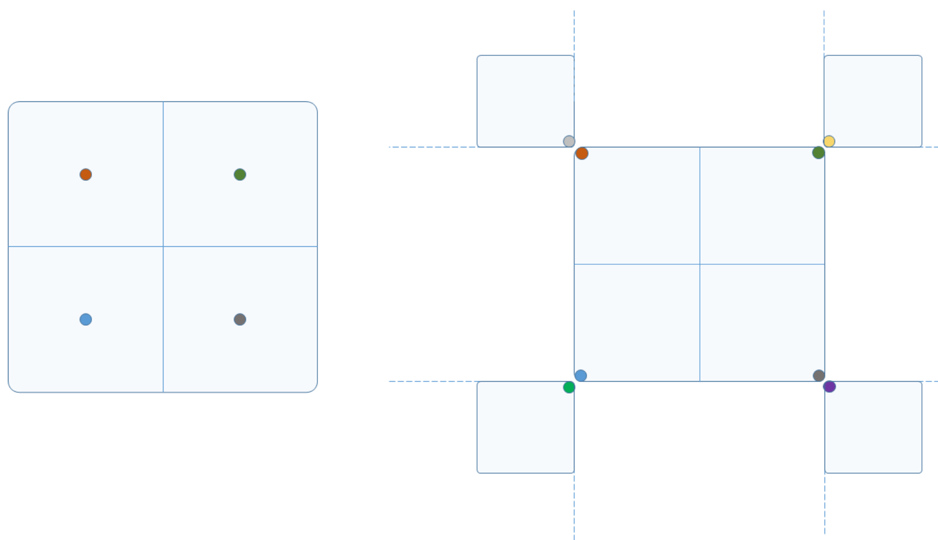

Figure 4.10: *Left diagram show the center stored voxels, right diagram shows the corner stored voxels. Corner stored voxels required to be averaged by all the neighbouring nodes in order to provide consistent interpolation.*

Our sparse portion of the data structure used by algorithm does not use texture memory. Implementation descision behind this is the modern hardware fetches global memory as fast as texture memory if it stays constant thorught the kernel call, meaning algorithm should only read from that global memory location [55]. This balances the global memory and texture memory usage because texture memory now can be fully utilized by 3D dense material and index textures and sparse portion of the algorithm utilizes global memory. Only difference is that global memory is required to do the interpolation by hand but since the memory fetches are as fast, code interpolation do not create a bottleneck in the algorithm. Another reason to use global memory operations is that, there is no 64-bit atomic compare and swap operation available on the GPU which prevents fast atomic average operation required by the down to top average stage which happens after SVO reconstruction.

After entire node system gets allocated, down to top averaging operation occurs in order to fully construct SVO for cone tracing. For each parent node, we call single CUDA thread which will average all potential 8 neighbouring values. Color and normal values will be averaged with the available children but occlusion value will be divided in to the maximum potential children count which is 8. In addition to that, leaf node occlusion memory location was used to hold the object count for averaging so that leaf node occlusion also get corrected.

4.6 Cone Trace on the SVO

Most computationally heavy portion of the entire method is the tree traversal. In order to get smooth results, tree sampling should be interpolated. Deterrences between single point sample and neighbour linear interpolation is shown on figure

4.11. However; our tree holds node centred voxels and for tri-linear sampling we require at most 8 memory fetches from tree in order to interpolate the sampled result properly.

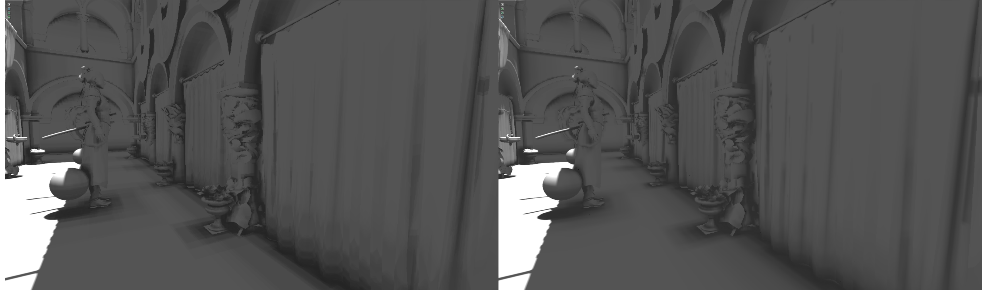

Figure 4.11: *Scene ambient occlusion with point sampling (left) and linear sampling (right).*

This sampling can be reduced to guaranteed single fetch if we store corner voxels. However, as stated above, it pressures SVO Reconstruction algorithm by forcing the algorithm to create consistencies between the adjacent neighbouring voxel groups. Hence reducing performance significantly.

Our tree sampling approach is similar to the [56]. We choose to implement the brute force approach by sampling all 8 neighbouring voxels. Worst case scenario, voxel fetch location can be on one of the corner edges of the voxel which forces 8 different nodes to be fetched. Middle edges of the voxel node will force 4 different nodes and center portion will require a single node. Application reduces the required samples by only sampling single cone at each traverse iteration. It provides similar visual quality but performance is higher but sampling artefacts can occur if sample distances are chosen too high. We choose this approach over corner node holding SVO structure since both operations will be done per frame and SVO reconstruction operation is much more complex because it relies on atomic operations.

Tracing is done using multiple cones from the surface patch. Our illumination method is simple Phong illumination which can be approximated using cones. In order to simulate Phong illumination, we will consider similar approach used by the Crassin [1]. Multiple high aperture cones for diffuse illumination and single low aperture cone for specular illumination are used to simulate indirect reflective lighting. This approach is illustrated in figure 4.13.

In our implementation we choose 3 cones for diffuse illumination which are centred around the surface normal. We sample 6 cones once per march iteration and this will create spiral pattern of sample points. This only covers the upper hemisphere of the rendering equation integrand but for diffuse illumination it is where the most of the irradiance is coming from because of the Lambert's cosine law. We omit lower hemispherical diffuse illumination in order to increase performance. For specular illumination we send a single cone which has a direction vector determined by Phong Illumination model. Specular cone is launched towards the reflection vector with an aperture determined by the specularity of the object. Metallic objects that reflect light will have a low aperture.

Sampling requirement for voxel cone tracing is quite high in our implementation.

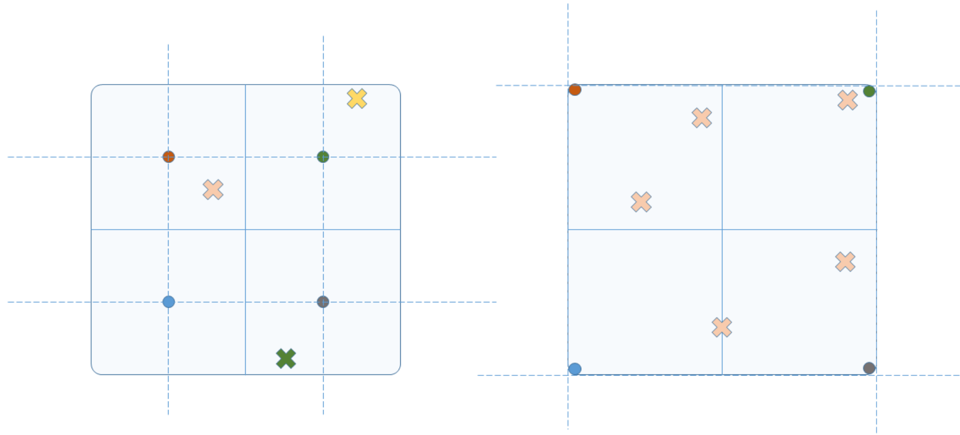

Figure 4.12: *2D representation of the voxel octree (actually quadtree). When SVO stores the nodes sampled, additional neighbouring nodes are required to be fetched in order to conduct linear interpolation. Sampling locations are marked with X sign. Orange sample location do not require additional node fetches. Green one is required to have one additional voxel fetch, yellow one requires three more voxel fetches. Right diagram show the sampling of the corner stored voxels. Notice that all of the fetches from this node do not require additional fetches.*

For each pixel on the screen we launch a total of two cones, each sampling the scene multiple times. For each sample, worst case, 8 SVO lookups are required which has a potential to traverse the entire tree depth. On worst case scenario with lower aperture cones, with a SVO depth of 11 (2048^3) total of 176 global memory operations are needed to be done for each sample per pixel. This introduces the main bottleneck about tracing method since global memory lookup on the GPU is slow, even more so without coalesced access.

4.7 Voxel Transformations & Applicability of the Common Animation Methods

In this section, we will present how transformation portion algorithm is used to derive certain animations for dynamic objects. These objects can have multiple joint transformation matrices or single transformation matrix. In both cases system handles the update operation over the pixel quite scalable. However, for each transformation method there will be approximations over per-frame voxelization which can be considered as a most accurate way of applying animations over voxel structures.

Static Objects

Most simplest form of object that can be found in a scene is the static object. Static objects do not move for their lifetime in the scene and they can be preprocessed by illumination algorithms most of the time. In contrast, in our ap-

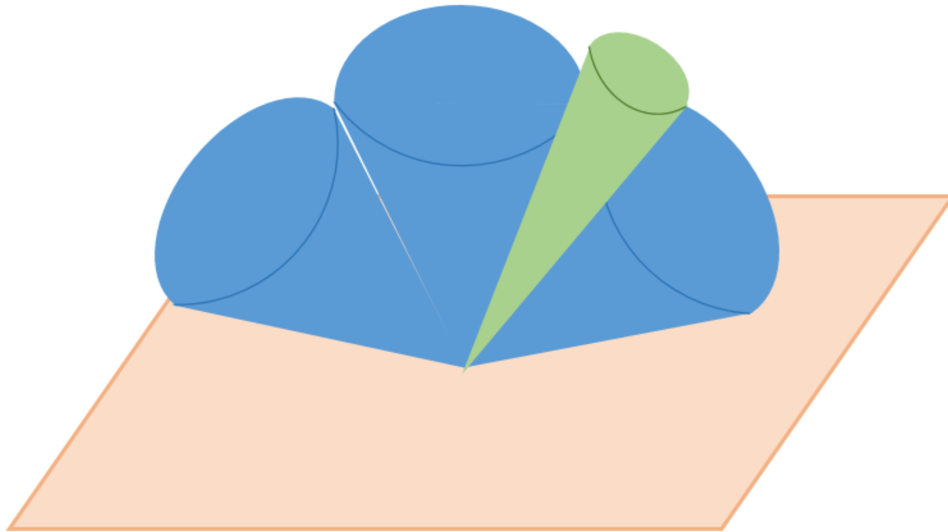

Figure 4.13: *Cone Tracing illustration on a surface patch represented by orange plane. Multiple diffuse(blue) and single specular (green) ray are launched in order to approximate Phong illumination.*

proach, static objects are still processed by the transformation stage. If model matrix is identity for that object; meaning that object is already in world space (world space and the model space are aligned), matrix load and multiplication can be skipped. However, voxels are required to be repositioned on the cascade since cascade moves as the camera moves.

Static objects can be fully approximated by our approach without introducing any defects.

Dynamic Objects

Dynamic objects reposition themselves every frame. Dynamic objects can be derived by the physics system or the animation system. These objects are hard to use as occluders because of their pixel coverage is hard to predict without actually rendering the object.

Our approach is to apply transformation matrix in per voxel basis for these objects. These objects assumed to be driven by a single transformation matrix. Hierarchical transformations are required to be reduced into a single matrix which is a trivial task. To increase performance and in order to approximate the scene efficiently, object may not be perfectly transformed into the voxel grid because of the model space and voxel grid world space mismatch.

Skeletally Animated Objects

Our voxelization method provides a simple solution for assigning weights for each voxel. Our skeletally deformed object has very small sized triangles; because of that we chose to use nearest vertex weights as a voxel weight. Because of the integer precision limitations we already have transformation inconsistencies on the voxel. Adding this inconsistency is hardly noticable for hight voxel size to triangle size ratios.

More accurate solution for this can be provided by applying the method of Dionne et al. [57]. However, this method requires solidly voxelized objects and our implementation use surface voxelized objects.

Figure 4.14: *Nyra character doing her jumping animation. First frame is the characters bind pose and there is no inconsistency of voxels. However after key-frame interpolation, transformation artefacts occur because of the method limitation.*

After assigning the voxel weights on voxelization, voxel transformation portion of the method uses the weights and the weight indices to load the required joint final transformation matrices provided by the animation system of the renderer. This data is available since parallel raster pipeline renders the skeletally animated object and provides the direct illumination of the object.

In detail, each voxel on the thread block requests final joint transformations required by itself. This request is stored in the block shared memory. Then block threads start to fetch requested matrices on to the block shared memory. After that, voxel transformation continues as explained above.

CUDA shared memory usage is a bottleneck if there is high amount of joint transformations available for the skeleton. In order not to reduce occupancy, shared memory allocation for each streaming multiprocessor needed to be adjusted. Nvidia® GTX 980Ti can easily supports 128 4 by 4 float and 3 by 3 pairs of matrices. 3 by 3 matrix holds rotation information and 4 by 4 matrix holds

complete transformation information of the joint, which is more than enough for most of the skeletally animated objects.

4.8 Memory Usage

Memory structure is designed to be as compact as possible. One of the most important memory constraint of the algorithm is that some voxel information stored on multiple stages redundantly. However, this redundancy is necessary since each stage treats the information with slight differences. Cache data holds model space voxel data. Page system holds world space position and normals. SVO system holds potential averaged world space normals and colors since multiple voxels may overlap after transformations. This slight differences make the algorithm to store same data multiple times.

For a sample case; each voxel is required to have at least 28 bytes distributed on the sub methods. If we consider skeletally deformed meshes; which holds at most 4 weight per vertex, amount of data required will increase by 8 bytes. In addition to that, extra allocations of the SVO is needed to be considered, since some of the data may not be used when we allocate each node with all of its children.

Still, we wanted to minimize the memory cost since this method is considered as a complimentary rendering illumination solution. Main rendering pipeline; which uses rasterization, is already using high amount of memory since it solves the most visually alluring portion of the rendering equation which is direct illumination. Our method mutually exists to support high fidelity direct illumination.

4.9 Method Limitations

Cascade Flickering and Unaligned Voxels

Since cascaded approach is inspired by cascaded shadow maps, some of the limitations of shadow mapping method can also be found in our solution. Camera anchored voxels are required to be translated differently because floating point world space to integer voxel space will introduce precision errors which create flickering. Shadow mapping solves this problem by moving the shadow map projection camera by the increments of the pixel coverage distance. This will prevent flickering but shadows in the scene will not be perfectly aligned. However, alignment problem is not noticeable unless pixel coverage size is high which is avoided since it introduces other problems like pixellated shadows etc.

However, SVO will have its intermediate nodes saturated with the data and those data are also prone to movement flickering. To prevent that, the movement required has to be changed with the voxel size of that level. For higher levels, voxel sizes will gradually increase and difference between the raster graphics and voxel system will even be larger.

Our implementation snaps the voxels in the system up to a certain level on the

tree, in order to provide balance between scene accuracy and flickering. Since the higher level of the octree will be used for very low-frequency illumination which has minimal effect on the scene and this makes the flickering hardly noticeable.

Scale and Shear Transformations

Another limitation of the method is that by definition, voxels do not support transformations which invalidates their definition. Voxels have to be uniform sized in order to be partitioned by a sparse voxel octree. Invalidating transformations include non-uniform scale and shear transformations. Uniform scaling is also problematic but can be used if and only if the transformed voxel size will align with the used cascade size after transformation.

For example; having model space voxel with a span of 1.2 world units on a voxel cascade that accepts 0.3 world units, it is not applicable to the method since cascades only accept voxel spans that have the same span as the voxel cascade. However, in that same example having a model space to world transformation had a scale factor of 0.25; which makes the span of the voxel same as the cascade space is, makes the resulting voxel acceptable.

If scale transformation is not used in animation, it can be pre-baked into the vertices at the modelling time in order to support scale and shear. If the object uses scaling animation, it is not possible to incorporate the animated object in to our solution.

Transformation Holes

Since all of the voxels are transformed individually, neighbouring voxels may end up in a same cascade node, because of the integer precision of the voxel index. Figure 4.15 illustrates the problem. Voxels have to be snapped to a certain grid after transformation and in our implementation we round the floating point world space position of the voxel to the nearest voxel cascade relative integer index.

Voxel hole problem is more noticeable on skeletally animated objects. Skeletally animated object voxels approximate their weight by choosing closest vertex weight as explained above. This approximation should introduce additional gaps on the objects (See fig. 4.14) in addition to the transformation artefacts created by the discrepancies between world space and voxel space transformation. Since the voxel model is used for indirect illumination, we leave the holes as is since voxel inconsistencies are hardly noticeable on the final image (Figure 4.16). However in order to use the cone tracing method for mirror-like reflections, transformation inconsistencies should be prevented.

Surface Voxelization

Since we use triangle rasterization to create voxels, our voxel models are empty inside. This introduces potential case where sampling can skip the surface voxel

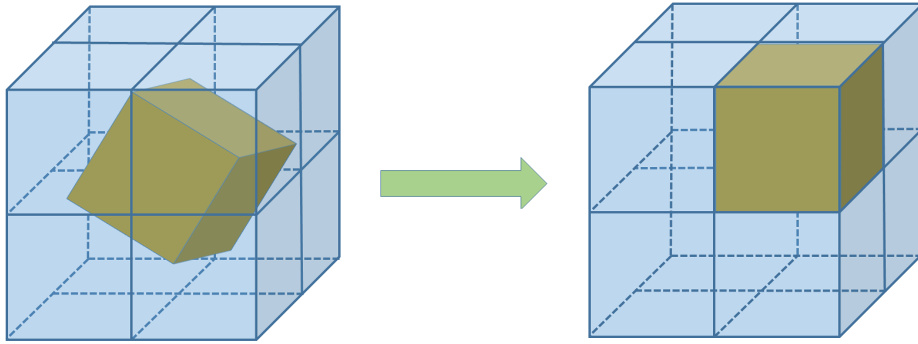

Figure 4.15: *After transformation (which includes a rotation in this case) voxels are needed to be aligned with the world space voxel grid which is always aligned with the world space axes. Nearest voxel position is chosen by the algorithm. This creates invalidates object rigidity and creates inconsistencies on the voxel system.*

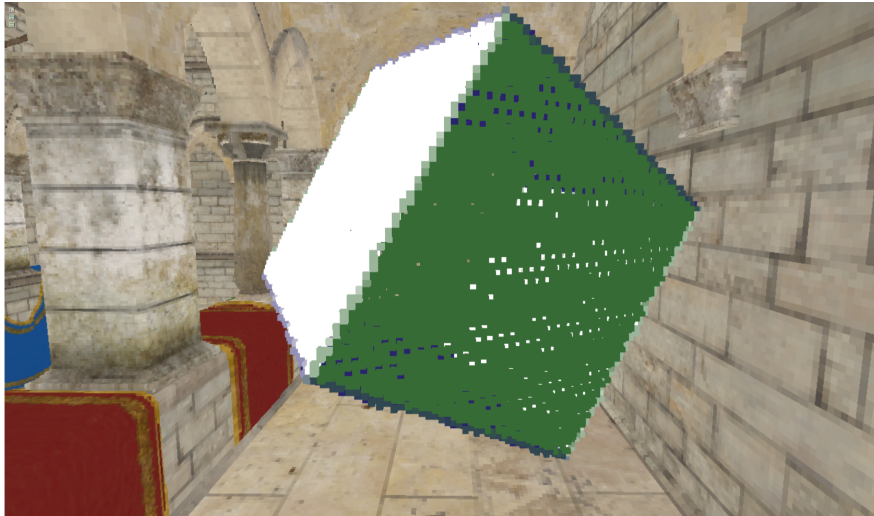

Figure 4.16: *Voxel holes can be seen on the rotating cube at the Sponza Atrium scene.*

by under sampling that location on lower cone apertures. In order to prevent the skipping, cones can oversample the tree but oversampling process will effect the performance. Other method that can be used to prevent this, voxelization scheme can be changed from surface voxelization to dense voxelization. However, this implies more voxels and having more voxels will impact the performance of SVO reconstruction and transformation.

CHAPTER 5

RESULTS

5.1 Scene Results

Our implementation used a simple deferred rendering implementation for direct illumination. Graphics Buffer (G-Buffer) stores world normals, world position in form of normalized device coordinate z (depth) information, unlit color (albedo) and surface specular value. Depth information is converted to world space floating point position by reversing the rasterization steps; first depth value and screen pixel coordinates are converted back into the normalized device coordinates (NDC) then from NDC it is multiplied by inverse of the projection and view matrices to get world space coordinates.

Our implementation use shadow mapping with mip-maps to simulate direct shadow, this implementation can be considered very simple rendering engine that is used by today's games. Our Deferred Rendering scaling with respect to screen resolution is shown in the figure 5.1.

All of the timings on this paper is done on the configuration provided below:

- **CPU** Intel Core i7-4790
- **RAM** 16 GB DDR3
- **OS** Windows 10
- **GPU** Nvidia® GTX 980Ti GPU (8GB GDDR5 RAM).

On all of our scenes, we used 1920x1080 resolution direct illumination. Since calculating direct illumination is not a bottleneck, we decided to use high resolution for increased visual fidelity. However, Our tracing resolution is lower since processing power requirement of the linearly interpolated cone tracing is quite high. We choose 1280x720 resolution for cone tracing buffer.

Beside from the pixel size, cone aperture and cone traverse maximum distance also directly effect the performance of the tracing. Higher cone apertures tends to be faster since they tend to fetch less frequently from the lower levels of the tree. In addition, maximum cone trace distance can be adjusted to increase performance.

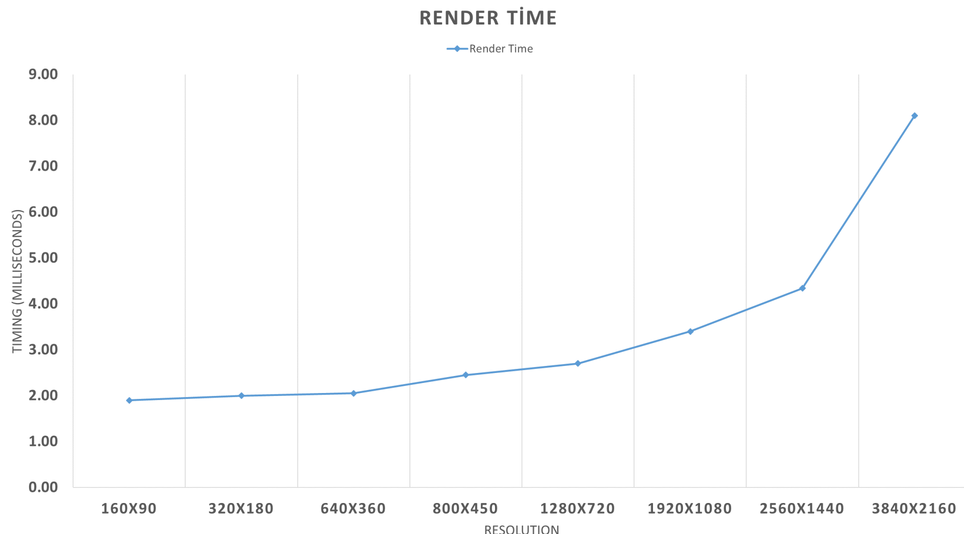

Figure 5.1: *Deferred rendering scaling on the GPU in the Crytek modified Sponza Atrium Scene. This timing also includes four cascaded shadow map generation for a singe directional light. As can be seen from the graph, up until certain resolution, pixel count is not enough to saturate the entire GPU.*

5.1.1 Sponza Atrium

One of our test scene is Sponza Atrium scene. We used modified version of the atrium which is done by Crytek®. It has 32 materials and 443 objects which are rendered by 460 draw calls. There are total of 292661 triangles on the scene. Single directional light illuminates the scene and there are multiple rotating objects and a single bone deformed mesh.

Atrium Sponza has single light entry point which makes it a good sample scene for showing indirect lighting. Scene has many small and large objects which are ideal for testing multi cascade voxel structures. Voxelization may not be able to fit some of the big objects into a voxel grid. However, algorithm is designed to utilize the objects without a base leaf node for that span.

Sponza Atrium has peak of 10 million voxels with a span of 0.3 world distance units with three 1024^3 sized voxel cascades. Voxel count gradually decreases higher span values and lower voxel cascade sizes. Visualization of the voxel coverages are shown in Figure 5.2. Our sparse voxel octree (SVO) construction times and voxel values for each coverage scheme can be seen in the Figure 5.3.

Coarse coverage sizes do not saturate the GPU, which can be seen on the graph by looking at $3, 4.8, 64^3$ and $3, 2.4, 128^3$ coverage schemes. Those took nearly the same amount of time which is an indication of the GPU under-saturation. I-O section of the algorithm tends to scale constantly since it is bound to the object segment count which barely increases for each coverage because in our implementation segment size is 1024 which makes the i-o portion of the algorithm; in theory, three order of magnitude faster than voxel count bounded portions of the algorithm.

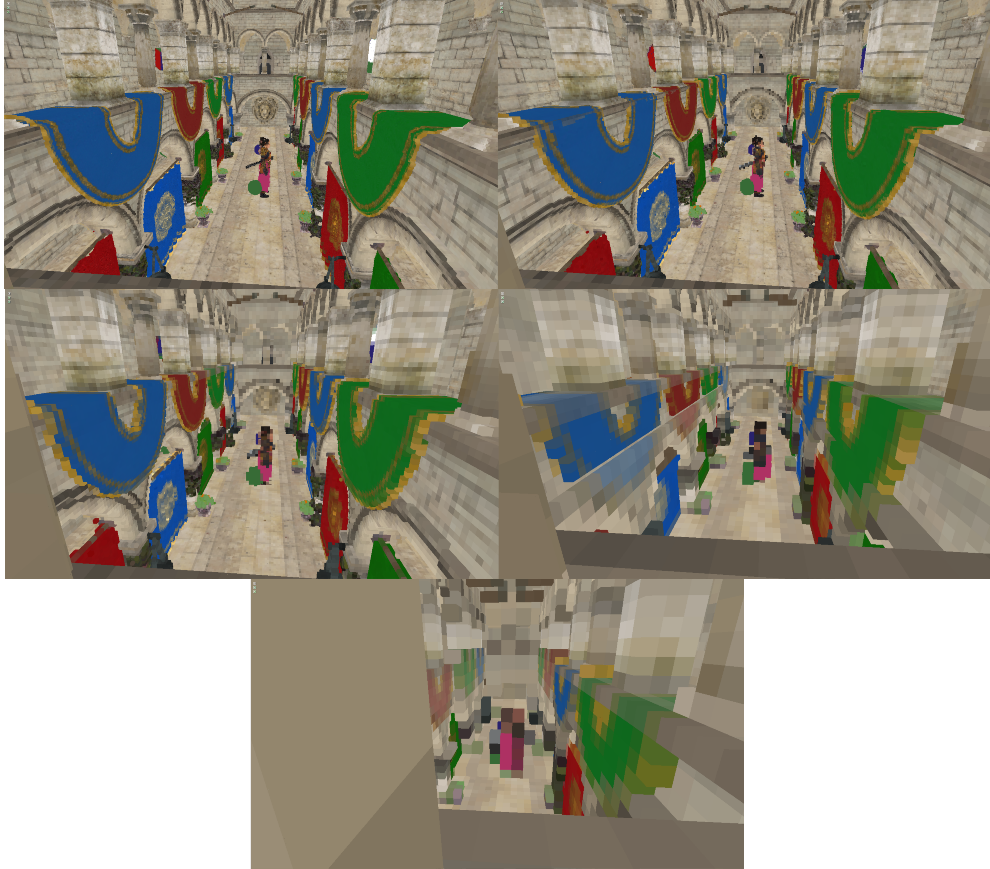

Figure 5.2: *Sponza Atrium Voxel Rendering for different coverage values.*

Table 5.1: *Sponza Atrium Timings. Timings of input, output, transformation, SVO reconstruction, and average stages of the algorithm is given. Trace portion is separated since it uses the constructed SVO structure and all other portions of the algorithm constructs the structure.*

Span	Cascade	I-O	Transform	Reconstruct	Average	Trace	Total
0.3	$3x1024^3$	1.5ms	5.01ms	9.86ms	3.18ms	35.12ms	54.22ms
0.6	$3x512^3$	0.85ms	3.42ms	6.34ms	1.30ms	28.7ms	40.61ms
1.2	$3x256^3$	0.47ms	1.23ms	1.60ms	0.39ms	20.60ms	24.29ms
2.4	$3x128^3$	0.40ms	0.51ms	0.50ms	0.16ms	15.0ms	16.57ms
4.8	$3x64^3$	0.35ms	0.46ms	0.25ms	0.05ms	14.30ms	15.41ms

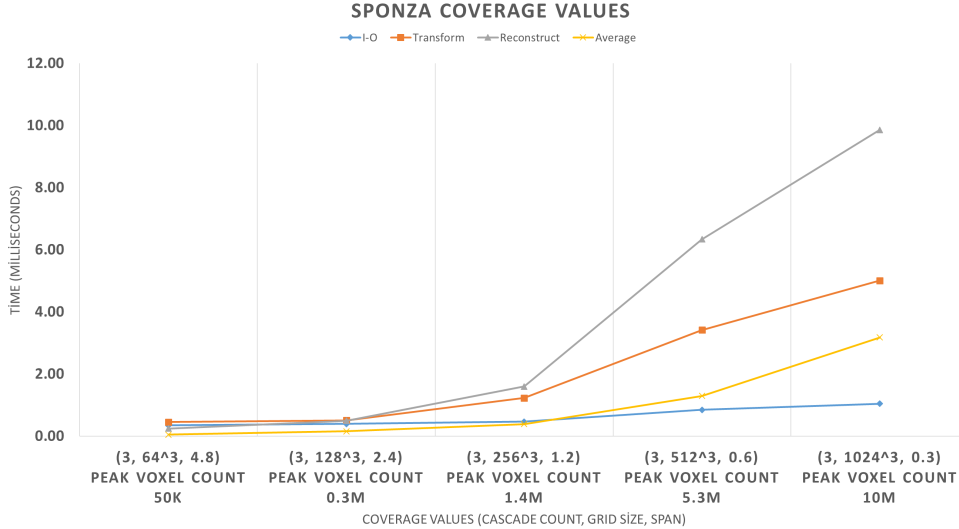

Figure 5.3: *Sponza Atrium Scaling. Only voxel tree construction portion of the algorithm is graphed since it is directly bounded by voxel count.*

In Table 5.1, we can see that tracing algorithm is the biggest bottleneck compared to other sub-parts of the algorithm. For higher resolution coverage schemes, algorithm results in interactive frame rates, for lower resolution coverage schemes, algorithm gives real-time frame rates. Scaling of the trace algorithm is bound to the screen resolution of the trace and cone aperture as well as voxel count. Therefore, algorithm parameters can be adjusted to reduce quality of the trace without reducing total scene voxel size.

Ambient occlusion results can be seen in Figure 5.4. Cone tracing method covers ambient occlusion quite well even though there is no direct lighting on the right side of the scene in Figure 5.4, shadows behind the curtains are approximated by the algorithm. Traditional shadow mapping methods can only cover direct lighting shadows and our method complements the direct shadows quite well. However, because of the voxel resolution limitations pixellated ambient occlusion can also be seen in the figure.

In Figure 5.5 our approximation of the real-time reflections can be seen. However, because of the performance limitations, mirror like objects like polished metals cannot be calculated in real-time frame rates. However, glossy surfaces; which have more blurry reflections, can be approximated quite well. Marble like floor on the Sponza Atrium is quite a good example for this phenomenon which can be seen in Figure 5.5. Our implementation is capable of capturing reflections from both static and animated dynamic objects.

However Figure 5.5 reflections can be approximated quite well with other methods [8], [44] since reflection contact point resides in the screen space. Both skirt of the Nyra character and the blue curtain can be given as examples for this situation. Although the screen space algorithms can give more realistic results, they cannot cover portions of the scene that is not inside the screen-space. Since our structure covers the entire scene, reflections can be approximated at the places that are not covered by the camera. In Figure 5.6 we can see the reflections of nearly the entire

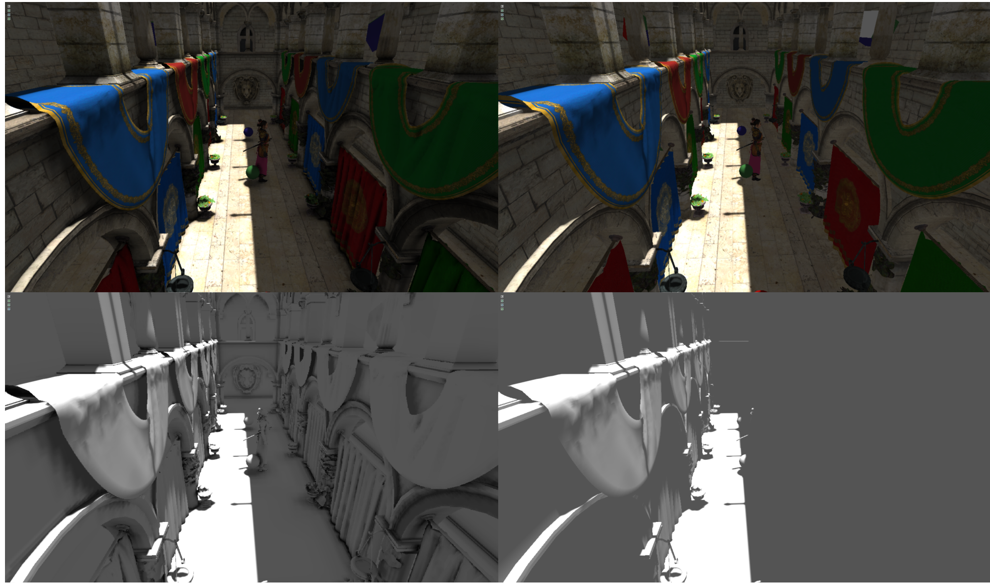

Figure 5.4: *Sponza Atrium Occlusion. At left column, ambient occlusion is enabled. Top row shows the final scene lighting image and bottom row shows only the light intensity texture.*

scene on the mirror-like white sphere. Our limitation of the mirror like objects can be seen here; in addition, voxel resolution artefacts can be seen more clearly in this image.

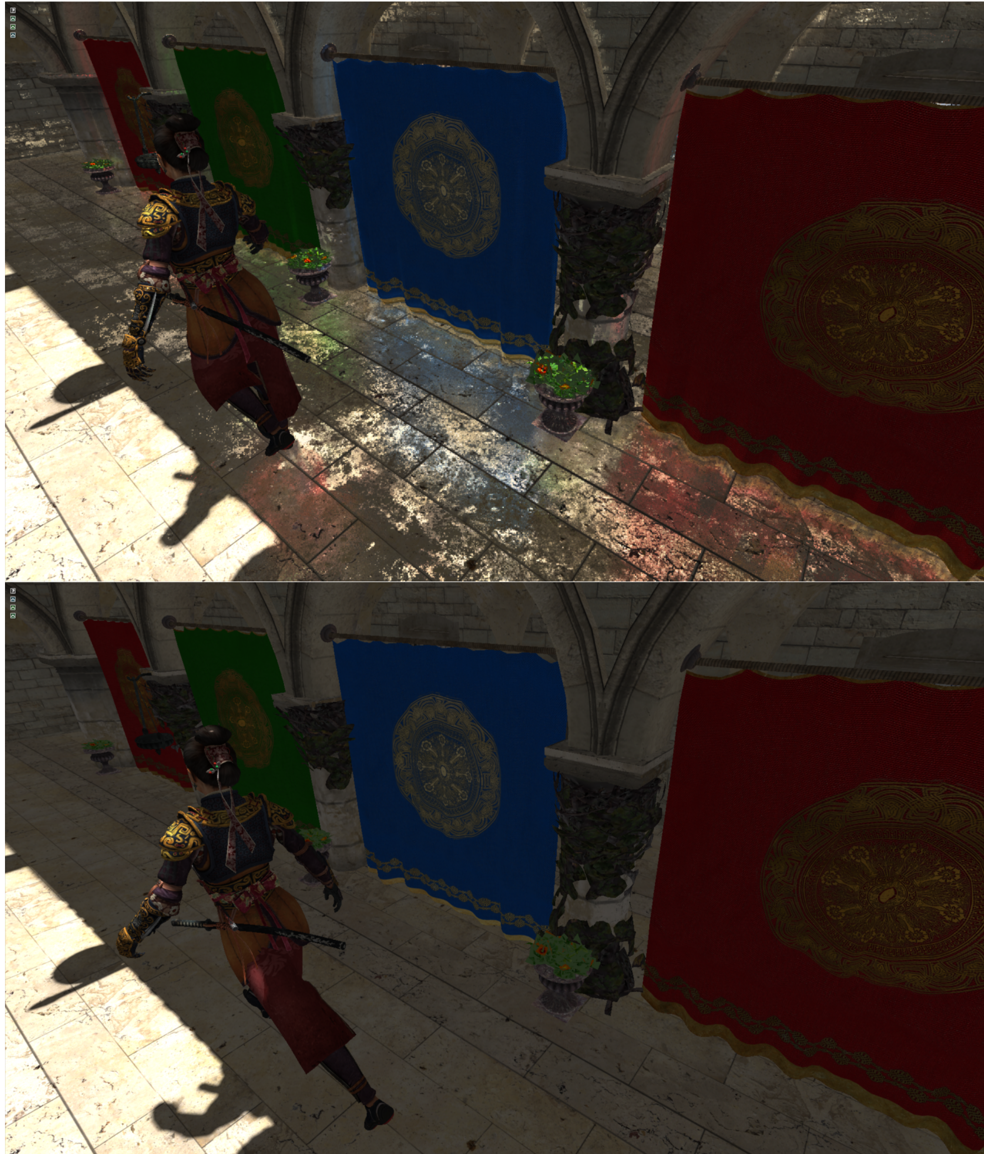

Figure 5.5: *Sponza Atrium Reflections. At top image reflections are on and at the bottom column reflections are off.*

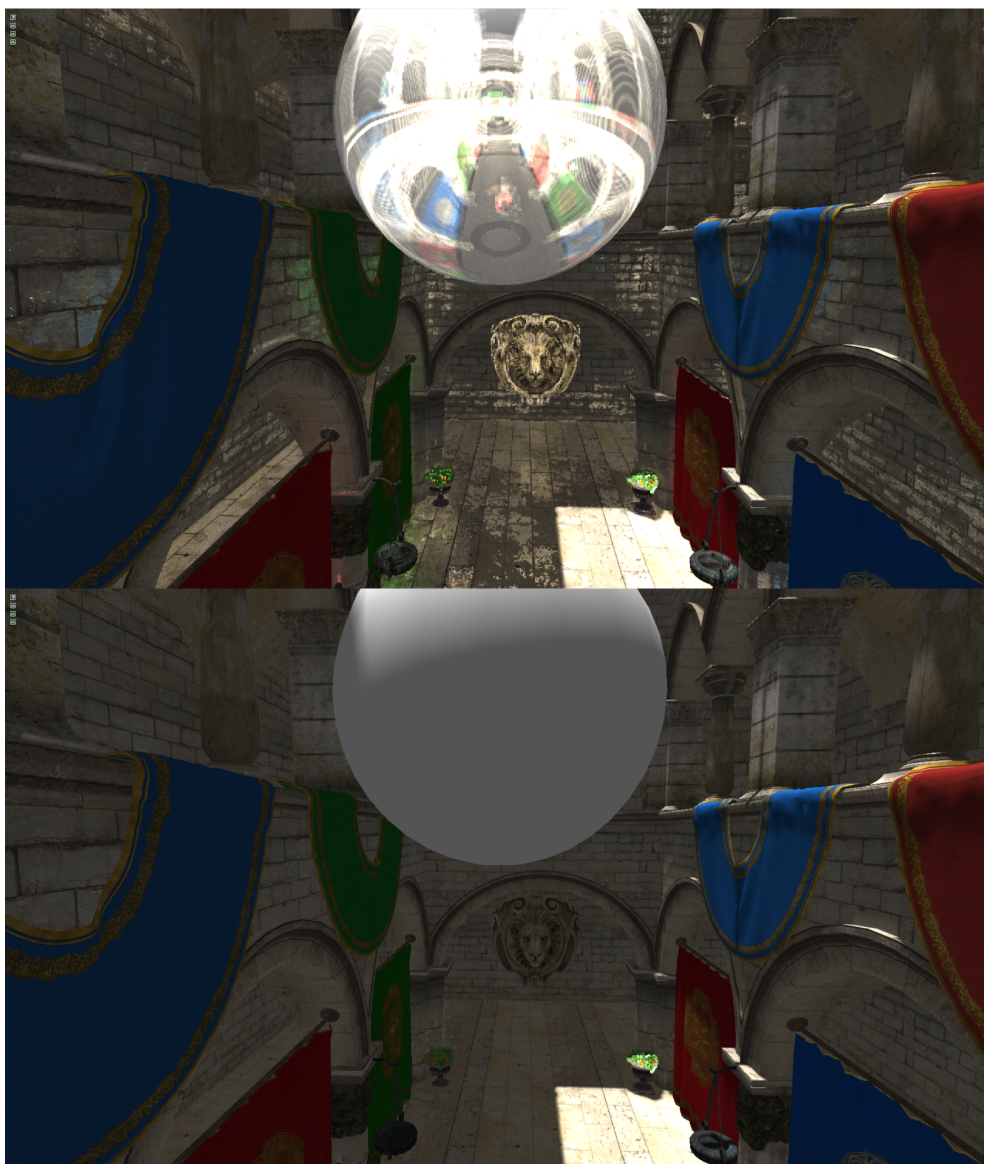

Figure 5.6: *Sponza Atrium Sphere Reflection. Full reflective white sphere reflecting light from entire scene. Notice the linear sampling artefacts since specular cone traced and sampled without linear interpolation.*

5.1.2 Sibernik Cathedral

Another scene we chose to deploy our algorithm on is the Sibernik Cathedral scene. Sibernik Cathedral has 11 materials and a single object that is rendered by 11 draw calls. Scene has total of 71341 triangles. Single directional light illuminates the entire scene. Sibernik cathedral has minimal light intake points which is applicable for approximating diffuse indirect lighting. Scene consists of single monolithic object in order to test our multi cascade input output operations on objects that is bigger than the voxel grid systems.

If Sibernik Cathedral covered with three 1024^3 sized voxel cascades where lowest level cascade having a span of 0.3 units, there will be total of 8 million voxels in update circulation. Voxel count gradually decreases with higher span values just like Sponza scene. Sibernik Cathedral coverage scheme images can be seen in Figure 5.7.

If we look at the voxel scaling (Figure 5.8), we can see that it follows the same structure as Sponza Scene. Only difference is that; although having 2 million less voxels, reconstruction portion of the algorithm takes more than two milliseconds longer to complete compared to sponza scene (Table 5.2). This shows that the scene layout also has impact on the SVO construction time. Sparsely laid out objects tend to take longer time since each object is required to allocate different parts of the tree which makes SVO structure to be allocated coarsely.

Ambient occlusion effects do not contribute to this scene as well as Sponza scene since this scene consists of flat surfaces with minimal obscurance (Figure 5.9). Only edges of the wall and pillars have ambient occlusion which is the expected result.

Another example of reflections can be seen in Figure 5.10. In this screen-shot, most of the cathedral is reflected from one of the brass colored windows in the scene. As can be seen on the screen-shot, limitation of the surface voxelized objects are quite apparent. Trace sample can pass through objects since single tin layer of voxels defines the object. However, this light sample pass-thought effect is only apparent on mirror-like objects.

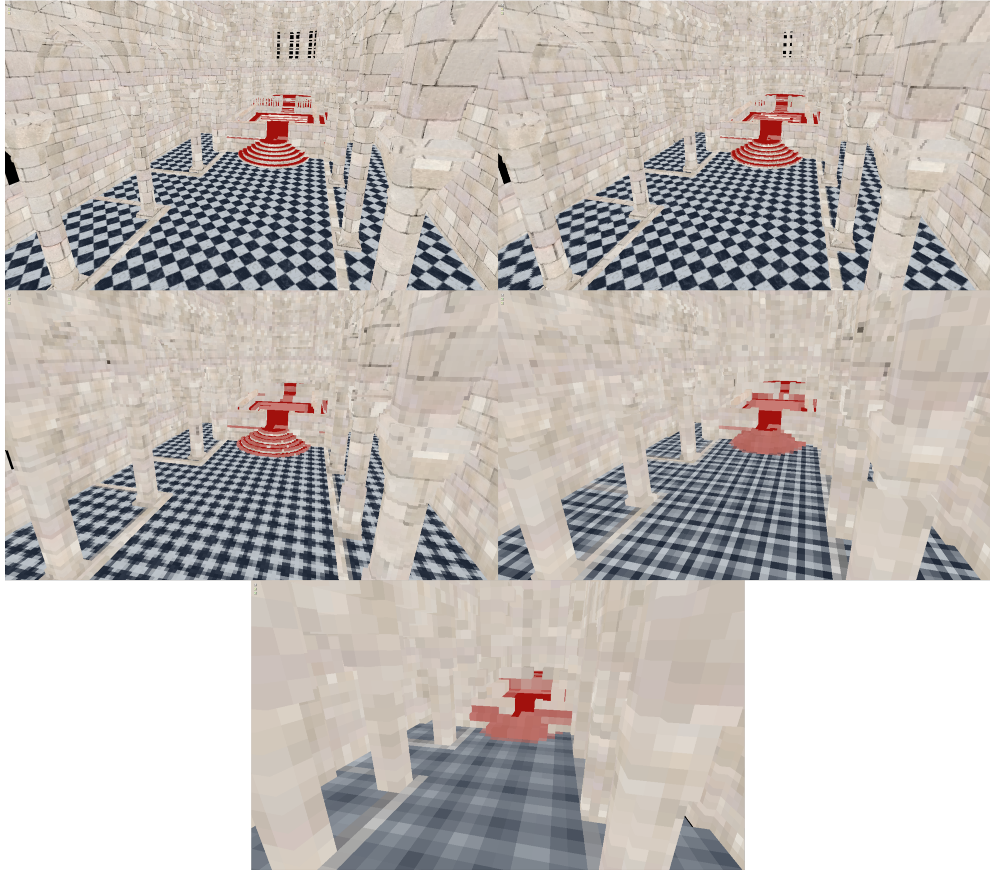

Figure 5.7: *Sibernik Cathedral Voxel Rendering for different coverage values.*

Table 5.2: *Sibernik Cathedral Timings. Timings of input, output, transformation, SVO reconstruction, and averaging stages of the algorithm is given. Trace portion is separated since it uses the constructed SVO structure and all other portions of the algorithm constructs the structure.*

Span	Cascade	I-O	Transform	Reconstruct	Average	Trace	Total
0.3	$3x1024^3$	0.13ms	6.6ms	12.3ms	3.1ms	45.3ms	68.79ms
0.6	$3x512^3$	0.16ms	2.5ms	3.96ms	0.89ms	36.48ms	44.47ms
1.2	$3x256^3$	0.24ms	0.85ms	1.01ms	0.24ms	29.3ms	31.64ms
2.4	$3x128^3$	0.64ms	0.38ms	0.31ms	0.09ms	23.27ms	24.21ms
4.8	$3x64^3$	1.49ms	0.2ms	0.13ms	0.05ms	12.3ms	12.81ms

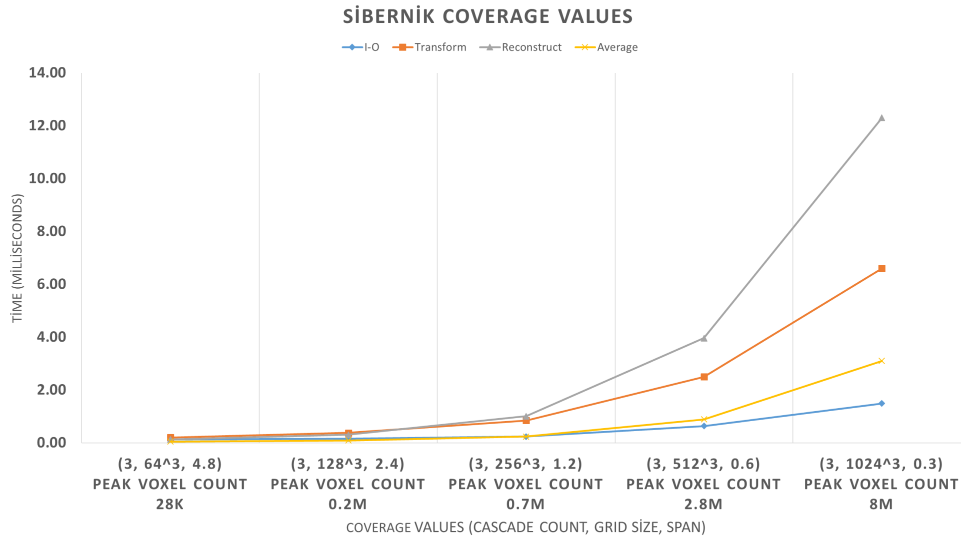

Figure 5.8: *Sibernik Cathedral Scaling. Only voxel tree construction portion of the algorithm is graphed since it is directly bounded by voxel count.*

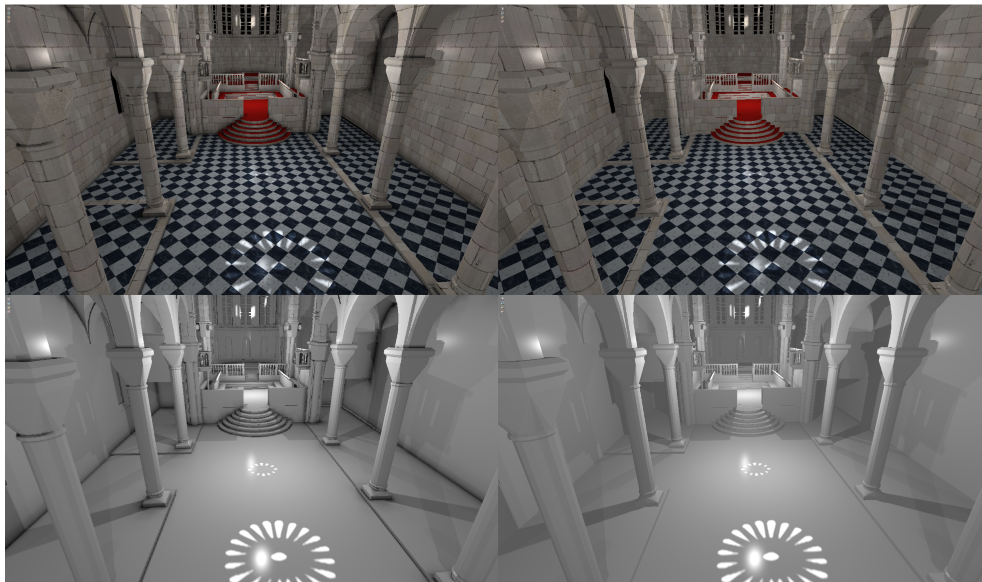

Figure 5.9: *Sibernik Cathedral Occlusion. At left column, ambient occlusion is enabled. Top row shows the final scene lighting image and bottom row shows only the light intensity texture.*

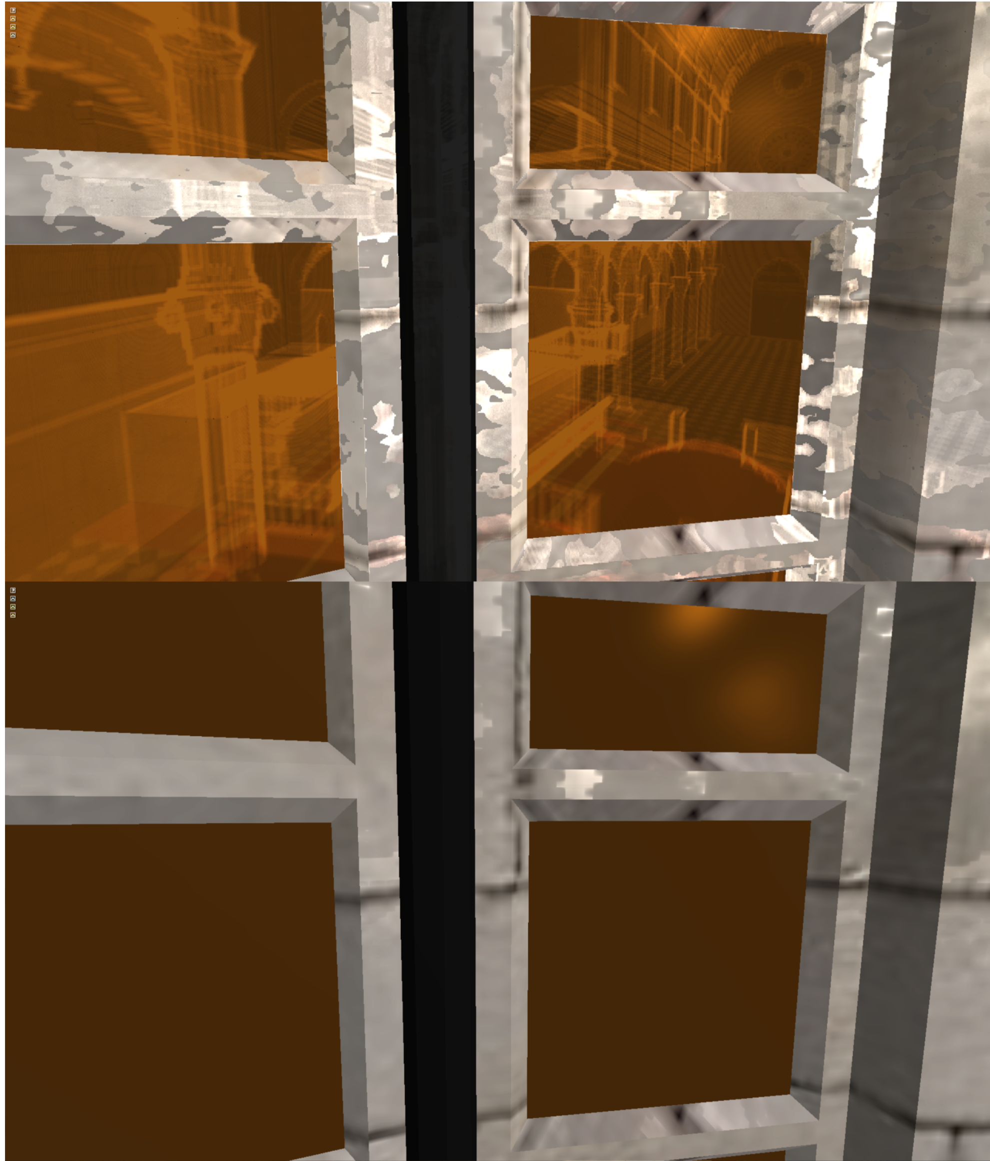

Figure 5.10: *Sibernik Cathedral Reflections. Entire scene reflected from brass coloured window.*

5.1.3 Cornell Box

Last scene we will choose to show our algorithm on is the custom Cornell Box scene with two rotating spheres each having a X shaped mark. Scene has 7 materials and 3 objects rendered by 7 draw calls. Total triangle count of this scene is 1612 triangles. Single directional light is shed towards the opening in an angle. Cornell Box is mainly used to cover light bleeding effect that is reflected towards the two coloured wall of the outer enclosure. Because of that, most of the objects have little specular material properties.

Highest coverage values used for Cornell Box were 3 cascades having 1024^3 sized grids with a span of 0.3 units. In this coverage setting, Cornell Box had 4.2 million voxels. Timings of the algorithm with respect to voxel size and voxel depth of the SVO are given in Figure 5.12. Coverage images of the different cascade values can be seen in Figure 5.11.

If we check the timings, we can see the same pattern compared to Sibernik Cathedral scene (Figure 5.3). Another issue about approximating scene with voxels is that even simple scenes like Cornell Box; which has 3 objects, can require millions of voxels in order to be approximated properly. Ambient occlusion phenomenon has minimal effect on this scene as well since there is no clustered objects that can create light obscurance. As can be seen in Figure 5.13, ambient occlusion at the corners of the outer enclose can be pointed out.

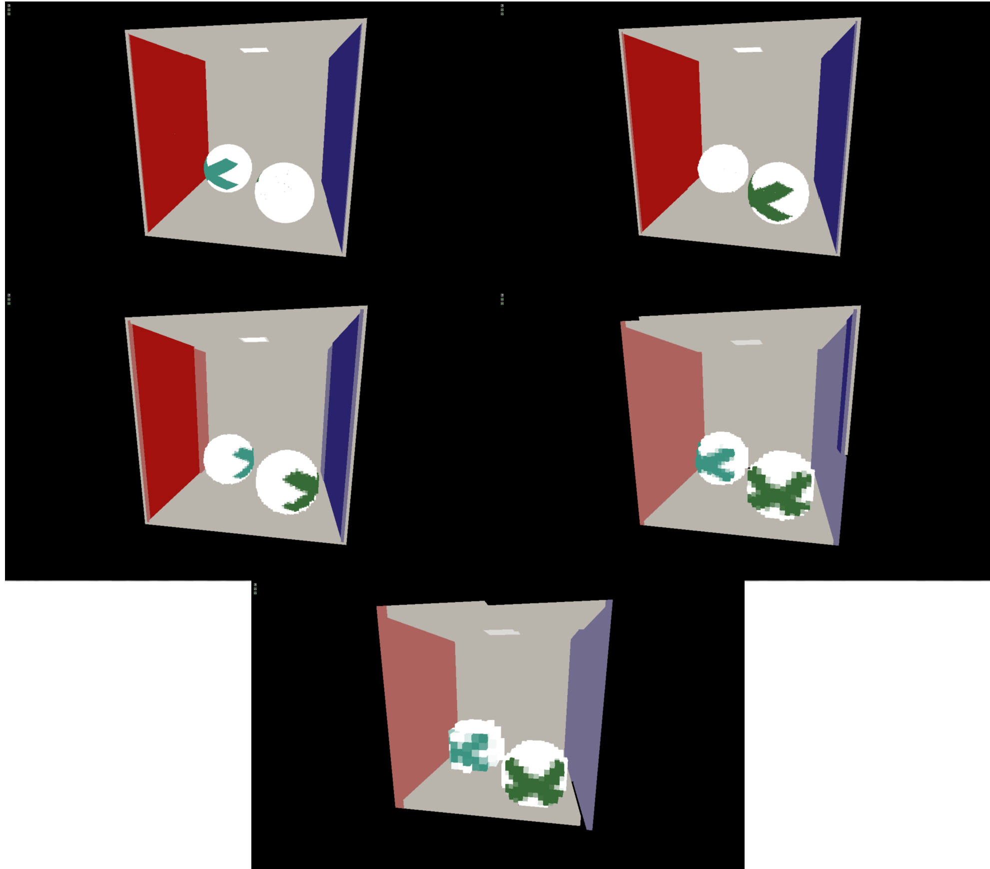

Figure 5.11: *Cornell Box Voxel Rendering for different coverage values.*

Table 5.3: *Cornell Box Timings. Timings of input, output, transformation, SVO reconstruction, and average stages of the algorithm is given. Trace portion separated since it uses the constructed SVO structure and all other portions of the algorithm constructs the structure.*

Span	Cascade	I-O	Transform	Reconstruct	Average	Trace	Total
0.3	$3x1024^3$	0.42ms	1.7ms	4.7ms	0.87ms	51.3ms	58.99ms
0.6	$3x512^3$	0.24ms	0.63ms	1.32ms	0.29ms	36.01ms	38.49ms
1.2	$3x256^3$	0.17ms	0.28ms	0.3ms	0.12ms	28.9ms	29.77ms
2.4	$3x128^3$	0.16ms	0.2ms	0.18ms	0.06ms	24.27ms	24.87ms
4.8	$3x64^3$	0.16ms	0.2ms	0.11ms	0.05ms	14.3ms	14.82ms

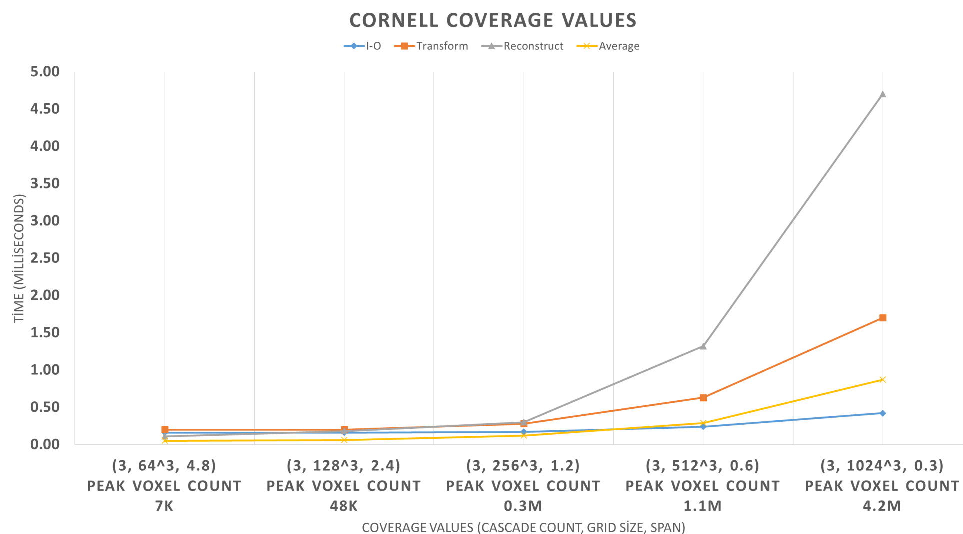

Figure 5.12: *Cornell Box Scaling. Only voxel tree construction portion of the algorithm is graphed since it is directly bounded by voxel count.*

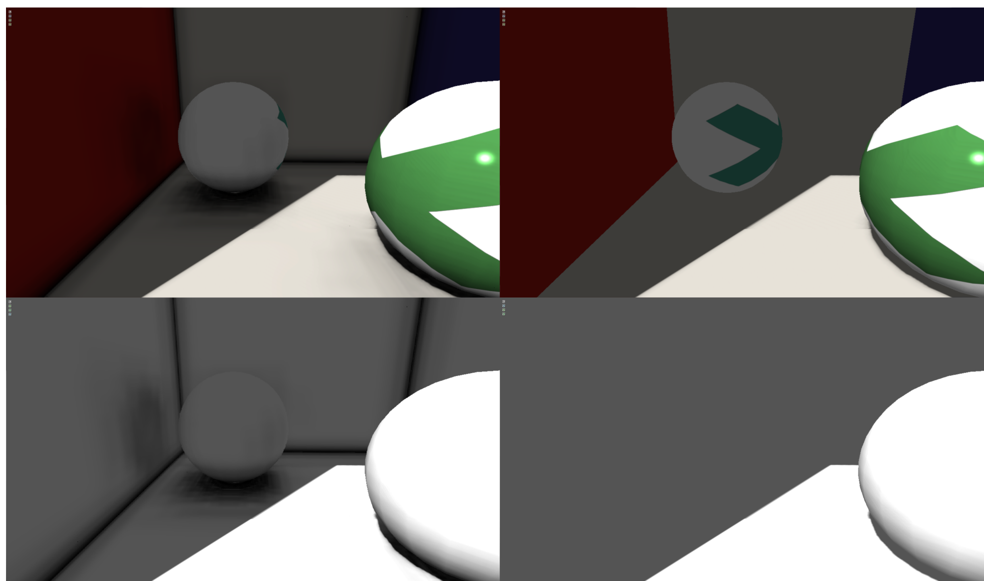

Figure 5.13: *Cornell Box Ambient Occlusion. At left column, ambient occlusion is enabled. Top row shows the final scene lighting image and bottom row shows only the light intensity texture.*

5.2 Dynamic Object Scalability

Since SVO construction is done per frame, entire system has a constant scaling for dynamically allocated objects. Joint transformation cost has the only difference between the static voxels and the joint transformed voxels. In Figure 5.14, scaling cost of multiple joint transformed objects are provided. As can be seen from the graph, additional cost from the joint transforms are minimal. Anyhow, joint transformed objects are used minimally on the real-time applicable scenes because their rendering and animation times are already consume high amount of the processing budget for each frame.

Multiple skeletal objects on this example are the same. However, all of those animated objects considered as a unique mesh and loaded and animated independently in order to actually determine the scaling factor of the joint transformations. Even though the scaling looks fine on the SVO level, the animation cost of the scene bottlenecked the frame since we do the animation calculations on the CPU. This limitation shows that, before having a SVO transformation bottleneck, CPU can choke the frame by not providing enough joint transforms to feed to the GPU. However, our implementation of the animation transformation did utilize a single core and the code did not optimized quite well. In order to conclude where the bottleneck will be for the scenes with many joint transformed objects.

5.3 Real-time Applicability

Our approach is interactive or real-time depending on the voxel coverage sizes. Depending on the resolution, cone tracing portion of the algorithm performance can be increased. However, utilizing these kind of approaches will result in poor visualization. Even on more applicable performance / quality ratios, algorithm is not real-time ready. You can still under sample the screen and apply Gaussian filter to achieve smooth and high quality results with better performances. However, this approaches can be applied if and only if you consider achieving low-frequency indirect illuminations.

Gaussian filter is already well established technique in real-time graphics in order to cover approximation artefacts and to achieve real-time frames. Gaussian filter is a useful tool to get rid of inconsistency artefacts on high-frequency results on the method but we did not choose to implement it since it is better to show limitations of the method than covering it with established methods such as Gaussian filter.

We can consider one degree aperture cone as a ray in order to compare the speed-up gained by a standard ray tracing method using SVO as a acceleration structure. Still apertures should be lower for specular reflections in order to capture reflections from the scene. However, diffuse indirect illumination (radiosity) is a low-frequency illumination higher cone apertures can be sufficient. Since lower aperture cones fetch from lower levels of the tree computation cost is higher (Figure 5.15).

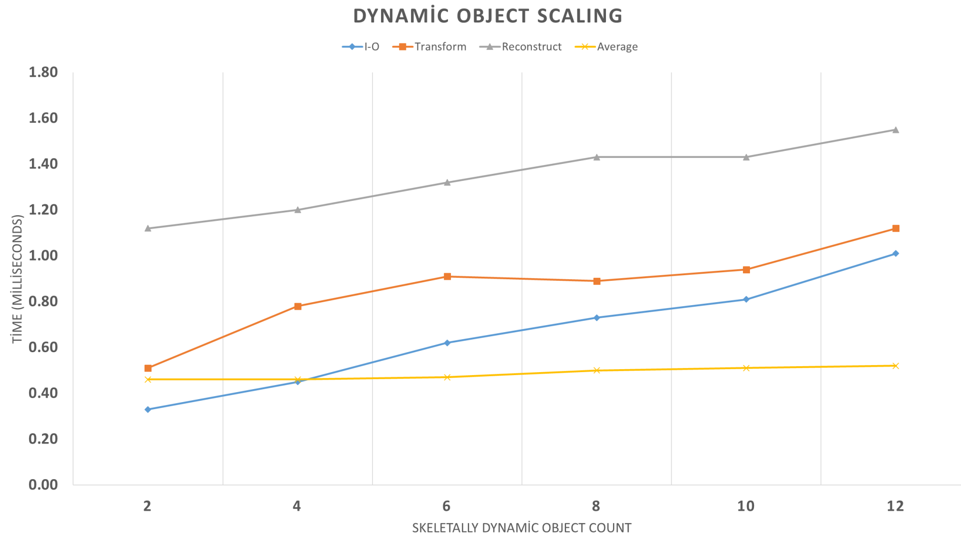

Figure 5.14: *Joint animated object per scene. Because of the memory limitations of our implementation, at most 10 skeletally dynamic objects could be calculated.*

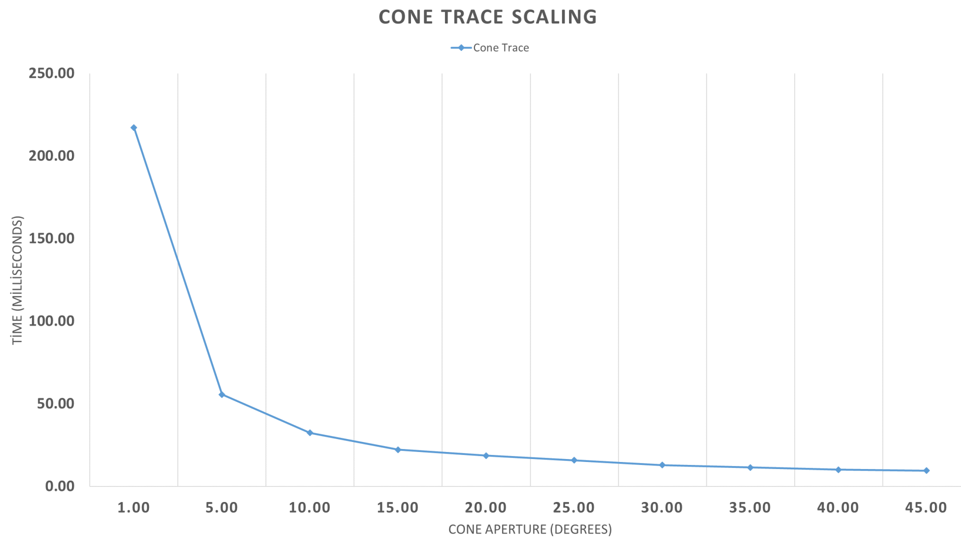

Figure 5.15: *Cone tracing timings with respect to cone aperture changes. One degree is given to approximate the timing of the ray tracing to give baseline to the scaling of the cone tracing method. At most 45 degree aperture is given since after 45 degrees opening cones will start to overestimate hemispherical integral.*

CHAPTER 6

CONCLUSIONS

6.1 Conclusion

In this thesis, we presented a novel approach to construct sparse voxel octree (SVO) in order for it to scale with high amounts of dynamic objects. In our approach, instead of voxelizing object every frame after animating, we consider voxels as a object representation and transform those voxels into world space every frame. After transforming voxels, we constructed SVO every frame in real-time speeds. After construction, SVO structure used to approximate rendering equation by sending series of cone shaped rays which is proposed by Crassin [1]. Cone rays, which are used to approximate the integration of positive hemisphere of a surface, are quite few compared to traditional rays. Therefore cone tracing technique achieves real-time computation times. Our approach supports static objects, rigid dynamic objects with transforms and dynamic objects with joint hierarchies. For all of these approaches, algorithm promises scalable results.

Our approach used to calculate ambient occlusion, indirect global illumination and reflections. Results are discussed on the scenes which are Sponza Atrium, Cornell Box and Sibernik Cathedral. Our voxelization technique scaled quite well on latest generation hardware and can support millions of voxels and constructed SVO in real-time.

Even though our approach can be used for real-time, other modifications are required to be done in order for it to be both visually appealing and computationally real-time consistent. In the next section, we will provide discussions about how this transformation based voxel structure can be improved further.

6.2 Future Work

Our algorithm suffers from voxel aliasing quite a lot. Just like pixel aliasing being a problem for modern games, voxel aliasing creates even more noticeable aliasing problem since voxel sizes are much larger than a pixel. Movement of objects; which covers all of the dynamic objects in the scene, will introduce visual flickering. However, those visual flickerings are less noticeable, because calculations only effect indirect illumination. Voxel aliasing can be reduced by holding a fraction for each axis and this fraction does not have to be precise and can be 8-bit

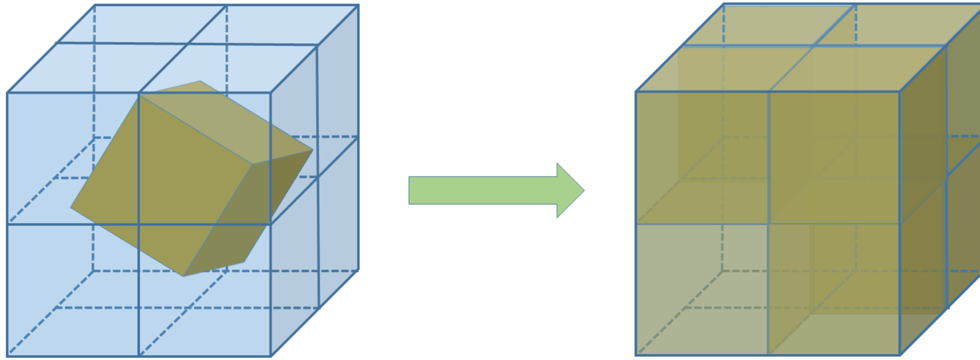

Figure 6.1: *After transformation, all eight adjacent voxels will be induced by occlusion which is determined by how much space this voxel is covering on that grid position. This will introduce 8 time more voxel values to be calculated by the SVO reconstruction algorithm.*

fixed point precision value that covers values between $[0, 1]$. Then at the construction stage, each voxel can be split into the nearest 8 neighbours by using these fractional values. Figure 6.1 illustrates the idea. However, this approach will increase the SVO reconstruction time since all voxels are required to be injected in to the SVO 8 times. This approach also introduces slight problems about atomic averaging since each voxel after transformation will not guaranteed to be fully occupied, which can be solved by changing the atomic averaging scheme.

Another improvement can be appplied on pre-voxelization of dynamic objects with joint transforms. We used nearest vertex sampling and it worked quite well in our joint deformed objects since the voxel sizes were greater than the average triangle size on the object. Even for finer voxel sizes, the voxel representation is a coarse representation anyway and noticable differences are minimal because, usage of the voxel representation is only for indirect illumination. However for finer voxel representation, better approach can be to utilize the method proposed in Dionne's paper [57]. Dionne's method is used to determine an automated method of generating vertex weights using voxelization. Since we already have voxelized object, this structure can be applied to this proposed method trivially. Only difference is that geodesic voxel binding requires dense voxelization and in our approach we used surface voxelization.

Next we can further improve the method by eliminating down to top averaging by averaging voxels during voxelization time. Instead of having only leaf nodes for each cascade, entire voxel mips of the object can be stored in cache then transformed accordingly. However this approach will introduce greater aliasing artefacts on higher levels of the tree. Unless, this method is combined with the voxel that holds fractions; which complements this approach and reduces aliasing artefacts, applicability of this method is minimal. Averaging process is not the bottleneck of the entire algorithm anyway, hence this improvement will yield minimal gains.

In our approach we used standart SVO structure that each parent holds 8 children. This approach can be further improved by using idea of Lefebvre et al. [49]. N^3-

tree method can be applied to reduce the tree depth but increases the memory requirement of the tree. This approach should increase the sampling on the GPU because most costly operation on the GPU is global memory transactions. By introducing lower tree level will make the SVO traversal faster.

By using N^3-tree fast linearly interpolatable SVO tree can be achieved by storing voxels at the corner points of the SVO node instead of storing voxels at the center point of the SVO node. Thus, with N^3-tree structure, neighbouring voxel structure memory transaction cost will be lower since it has a lower probability to traverse the entire tree to find its neighbouring node.

REFERENCES

[1] C. Crassin, "Gigavoxels: a voxel-based rendering pipeline for efficient exploration of large and detailed scenes", PhD thesis, Université de Grenoble, 2011, p. 207.

[2] L. Williams, "Casting curved shadows on curved surfaces", *SIGGRAPH Comput. Graph.*, vol. 12, no. 3, pp. 270–274, Aug. 1978, ISSN: 0097-8930. DOI: 10.1145/965139.807402.

[3] J. T. Kajiya, "The rendering equation", in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '86, New York, NY, USA: ACM, 1986, pp. 143–150, ISBN: 0-89791-196-2. DOI: 10.1145/15922.15902.

[4] M. Stamminger and G. Drettakis, "Perspective shadow maps", *ACM Trans. Graph.*, vol. 21, no. 3, pp. 557–562, Jul. 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566616.

[5] T. Saito and T. Takahashi, "Comprehensible rendering of 3-d shapes", *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4, pp. 197–206, 1990, ISSN: 00978930. DOI: 10.1145/97880.97901.

[6] B. Karis and E. Games, "Real shading in unreal engine 4", in *SIGGRAPH Course: Physically Based Shading in Theory and Practice*, 2013.

[7] S. Lagarde and d. C. Rousiers, "Moving frostbite to pbr", in *SIGGRAPH Course: Physically Based Shading in Theory and Practice*, 2014.

[8] T. Sousa, N. Kasyan, and N. Schulz, "Secrets of cryengine 3 graphics technology", in *SIGGRAPH Conferencce Presentation*, Vancouver, Canada: Crytek, 2011. [Online]. Available: http://www.crytek.com/cryengine/presentations/secrets-of-cryengine-3-graphics-technology.

[9] B. T. Phong, "Illumination for computer generated pictures", *Commun. ACM*, vol. 18, no. 6, pp. 311–317, Jun. 1975, ISSN: 0001-0782. DOI: 10.1145/360825.360839.

[10] B. Yalciner, *Githesis*, 2016. [Online]. Available: https://github.com/yalcinerbora/GIThesis (visited on 07/29/2016).

[11] B. Yalciner, *Gfg file format*, 2016. [Online]. Available: https://github.com/yalcinerbora/GFGFileFormat (visited on 07/29/2016).

[12] T. Ritschel, C. Dachsbacher, T. Grosch, and J. Kautz, "The state of the art in interactive global illumination", *Comput. Graph. Forum*, vol. 31, no. 1, pp. 160–188, Feb. 2012, ISSN: 0167-7055. DOI: 10.1111/j.1467-8659.2012.02093.x.

[13] E. P. Lafortune and Y. Willems, "Bi-directional path tracing", in *Compugraphics '93, Compugraphics '93, Alvor, Portugal*, Dec. 1993, pp. 145–153.

[14] E. Veach and L. J. Guibas, "Metropolis light transport", in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '97, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 65–76, ISBN: 0-89791-896-7. DOI: 10.1145/258734.258775.

[15] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces", *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 213–222, Jan. 1984, ISSN: 0097-8930. DOI: 10.1145/964965.808601.

[16] D. S. Immel, M. F. Cohen, and D. P. Greenberg, "A radiosity method for non-diffuse environments", *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 133–142, Aug. 1986, ISSN: 0097-8930. DOI: 10.1145/15886.15901.

[17] A. Keller, "Instant radiosity", in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '97, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 49–56, ISBN: 0-89791-896-7. DOI: 10.1145/258734.258769.

[18] C. Dachsbacher and M. Stamminger, "Reflective shadow maps", in *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, ser. I3D '05, Washington, District of Columbia: ACM, 2005, pp. 203–231, ISBN: 1-59593-013-2. DOI: 10.1145/1053427.1053460.

[19] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz, "Imperfect shadow maps for efficient computation of indirect illumination", *ACM Trans. Graph.*, vol. 27, no. 5, 129:1–129:8, Dec. 2008, ISSN: 0730-0301. DOI: 10.1145/1409060.1409082.

[20] T. Ritschel, E. Eisemann, I. Ha, J. D. K. Kim, and H.-P. Seidel, "Making imperfect shadow maps view-adaptive: High-quality global illumination in large dynamic scenes", *Computer Graphics Forum*, vol. 30, no. 8, pp. 2258–2269, 2011, ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2011.01998.x.

[21] C. Sun and E. Agu, "Many-lights real time global illumination using sparse voxel octree", in *Advances in Visual Computing: 11th International Symposium, ISVC 2015, Las Vegas, NV, USA, December 14-16, 2015, Proceedings, Part II*, G. Bebis, R. Boyle, B. Parvin, D. Koracin, I. Pavlidis, R. Feris, T. McGraw, M. Elendt, R. Kopper, E. Ragan, Z. Ye, and G. Weber, Eds. Cham: Springer International Publishing, 2015, pp. 150–159, ISBN: 978-3-319-27863-6. DOI: 10.1007/978-3-319-27863-6_14.

[22] H. W. Jensen, "Global illumination using photon maps", in *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, Porto, Portugal: Springer-Verlag, 1996, pp. 21–30, ISBN: 3-211-82883-4.

[23] L. Szirmay-Kalos, B. Aszódi, I. Lazányi, and M. Premecz, "Approximate ray-tracing on the GPU with distance impostors", *Computer Graphics Fo-*

rum, vol. 24, no. 3, pp. 695–704, 2005, ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2005.0m894.x.

[24] M. A. Shah, J. Konttinen, and S. Pattanaik, "Caustics mapping: An image-space technique for real-time caustics", *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 2, pp. 272–280, 2007, ISSN: 1077-2626.

[25] M. McGuire and D. Luebke, "Hardware-accelerated global illumination by image space photon mapping", in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09, New Orleans, Louisiana: ACM, 2009, pp. 77–89, ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572783.

[26] C. Yao, B. Wang, B. Chan, J. Yong, and J.-C. Paul, "Multi-image based photon tracing for interactive global illumination of dynamic scenes", in *Proceedings of the 21st Eurographics Conference on Rendering*, ser. EGSR'10, Saarbrücken, Germany: Eurographics Association, 2010, pp. 1315–1324. DOI: 10.1111/j.1467-8659.2010.01727.x.

[27] F. Sanglard, *Quake 2 source code review*, 2011. [Online]. Available: http://fabiensanglard.net/quake2/quake2_opengl_renderer.php (visited on 08/05/2016).

[28] N. Stefanov, "Deferred radiance transfer volumes: Global illumination in far cry 3", in *GDC 2012 Presentation*, Ubisoft, 2012. [Online]. Available: http://www.gdcvault.com/play/1015326/Deferred-Radiance-Transfer-Volumes-Global.

[29] P.-P. Sloan, J. Kautz, and J. Snyder, "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments", *ACM Trans. Graph.*, vol. 21, no. 3, pp. 527–536, Jul. 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566612.

[30] A. W. Kristensen, T. Akenine-Möller, and H. W. Jensen, "Precomputed local radiance transfer for real-time lighting design", in *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH '05, Los Angeles, California: ACM, 2005, pp. 1208–1215. DOI: 10.1145/1186822.1073334.

[31] A. Kaplanyan and C. Dachsbacher, "Cascaded light propagation volumes for real-time indirect illumination", in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '10, Washington, D.C.: ACM, 2010, pp. 99–107, ISBN: 978-1-60558-939-8. DOI: 10.1145/1730804.1730821.

[32] B. Thaut, *Light propogation volumes implementation*, 2011. [Online]. Available: http://3d.benjamin-thaut.de/?p=16 (visited on 08/01/2016).

[33] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing", *Computer Graphics Forum*, vol. 30, no. 7, pp. 1921–1930, 2014, ISSN: 14678659. DOI: 10.1111/j.1467-8659.2011.02063.x.

[34] R. Cupisz, *Light probes*, blogs.unity3d.com, Ed., Mar. 2011. [Online]. Available: http://blogs.unity3d.com/2011/03/09/light-probes/ (visited on 08/10/2016).

[35] Unity, *Unity documentation: Light probes*, docs.unity3d.com, Ed., Aug. 2016. [Online]. Available: https://docs.unity3d.com/Manual/LightProbes.html (visited on 08/12/2016).

[36] Epic, *Unreal engine 4 documentation: Indirect lighting cache*, docs.unrealengine.com, Ed., Aug. 2016. [Online]. Available: https://docs.unity3d.com/Manual/LightProbes.html (visited on 08/12/2016).

[37] J. Chaney, K. S. Engelstoft, and A. Simonar, "Advanced global illumination in unity 5", in *Unite 2015*, 2015.

[38] G. Hazel, C. Doran, and W. Joseph, "Reconstructing diffuse lighting from spherical harmonic data", in *CEDEC 2015*, 2015.

[39] M. Valient, "The rendering technology of killzone 2", in *GDC 2009 Presentation*, Guerilla Games, 2009. [Online]. Available: https://www.guerrilla-games.com/read/the-rendering-technology-of-killzone-2.

[40] S. Zhukov, A. Iones, and G. Kronin, "An ambient light illumination model", in *Rendering Techniques '98: Proceedings of the Eurographics Workshop*, G. Drettakis and N. Max, Eds. Vienna: Springer Vienna, 1998, pp. 45–55, ISBN: 978-3-7091-6453-2. DOI: 10.1007/978-3-7091-6453-2_5.

[41] M. Mittring, "Finding next gen: Cryengine 2", in *ACM SIGGRAPH 2007 Courses*, ser. SIGGRAPH '07, San Diego, California: ACM, 2007, pp. 97–121, ISBN: 978-1-4503-1823-5. DOI: 10.1145/1281500.1281671.

[42] P. Shanmugam and O. Arikan, "Hardware accelerated ambient occlusion techniques on gpus", in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ser. I3D '07, Seattle, Washington: ACM, 2007, pp. 73–80, ISBN: 978-1-59593-628-8. DOI: 10.1145/1230100.1230113.

[43] L. Bavoil, M. Sainz, and R. Dimitrov, "Image-space horizon-based ambient occlusion", in *ACM SIGGRAPH 2008 Talks*, ser. SIGGRAPH '08, Los Angeles, California: ACM, 2008, 22:1–22:1, ISBN: 978-1-60558-343-3. DOI: 10.1145/1401032.1401061.

[44] T. Stachowiak and Y. Uludag, "Advances in real-time rendering course: Sctochastic screen-space reflections", in *SIGGRAPH 2015 Course*, EA, 2015. [Online]. Available: http://www.slideshare.net/DICEStudio/stochastic-screenspace-reflections.

[45] Y. Uludag, "Hi-z screen-space cone-traced reflections", in *GPU Pro 5*, W. Engel, Ed., CRC Press, 2014, pp. 149–192.

[46] L. Hermanns and T. A. Franke, "Screen space cone tracing for glossy reflections", in *ACM SIGGRAPH 2014 Posters*, ser. SIGGRAPH '14, Vancouver, Canada: ACM, 2014, 102:1–102:1, ISBN: 978-1-4503-2958-3. DOI: 10.1145/2614217.2614274.

[47] P. Debevec, "Rendering synthetic objects into real scenes : bridging traditional and image-based graphics with global illumination and high dynamic range photography", *In Computer Graphics Proceedings, Annual Conference Series (Proc. ACM SIGGRAPH '98 Proceeding)*, pp. 189–198, 1998, ISSN: 00978930. DOI: 10.1145/280814.280864.

[48] J. Amanatides, "Ray tracing with cones", *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 129–135, Jan. 1984, ISSN: 0097-8930. DOI: 10.1145/964965.808589.

[49] S. Lefebvre, S. Hornus, and F. Neyret, "Gpu gems 2 - programming techniques for high-performance graphics and general-purpose computation", in. Addison Wesley, 2005, ch. Octree Textures on the GPU, pp. 595–613.

[50] M. Toksvig, "Journal of graphics, GPU and game tools", in. 2005, vol. 10, ch. Mipmapping Normal Maps, pp. 65–71.

[51] C. Crassin and S. Green, in *OpenGL Insights*. CRC Press, Patrick Cozzi and Christophe Riccio, Jul. 1, 2012.

[52] M. Schwarz and H.-P. Seidel, "Fast parallel surface and solid voxelization on GPUs", *ACM Transactions on Graphics*, vol. 29, no. 6 (Proceedings of SIGGRAPH Asia 2010), 179:1–179:9, Dec. 2010.

[53] J. Hasselgren, T. Akenine-Möller, and L. Ohlsson, in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005, ISBN: 0321335597.

[54] M. Takeshige, *The basics of gpu voxelization*, developer.nvidia.com, Ed., [Online; accessed 11-August-2016], Mar. 2015. [Online]. Available: https://developer.nvidia.com/content/basics-gpu-voxelization.

[55] Nvidia, *Tuning cuda applications for maxwell*, developer.nvidia.com, Ed., Feb. 2014. [Online]. Available: http://docs.nvidia.com/cuda/maxwell-tuning-guide/#axzz4HG3ApEyW (visited on 08/11/2016).

[56] P. Ljung, C. Lundström, and A. Ynnerman, "Multiresolution interblock interpolation in direct volume rendering", in *EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization*, B. S. Santos, T. Ertl, and K. Joy, Eds., The Eurographics Association, 2006, ISBN: 3-905673-31-2. DOI: 10.2312/VisSym/EuroVis06/259-266.

[57] O. Dionne and M. Lassa, "Geodesic voxel binding for production character meshes", in *Processing of the 12th ACM SIGGRAPH Eurographics Symposium on Computer Animation*, ser. SCA '13, Anaheim, California: ACM, 2013, pp. 173–180, ISBN: 978-1-4503-2132-7. DOI: 10.1145/2485895.2485919.

APPENDICES

APPENDIX A

Thesis Runtime Information

Overview

Thesis runtime is the implementation of the proposed method in this thesis. It can be downloaded from [10]. Most recent branch is the ”surfinterpol” branch. GitHub repository only holds the source code and shader code of the thesis and working directory can be found on this URL `https://www.dropbox.com/s/c6lacwxzhzfhyt7/WorkingDir.rar?dl=0`.

After downloading contents of the “WorkingDir” zip file needed to be extracted into the “WorkingDir” folder. WorkingDir folder contents should look like in Figure A.1.

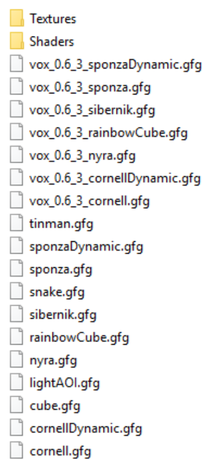

Figure A.1: *Thesis runtime working directory layout should look like this after extracting “WorkingDir.ra” file.*

It is tested on only Microsoft Windows platform and compiled only using Visual

Studio 2015. Feel free to port it to other compilers and platforms. Most of the libraries used by the thesis runtime are provided in the source code under Lib folder. However, big distributions like CUDA are required to be downloaded separately. Our implementation used Compute Capability 3.0 functionalities, because of that required graphics cars should at least have Kepler Architecture (GTX 600 Series cards, Quadro K Series cards and onward).

After those steps, code should compile and run fine. Runtime can be downloaded in a binary from from this link `https://www.dropbox.com/s/fstlmykiryr2pr3/GIThesisRT.rar?dl=0`.

Empty Solution and Input

Thesis runtime is separated into solutions. At first, Sponza scene with empty solution should come up. Empty solution meaning that there is no solution for the indirect illumination. Numpad 7 loads previous solution and Numpad 9 loads next solution on the solution chain. There is only single a solution in the chain currently, which is our implementation.

Scenes can be switched using numpad buttons 4 and 5. Entire scene chain is loaded in initialization time in order to switch between scenes quickly. At the time of this writing, there are total of seven scenes; which are Sponza Atrium, Oscillating Sponza Atrium, Cornell Box, Sibernik Cathedral, Simple Cube and Nyra.

There are three camera modes for the runtime. Program starts with static camera that you cannot change in any way. Pressing numpad 3 will change the camera movement to center of influence focused mode. Moving mouse while pressing mouse button 1 will rotate the camera with respect to the center of influence. You can move the center of influence by moving the mouse while pressing mouse button 3 and zoom in and out by using mouse scroll wheel. This camera movement is inspired by the classing modelling tool camera movements. Last movement is FPS movement; camera can be used to look around by pressing mouse button 1 and moving the mouse. W A S D keyboard buttons can be used to move the

camera. You can cycle through the camera mode by using numpad 4 and numpad 1. Finally numpad 2 moves the camera into a hard coded position. This used to take consistent screen shots for the written part of the thesis.

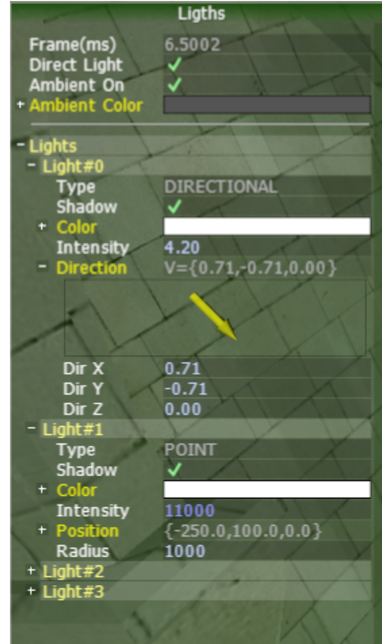

Figure A.2: *Thesis runtime light bar. Shows the scenes entire light entities. All of the lights can be adjusted using this GUI.*

All lights are rendered dynamically in the scene and lights can be adjusted and tweaked by using the light bar. Light bar shows the current frame time, on/off buttons for entire direct lighting and constant ambient lighting of the scene. Light direction, position, color and intensity values of each light can also be adjusted from here.

Thesis Solution

After changing solution, thesis bar will come up. Thesis bar has render drop down menu to show the system requested by the user.

On the Render Menu Fig A.3:

- **Voxel Caches** option is used to render voxel caches of the cascades. Each voxel rendered as a single cube because of that, performance can be low. Entire model space object cache is transformed to the world space in order

to get consistent world overview. “numpad +” and “numpad -” buttons can be used to render cascade caches.

- **Voxel Pages** option renders the voxel page system. Like Voxel Caches this option renders each voxel as a cube.
- **Voxel Octree** option renders consructed SVO. In order to debug the validity of SVO, simple ray marching algorithm is used to render the SVO structure. Because of this, performance is quite high compared to raster approaches use in “Voxel Pages” and “Voxel Caches” options. “numpad +” and “numpad -” buttons can be used to show different levels of the octree.
- **SVO Lookup** option uses G-Buffer position to query SVO to check if world pixels and voxels are algined with each other. If pixel misses the octree completely it will show up as gray. “numpad +” and “numpad -” buttons can be used to sample from different levels of the octree.
- **Light Intensity** option renders light intensity buffer which contains final lighting of the scene. Last step of the rendering pipeline is to multiply the values of this buffer with the color buffer.
- **Deferred** option shows the final image.

On “Voxel Caches”, “Voxel Pages”, “Voxel Octree” parts “numpad /” and “numpad *” can be used to see normals and occlusion portions of the voxel structure. Thesis bar initially will start rendering Voxel Page system using rasterizer.

In addition to that there is on/off buttons for Ambient Occlusion and Global Illumination. Moreover, it shows how many voxels are loaded by the cascade caches and usage of those voxels on the page system.

Timing portion of the thesis bar shows timings for each sub portion of the algorithm. Misc time is dependant on the render option, most of it shows the required overhead in order to show voxels into the scene.

When user chooses the option “Deferred” of “Light Intensity” cone bar will show up (See Fig A.4). Cone bar is used to effect cone tracing parameters. These include;

- **Cone Angle** is cone aperture in degrees. Increasing this value will increase performance but ambient occlusion and diffuse indirect illumination will be

Figure A.3: *Thesis runtime bar. Shows the voxel counts and timings. Also it has options to render voxel systems.*

overestimated.

- **Sample Factor** is used to determine distance between samples. Distance between the next sample and the previous sample will be multiplied by this value. Increasing this will increase performance but calculation will be less accurate.
- **Falloff Factor** determines how much light degrades with the distance. This option only effects visually and does not effect performance.
- **Max Distance** is the maximum world units distance of a cone can traverse. Reducing this will increase performance slightly.
- **Intensity** is the multiplication factor in order to see the effect more clearly and in a more expressive way.

Figure A.4: *Thesis cone bar. Shows the adjustable parameters for cone angles and sampling from the SVO.*

Voxelizer

Voxelizer program is a command line program which creates voxel cascades from GPU Friendly Graphics (GFG) files. For each object on the GFG file, it applies the algorithm explained in the Chapter 4, then outputs it to a another GFG file. Arguments of the Voxelizer.exe is as follows:

- **"-f"** switch is used to feed files to the voxelizer. These files should not be skeletal files. Those files will be fed using -fs swtich.
- **"-fs"** switch is used for files only contain skeletally animated mesh. Currently voxelizer and thesis runtime supports GFG files with single animation containing single skeletally animated mesh.
- **"-span"** switch is used to determine voxel span
- **"-cas"** switch is used to feed cascade count. For each cascade voxel span size is doubled.
- **"-splat"** switch is for determining a ratio to oversample the triangle by increasing frame buffer size.

Voxelizer then writes to files with "vox_'span'_'cascadeCount'_" prefix. Thesis runtime, while in solution load time, checks GFG file that has appropriate coverage value and use that file in the initialized solution.

APPENDIX B

GPU Friendly Graphics (GFG) File Format

Overview

Thesis runtime uses GFG file format which is designed along with the implementation of our solution. GFG file format is designed to support direct GPU vertex definitions provided by OpenGL or DirectX, hence the name GPU Friendly Graphics. GFG file format lays out the object vertices in such a way that single memory copy operation can be enough to copy the data from CPU memory to GPU memory (two if you include indices buffer). It also supports pre-compressed vertex data which can be used to open in shader by code or if GPU natively supports such data it can be un-packed by the hardware. GFG supports single indexed vertex layout since OpenGL and DirectX only supports such data.

GFG file format also supports simple materials and multiple material definitions on the same object. Those materials are used for shading of the different triangles in the object mesh. In addition to that, GFG File support skeletal animations and can store multiple animations in a single GFG File.

Maya Exporter

GFG File format comes with a Autodesk® Maya Importer/Exporter. Installation instructions can be found on the GFG GitHub repository [11]. Since the only file format supported by the thesis system is GFG, scenes that will be used by the thesis are required to be converted to GFG. Maya exporter can be used by this

purpose. Maya export options can be seen in Figure B.1.

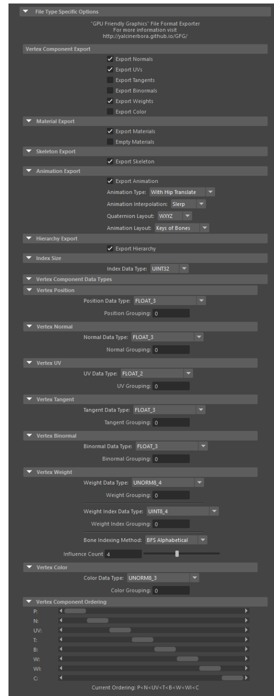

Figure B.1: *Export options of the GFG Maya.*

Thesis Integration

Thesis solution only supports Maya Phong materials for all the files. All of the textures used by the objects should be copied into the "Textures" folder in the working directory folder. Vertex layout and ordering should not be changed, default ordering is only supported by the thesis. For animated objects, GFG only supports joint rotations and hip translations. Other key frames will be ignored. Only export positions, normals and UVs for static objects should be exported. For skeleletally animated objects, export weight and weight indices should be exported in addition to positions, normals and UVs. Other vertex components may create inconsistencies or straight up crash the runtime with the shaders provided in the source.

Another limitation is that the models should have their scale and, if available, shear transformations incorporated into their vertices, since voxel transformations cannot cover scale and shear transformations as stated on the method limitations.

At the moment, source does not provide argument intake for scenes. In future iterations, there will be a command line argument system for loading custom multiple GFG files, defining custom lights and providing voxel system span and size values directly.