# INTERACTIVE DIFFUSE GLOBAL ILLUMINATION DISCRETIZATION METHODS FOR DYNAMIC ENVIRONMENTS

Athanasios Gaitatzes

University of Cyprus, 2012

Global illumination still finds limited use in interactive applications due to the overwhelming computational cost of solving for the transport of light in dynamic scenes, which is normally estimated in graphics through the *Rendering equation*. The solutions proposed in this dissertation are based on approximations that concentrate on discretization methods of the problem domain. First we considered the creation of a discretized representation of the visibility function around an object, as the exact visibility computation is expensive to compute in real-time. Then we examined the creation of a discretized representation of the incoming light in order to estimate diffuse interactions from multiple light bounces. Finally, we investigated the creation of a discretized representation of the scene geometry and use it for accelerating the above process.

For accelerating the visibility computation of the lighting equation in dynamic scenes composed of rigid objects, we pre-computed the visibility as seen from the environment, onto the bounding sphere surrounding the object and encoded it into maps. The visibility function is encoded by a four-dimensional visibility field that describes the distance of the object in each direction for all positional samples on a sphere around the object. Thus, we are able to speed up the calculation of most algorithms that trace visibility rays to real-time frame rates.

In order to estimate the diffuse interactions of light in dynamic environments we examined the creation of a discretized representation of the incoming light. We used a *Virtual Point Light* illumination model, representing indirect lighting as direct illumination from a cloud of point lights,

Athanasios Gaitatzes

on the volume representation of a complex scene. Unlike other dynamic VPL-based real-time approaches, our method handles occlusion (shadowing and masking) caused by the interference of geometry and is able to estimate diffuse inter-reflections from multiple light bounces in addition to energy from emissive materials.

As the bottleneck of the VPL-based approach was the volume generation of the scene, we investigated the discretization of the geometry of dynamic environments. We developed two real-time surface voxelization algorithms and a volume data caching structure, the *Volume Buffer*, which encapsulates functionality, storage and access similar to a frame buffer object, but for three-dimensional scalar data. The *Volume Buffer* can accumulate up to 1024 bits of arbitrary data per voxel, as required by the specific application. The efficient voxelization algorithm enables the fast generation of arbitrary scalar volume data attributes and is easy to integrate into existing frameworks, thus being able to produce illumination from multiple light bounces.

Additionally, we introduced the concept of *Incremental Voxelization* for the multi-valued, scalar volume rasterization of fully dynamic scenes. Where current image-based voxelization algorithms repeatedly regenerate the volume using the deferred geometry image buffers of a single frame, we incrementally update the existing voxels and therefore, produce a more complete voxelization of the scene, offering improved quality and stability at a small overhead.

Athanasios Gaitatzes

**INTERACTIVE DIFFUSE GLOBAL ILLUMINATION DISCRETIZATION METHODS**

**FOR DYNAMIC ENVIRONMENTS**

Athanasios Gaitatzes

M.S. Computer Science, Purdue University, 1991
B.S., Computer Science, Purdue University, 1989

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

August, 2012

Athanasios Gaitatzes

© Copyright by

Athanasios Gaitatzes

All Rights Reserved

2012

Athanasios Gaitatzes

# APPROVAL PAGE

Doctor of Philosophy Dissertation

**INTERACTIVE DIFFUSE GLOBAL ILLUMINATION DISCRETIZATION METHODS FOR DYNAMIC ENVIRONMENTS**

Presented by

Athanasios Gaitatzes

Research Supervisor ______________________
Dr. Yiorgos Chrysanthou

Committee Member ______________________
Dr. Constantinos Pattichis

Committee Member ______________________
Dr. Pedro Trancoso

Committee Member ______________________
Dr. Andreas Lanitis

Committee Member ______________________
Dr. Celine Loscos

University of Cyprus

August, 2012



Athanasios Gaitatzes

# DEDICATION

On September 22nd 2009 at 4:40 in the morning, a girl and a boy came into this world prematurely at 28 weeks and 2 days. They weighed only about 1200gr each. From their first moments they were struggling to survive in the neonatal intensive care unit of the hospital "Mother" at Amarousio of Attica Greece, under the supervision of an excellent team of doctors. My wife Zografia and I, decided to name our beautiful girl Eustratia-Angela, after her grandfather and our little boy George-Angelos, after his grandfather and grandmother.

**I would like to dedicate this research to my daughter Eustratia-Angela.**

**I wish I had the opportunity to get to know her.**

Little boy George-Angelos remained in the neonatal intensive care unit until the 21st of December. He is now about 30 months old and in very good health.

Athanasios Gaitatzes

# ΑΦΙΕΡΩΣΗ

Στις 22 Σεπτεμβρίου 2009 στις 4:40 το πρωί, ένα κοριτσάκι και ένα αγοράκι γεννήθηκαν πρόωρα στις 28 εβδομάδες και 2 ημέρες. Ζύγιζαν μόνο 1200γρ. το καθένα. Από την πρώτη στιγμή που γεννήθηκαν, αγωνίζονται να επιβιώσουν στη μονάδα εντατικής θεραπείας νεογνών του νοσοκομείου "Μητέρα" στο Αμαρούσιο Αττικής υπό την επίβλεψη μιας εξαιρετικής ομάδας γιατρών. Η γυναίκα μου Ζωγραφιά και εγώ, αποφασίσαμε να ονομάσουμε το όμορφο κοριτσάκι μας Ευστρατία-Αγγέλα, όπως λένε τον παππού της και το αγοράκι μας Γιώργο-Άγγελο, όπως λένε τον παππού και τη γιαγιά του.

**Θα ήθελα να αφιερώσω αυτή την έρευνα στην κόρη μου Ευστρατία-Αγγέλα.**

**Θα ήθελα να είχα την ευκαιρία να την γνωρίσω.**

Ο μικρός Γιώργος-Άγγελος παρέμεινε στην μονάδα εντατικής θεραπείας νεογνών μέχρι την 21η Δεκεμβρίου. Είναι τώρα περίπου 30 μηνών και χαίρει άκρας υγείας.

Athanasios Gaitatzes

# ACKNOWLEDGEMENTS

I would like to thank everyone who has helped and supported me throughout my Ph.D. experience.

First, I would like to express my sincere appreciation to my dissertation supervisor, Professor Yiorgos Chrysanthou. I am very grateful for his continuous and inspiring guidance during my studies. His valuable comments, suggestions and continuous encouragement were key to the successful completion of this dissertation.

Furthermore, I would like to express my gratitude to Professor Georgios Papaioannou for increasing my passion for this field and for his exhilarating conversations and help whenever I needed it.

I would also like to thank my other co-authors Andreadis Anthousis and Pavlos Mavridis for their insightful conversations.

Finally and most importantly, I wish to express my gratitude to my family and specially my wife Zografia, for her unwavering support, patience and understanding throughout the years of this dissertation.

Athanasios Gaitatzes

# CREDITS

The work presented in this dissertation appeared or has been submitted in the following journals, conferences and books:

**Journals**

- Gaitatzes A., Chrysanthou Y., Papaioannou G.: "Presampled Visibility for Ambient Occlusion". In Journal of WSCG (2008).

  http://wscg.zcu.cz/WSCG2008/Papers_2008/journal/B07-full.pdf

- Gaitatzes A., Papaioannou G., Chrysanthou Y.: "Incremental Image-based Voxelization for Real-time Indirect Illumination", Submitted for publication.

**Book Chapters**

- Gaitatzes A., Papaioannou G.: "Progressive Screen-space Multi-channel Surface Voxelization", In Wolfgang Engel (ed.) *GPU Pro 4: Advanced Rendering Techniques*, AK Peters / CRC Press. (2013).

  http://www.crcpress.com/product/isbn/9781466567436

**International conferences**

- Papaioannou G., Gaitatzes A., Christopoulos D.: "Efficient Occlusion Culling using Solid Occluders". In Proc. of the *14-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, (WSCG) (2006).

  http://wscg.zcu.cz/WSCG2006/Papers_2006/full/G79-full.pdf

Athanasios Gaitatzes

- Koniaris C., Gaitatzes A., Papaioannou G.: "An Automated Modeling Method for Multiple Levels of Real-Time Trees". In Proc. of the *First International IEEE Conference in Serious Games and Virtual Worlds*, (VS GAMES) (2009). http://www.computer.org/portal/web/csdl/doi/10.1109/VS-GAMES.2009.15
- Gaitatzes A., Andreadis A., Papaioannou G., Chrysanthou Y.: "Fast Approximate Visibility on the GPU using pre-computed 4D Visibility Fields". In Proc. of the *18-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, (WSCG) (2010). http://graphics.cs.aueb.gr/graphics/docs/papers/aowscg2010.pdf
- Gaitatzes A., Mavridis P., Papaioannou G.: "Interactive Volume-based Indirect Illumination of Dynamic Scenes". In Proc. of the *13-th International Conference on Computer Graphics and Artificial Intelligence*, (3IA) (2010).

  Also in Plemenos D., Miaoulis G. (eds.) *Intelligent Computer Graphics, Studies in Computational Intelligence*, 2010, vol. 321, pp. 229–245. Springer Berlin / Heidelberg.

  http://www.springerlink.com/content/e763tt336vh10852/
- Mavridis P., Gaitatzes A., Papaioannou G.: "Volume-based Diffuse Global Illumination". In Proc. of the *2010 International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing*, (CGVCVIP) (2010). http://www.iadisportal.org/digital-library/volume-based-diffuse-global-illumination
- Gaitatzes A., Mavridis P., Papaioannou G.: "Two Simple Single-pass GPU methods for Multi-channel Surface Voxelization of Dynamic Scenes". In Proc. of the *19-th Pacific Conference on Computer Graphics and Applications* - Short Papers, pp. 31–36, (PG) (2011). http://diglib.eg.org/EG/DL/PE/PG/PG2011short/031-036.pdf

Athanasios Gaitatzes

# TABLE OF CONTENTS

**List of Tables** xiv

**List of Algorithms** xvi

**List of Figures** xvii

**Chapter 1: Introduction** 1

1.1 Motivation . . . . . 1

1.2 Scope . . . . . 3

1.3 Contributions . . . . . 4

1.4 Organization . . . . . 7

**Part I Background and Related Work** 11

**Chapter 2: Theoretical Background** 13

2.1 The Physics of Light Transport . . . . . 14

2.2 Basic radiometric quantities . . . . . 16

2.3 Bidirectional Reflectance Distribution Function . . . . . 20

2.4 The Rendering Equation . . . . . 21

2.5 Programmable Hardware Evolution . . . . . 23

2.5.1 Vertex Processing Unit . . . . . 24

2.5.2 Pixel Processing Unit . . . . . 27

2.5.3 Geometry Processing Unit . . . . . 28

2.6 Deferred Shading . . . . . 28

Athanasios Gaitatzes

Chapter 3: Related Work 31

3.1 Methods that use Pre-computations . . . 32

3.1.1 Lightmaps . . . 32

3.1.2 Precomputed Radiance Transfer . . . 33

3.2 Methods that Simplify the Lighting Equation . . . 34

3.2.1 Ambient Occlusion . . . 34

3.2.1.1 Ambient Occlusion on the GPU - Screen Space AO . . . 36

3.2.1.2 Field Computations around an Object . . . 38

3.3 Methods that Discretize the Scene Geometry . . . 38

3.3.1 Radiosity . . . 39

3.3.2 Voxelization . . . 40

3.3.2.1 Geometry-based Surface Voxelization . . . 41

3.3.2.2 Image-based Surface Voxelization . . . 42

3.4 Methods that Discretize the Light Representation . . . 43

3.5 Brute Force Methods . . . 48

3.5.1 Ray-Tracing . . . 48

3.5.2 Real-time Ray-Tracing on the GPU . . . 50

Part II Discretization of Visibility – Ambient Occlusion and Secondary Light Bounces 53

Chapter 4: Fast approximate Visibility using pre-computed 4D Visibility Fields 55

4.1 Motivation . . . 55

4.2 Overview . . . 55

4.3 Introduction . . . 56

Athanasios Gaitatzes

4.4 Overview of the Visibility Fields . . . 59
4.4.1 Visibility Field Computation . . . 60
4.4.2 Visibility Field Indexing . . . 62
4.4.3 Selecting Samples around the Object . . . 63
4.4.4 Sampling a Hemisphere of Directions . . . 63
4.5 Visibility Fields on the GPU . . . 65
4.5.1 Ambient Occlusion . . . 65
4.5.2 Ray tracing . . . 66
4.6 Implementation & Evaluation on the CPU . . . 67
4.6.1 Ambient Occlusion . . . 67
4.6.1.1 Storage and Error Considerations . . . 67
4.6.1.2 Using the 8-bit maps . . . 68
4.6.1.3 Further Memory Optimization . . . 69
4.7 Implementation & Evaluation on the GPU . . . 71
4.7.1 Ambient Occlusion . . . 72
4.7.2 Ray tracing . . . 77
4.8 Limitations . . . 82
4.9 Summary . . . 83

**Part III Discretization of Illumination – Virtual Point Light Methods 85**

**Chapter 5: Interactive Volume-based Indirect Illumination of Dynamic Scenes 87**

5.1 Motivation . . . 87
5.2 Overview . . . 87
5.3 Introduction . . . 88

Athanasios Gaitatzes

5.4 Mathematical Background . . . 89
5.4.1 Review of Spherical Harmonics . . . 89
5.4.2 Radiance Transfer . . . 90
5.5 Method Overview . . . 91
5.5.1 Real-Time Voxelization . . . 94
5.5.2 Iterative Radiance Distribution . . . 96
5.5.3 Final Illumination Reconstruction . . . 97
5.6 Implementation & Evaluation . . . 99
5.7 Discussion . . . 104
5.8 Summary . . . 105

**Part IV Discretization of Geometry – Voxelization Methods 107**

**Chapter 6: Two Simple Single-pass GPU methods for Multi-channel Surface Voxelization of Dynamic Scenes 109**
6.1 Motivation . . . 109
6.2 Overview . . . 109
6.3 Introduction . . . 110
6.4 Overview of Voxelization methods . . . 112
6.4.1 Geometry Shader Triangle Slicing . . . 113
6.4.2 Pixel Shader Fragment Clipping . . . 115
6.5 Implementation . . . 117
6.6 Performance & Evaluation . . . 120
6.7 Discussion . . . 126
6.8 Summary . . . 128

Athanasios Gaitatzes

**Chapter 7: Incremental Image-based Multi-valued Voxelization for Global Illumination** **131**

7.1 Motivation . . . 131
7.2 Overview . . . 131
7.3 Introduction . . . 132
7.4 Overview of Voxelization method . . . 137
7.4.1 Clean-up phase . . . 139
7.4.2 Injection phase . . . 142
7.4.3 Single-pass Incremental algorithm . . . 142
7.5 Incremental Voxelization for Lighting . . . 144
7.6 Implementation . . . 145
7.7 Performance & Evaluation . . . 146
7.8 Optimizations . . . 155
7.9 Limitations . . . 155
7.10 Discussion & Summary . . . 157

**Chapter 8: Conclusion** **159**

8.1 Summary of Contributions . . . 159
8.2 Thoughts about Future Work . . . 162

**Bibliography** **165**

**Author Index** **175**

Athanasios Gaitatzes

# LIST OF TABLES

1 The *visibility fields* algorithm applied to several different types of models and their respective CPU timings (using machine type 1). In the above images we used 256 sample rays with a concentric map sampling distribution. The ambient occlusion computation is done using the 4226 / 32x32 maps. . . . . . . . . . . . . . . . . 71

2 Time measurements of our test scenes in milliseconds. Only the voxelization and propagation times are relevant to our work. The total rendering time includes the direct lighting computation and other effects and is given as a reference. Note that higher grid sizes are prohibitive using the current hardware. . . . . . . . . . . . . 100

3 Running time (in ms) for the construction of a half-float (16bit) single channel Occupancy Volume buffer for the two surface voxelization methods, based on the number of vertices that the geometry shader outputs. The third column gives the actual grid sizes as tight volume grids are generated dynamically. The last column reports the number of the resulting voxels. . . . . . . . . . . . . . . . . . . . . . 121

4 Comparison of the running time (in ms) for the bunny model for a floating (32bit) four channel buffer and different sizes of *multiple render targets* (MRTs). . . . . . 124

5 Comparison of the running time (in ms) and number of voxels produced by different approaches. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125

6 Comparison of the running time on various types of hardware of the bunny model at $128^3$ resolution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126

7 Comparison of the running time for 16- and 32-bit floating point buffers and 15 geometry shader vertices output. . . . . . . . . . . . . . . . . . . . . . . . . . 127



Athanasios Gaitatzes

8 Voxelization timings (in ms) of various scenes and methods. VP stands for *VoxelPipe* and GS is the Geometry Slicing method of Gaitatzes et al. with 11 output vertices. IV stands for *Incremental Voxelization*. We present the total (injection + cleanup) performance values of our 2D textures implementation using an injection grid proportional to the volume size, which is our algorithm's worst case as can be seen from the red plot of Figure 73. . . . . . . . . . . . . . . . . . . . . . . . . 152

9 Comparison of a full voxelization. We record the normalized (with respect to the mesh bounding box diagonal) average Hausdorff distance (percent). Mesh $X$ is the original mesh to be voxelized and $Y$ is the point cloud consisting of the voxel centers of the voxelization using *IV* (column 3) and a geometry-based full scene voxelization (column 4). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153


Athanasios Gaitatzes

# LIST OF ALGORITHMS

1 Pseudo-code for *Visibility Fields* computation at preprocessing time. . . . . . . . . 59

2 Pseudo-code for Ambient Occlusion rendering using *Visibility Fields* during real-time processing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

3 Fragment shader pseudo-code for Ambient Occlusion rendering, using *Visibility Fields*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

4 Pseudo-code for Scene Voxelization . . . . . . . . . . . . . . . . . . . . . . . . 95

5 Geometry Shader used for triangle slicing (Z-Pass). (ECS: Eye Coordinate Space) . 116

6 Geometry and Pixel Shaders used for triangle rasterization (Z-Pass). (ECS: Eye Coordinate Space, NDC: Normalized device Coordinates). . . . . . . . . . . . . 120

7 *Incremental Voxelization* pseudo-code using Camera G-buffers. . . . . . . . . . . . 139

Athanasios Gaitatzes

# LIST OF FIGURES

1 Photograph of a scene exhibiting global illumination effects like multiple diffuse and specular bounces, caustics and scattering. In this dissertation we are interested in simulating effectively diffuse illumination i.e. the green arrows. (Image courtesy of Tobias Ritschel). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 The *visibility fields* algorithm applied to several different types of models in order to compute inter-ambient occlusion ($1^{st}$ & $2^{nd}$ images) and intra-object ambient occlusion rendered on the GPU ($3^{rd}$ & $4^{th}$ images). . . . . . . . . . . . . . . . . . 5

3 Final images with direct and indirect lighting. . . . . . . . . . . . . . . . . . . . . 6

4 Multi-valued voxelization of several models showing the albedo channel. . . . . . 7

5 Side-by-side comparison of a single-frame image-based voxelization (left inset) vs. *incremental* image-based voxelization (right inset). The curtains that are hidden behind the colonnade do not obstruct the light in the case of the single-frame voxelization. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

6 A series of voxelizations of the dragon model at $128^3$ resolution showing the normals channel. The voxelization is *incrementally* updated over several frames as the camera moves around the model. . . . . . . . . . . . . . . . . . . . . . . . . 8

7 Visible spectrum . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

8 Spectral sensitivity curves $V(\lambda)$ and $V'(\lambda)$ for the human eye. . . . . . . . . . . . 15

9 The *radiant power* $\Phi$ of a light source is given by its total emitted radiation. . . . . 16

10 Definition of solid angle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

11 Typical directional distribution of *radiant intensity* for an incandescent bulb. . . . 18

12 Definition of radiance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19



Athanasios Gaitatzes

13 Determining the intensity at a point on a surface. . . . . . . . . . . . . . . . . . 20

14 The OpenGL® 1.0 graphics rendering pipeline; no programmable stages. . . . . . 23

15 The OpenGL® 4.1 graphics rendering pipeline; the programmable stages are in purple. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

16 Images from demos showing the capabilities of the hardware which translates in better image quality; courtesy of NVIDIA® . . . . . . . . . . . . . . . . . . . 25

17 NVIDIA GPU Shader Processors . . . . . . . . . . . . . . . . . . . . . . . . 26

18 Peak GFLOPS Chart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

19 Memory Bandwidth Chart . . . . . . . . . . . . . . . . . . . . . . . . . . 27

20 G-buffers used for Deferred Shading. . . . . . . . . . . . . . . . . . . . . . 29

21 Light mapping example. . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

22 An object under area lighting, without self-shadowing and with self-shadowing . . . . . 33

23 Ambient occlusion example courtesy of Morgan McGuire. . . . . . . . . . . . . 35

24 Radiosity examples from Cornell University. . . . . . . . . . . . . . . . . . . . 39

25 Photon mapping examples courtesy of Henrik Wann Jensen. . . . . . . . . . . . . 43

26 Ray-tracing example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

27 A hemisphere of rays emanating from the bounding sphere towards the object is precomputed for a large number of sample points on the sphere. . . . . . . . . . . 57

28 Volume texture of a *visibility field*. Row by row each map is placed into a slice of the volume texture thus minimizing the volume space requirements. As a result a $512^3$ volume will hold four $256^2$ maps per slice. . . . . . . . . . . . . . . . . . . 57

29 The distance from the surface to the object is split into distance $d_1$ (i.e. surface to object bounding-sphere ray intersection) and distance $d_2$ (bounding-sphere to object ray intersection, retrieved from the appropriate *visibility map*). . . . . . . . 60

Athanasios Gaitatzes

30 512x512 visibility maps of a model of a cow (top) and a cube with a hole in it (bottom). Using uniform Sampling of rays (left), Rejection Sampling (middle), Concentric Map Sampling (right). Different $(\theta, \phi)$ to $(s, t)$ mappings, produce different visibility maps. . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

31 Diagram of visibility computation for intra-object occlusion. . . . . . . . . . . 62

32 Sampling a hemisphere of rays. (a) Polar Mapping of rays, (b) Rejection Sampling, (c) Concentric Map Sampling. . . . . . . . . . . . . . . . . . . . . . . . . 64

33 4 bytes per ray for storage (left), 1 byte per ray for storage (right). There are no obvious visible differences, when the *visibility fields* are used for ambient occlusion. 67

34 The image differences between the Reference image and the visibility map methods, show that using Concentric map sampling produces much better quality results as compared to Uniform sampling. Image differences are exaggerated by a factor of 5. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

35 Cumulative table using 256 sample rays from each vertex of the tessellated corner (3x33x33) with a concentric map sampling distribution. The numbers under the images correspond to the preprocessing time, the run-time ambient occlusion computation on the CPU in seconds (using machine type 2) and the RMS error compared to the Reference image of Figure 34. . . . . . . . . . . . . . . . . . . 70

36 Using the 1 bit per direction optimization method with 4226 occlusion maps (positional samples) of size 64x64 and Concentric map sampling of 256 rays we get results which are comparable with the corresponding image from Figure 35, giving an RMS error of 4.4030. . . . . . . . . . . . . . . . . . . . . . . . . . . . 72



Athanasios Gaitatzes

37 Inter-object ambient occlusion (close-up) of a bunny model using the *visibility fields* method with 256 rays per pixel implemented on the GPU. We report the draw time and the RMS error. On the bottom right the reference image rendered on the GPU using 256 rays per pixel in 7126 ms. The model itself is rendered using fixed-pipeline direct rendering. . . . . . . . . . . . . . . . . . . . . . . 73

38 The draw time (left) and the RMS error (right) of the bunny model (39000 tris) plotted against different rays/pixel versus the size of the *visibility maps*. We observe that the frame draw time is not dependant on the number of *visibility maps* used or their size but rather on the amount of rays used. . . . . . . . . . . . . . . 74

39 The GPU *visibility fields* algorithm applied to several different types of models in order to compute inter-ambient occlusion. We used 4226 64x64 visibility maps, requiring 16.5 MB of space and 121 rays per pixel. The models themselves are rendered using fixed-pipeline direct rendering. . . . . . . . . . . . . . . . . . . . 75

40 Intra-object ambient occlusion rendered on the GPU using 16642 64x64 visibility maps requiring 65 MB of space and 121 rays per pixel. . . . . . . . . . . . . . . 76

41 A scene of the Sponza Atrium with a bunny (38889 tris), a cow (92864 tris) and an elephant (157160 tris) rendered in three passes (one per object) with the *visibility fields* algorithm using 4226x64x64 maps and rendering in 2.5 frames per second. . 77

42 Close-up of the bunny ears rendered using the *visibility fields* method for the generation of soft shadows using 3 lights and 20 shadow ray samples on the GPU. We report the required time, the RMS error and the total space requirements. . . . . . 78

43 Reference image of the bunny, rendered using the BVH method with 3 lights and 256 rays per pixel taking 913,210 ms on the GPU. On the right, close-up of the ears. 79

Athanasios Gaitatzes

44 Soft shadow of the horse (96966 tris) using 1 light rendered using the visibility fields and 20 shadow ray samples. In contrast the reference image, using the BVH method and 256 shadow ray samples, on the CPU required 760012 ms and on the GPU 322100 ms. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

45 Close-up of a more complex scene using 3 point lights and 20 shadow ray samples rendered in 3268 ms using the *visibility fields* method. The BVH GPU method for sharp shadows takes 48047 ms. . . . . . . . . . . . . . . . . . . . . . . . . . . 81

46 Polished reflection of the bunny (39000 tris) using 4 rays per reflective pixel. From left to right: close-up views of our *visibility fields* GPU method where we report the draw time, the RMS error and the space requirements. Last is the reference image using the BVH method and its draw time. . . . . . . . . . . . . . . . . . . . 81

47 Polished reflection of an elephant (157160 tris) using 4 rays per reflective pixel. From left to right: close-up views of our *visibility fields* GPU method where we report the draw time, the RMS error and the space requirements. Last is the reference image using the BVH method and its draw time. . . . . . . . . . . . . . . . . 82

48 Inter-ambient occlusion of a cane (elongated object) rendered on the GPU using 121 rays per pixel. On the left using 4226 128x128 maps and on the right using 16642 128x128 maps. (The image brightness is doubled for clarity) . . . . . . . . 83

49 Several environments voxelized into a $64^3$ grid. Column 1: a model of 10,220 triangles (Arena). Column 2: a model of 109,170 triangles (Knossos). Column 3: the Sponza II Atrium of 135,320 triangles (cross section depicted). All buffers are floating point and have values $\in$ [-1,1]. As such when viewing the buffers some black voxels may appear (see the normals map) indicating negative values. . . . . 93

Athanasios Gaitatzes

50 Slice-based voxelization (left) and composition of the three sub-volume passes into one voxelized volume (right). . . . . . . . . . . . . . . . . . . . . . . . . . 95

51 Radiance Gathering Illustration (a). The radiance for the center voxel is gathered from the values stored at the voxels of the surrounding cells. Radiance shooting (b) in the radiance propagation procedure is equivalent to radiance gathering (c). . 97

52 Simplified example of the propagation and light reflection process. . . . . . . . . 98

53 Test scene solution. From left to right: reference solution computed with ray-tracing (indirect illumination only), our solution (indirect illumination only) and final image with direct and indirect lighting. . . . . . . . . . . . . . . . . . . . . 101

54 Room scene solution; (a) lit with Direct lighting only. (b) Radiosity with 64 iterations. (c) Direct and indirect illumination using our method. (d) The indirect illumination using light propagation volumes [54]. (e) Reference radiosity using 2-bounce path tracing. (f) Reference final image using path tracing. . . . . . . . 102

55 Sponza Atrium II scene solution. From left to right: direct lighting, indirect illumination only and final image with direct and indirect lighting. . . . . . . . . . . 103

56 Arena scene solution. From left to right: direct only lighting, indirect illumination using our method and final image with direct and indirect lighting. . . . . . . . . 103

57 Knossos scene solution. From left to right: direct lighting, radiosity using our method and final image with direct and indirect lighting. . . . . . . . . . . . . . 104

58 Axial voxelization pass (left) and composition of the three sub-volume passes into one voxelized volume (right). . . . . . . . . . . . . . . . . . . . . . . . . . . 112

Athanasios Gaitatzes

59 Voxelization of the Knossos model (109170 triangles) into a $128^3$ grid. The volumes in the order that they appear are the occupancy volume, the albedo volume, the normals volume and the lighting volume and the $2^{nd}$ order spherical harmonics volume of the direct illumination (R component). . . . . . . . . . . . . . . . . . 114

60 Voxelization of the Sponza II model (219305 triangles) into a $128^3$ grid; cross section depicted here. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114

61 Voxelization of the Dragon model (871414 triangles) into a $128^3$ grid. . . . . . . . 115

62 Geometry shader triangle slicing. The incoming triangles are sliced into stripes and each stripe is rasterized into the associated layer. (See Algorithm 5) . . . . . . 117

63 The six possible triangle strip configurations with respect to the volume grid. (See Algorithm 5) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

64 Pixel shader clipping method. The Geometry shader rasterizes each triangle into all the volume slices it intersects and the Pixel shader discards the fragments based on their depth (See Algorithm 6). . . . . . . . . . . . . . . . . . . . . . . . . . 119

65 Visual comparison of the geometry shader triangle slicing method and the pixel shader clipping methods for the bunny model at $128^3$ volume resolution. Result with 11 output vertices. Gray voxels are common to both method variations. Green voxels are only present in the geometry shader triangle slicing, while red voxels are only generated by the pixel shader clipping. The total number of different voxels amounts to 33 which is about 0.15% of variation. . . . . . . . . . . . . 123

66 Comparison of the voxelization using the pixel shader clipping method at $256^3$ volume resolution with 3, 6, 9 and 12 geometry shader vertices output. The number of voxels produced are 52382, 87690 and 89696 (complete voxelization) for 3, 6, 9 and above geometry shader output vertices, respectively. . . . . . . . . . . 124

Athanasios Gaitatzes

67 Top: Image-based voxelization after one step of the process having injected the camera and light buffers. Middle: Voxelization of the scene after the camera has moved for several frames. Bottom: Example of resulting global illumination. . . . 135

68 *Incremental Voxelization* (IV) of a scene. Red voxels correspond to image-based voxelization using image buffers from the current frame only, while other colors refer to voxels generated during previous frames using IV. Right: Volume-based global illumination results using the corresponding volumes. IV achieves more correct occlusion and stable lighting. . . . . . . . . . . . . . . . . . . . . . . . 136

69 Schematic overview of the algorithm. During the cleanup phase each voxel is tested against the available depth images. If the projected voxel center lies in front of the recorded depth, it is cleared; otherwise it is retained. During the injection phase, voxels are "turned-on" based on the RSM-buffers and the Camera-based depth buffer. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138

70 Cleanup stage: Voxels beyond the boundary depth zone are retained (orange), while voxels closer to the buffer center of projection are rejected (red). Voxels that correspond to the depth value registered in the buffer must be updated (green). 141

71 Comparison of the voxelization of the Crytek Sponza II Atrium. (a), (b) Single frame image-based voxelization from two distinct viewpoints where it is not possible to capture all environment details as no information exists in the buffers. (c) *Incremental Voxelization* (*IV*) produced over several frames. (d) Complex illumination using *IV*. (e), (f) Indirect lighting buffers corresponding to the single frame voxelization of (a) and (b). (g), (h) *IV* indirect lighting buffers (of the voxelization in c). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147


Athanasios Gaitatzes

72 Image-based voxelization of a dynamic scene containing an articulated object using only camera-based injection. . . . . . . . . . . . . . . . . . . . . . . . . . . 148

73 Top: Running time (in ms) for the cleanup and injection stages against different volume resolutions for the Crytek Sponza II Atrium model. We used a single G-buffer (camera) as input and 1 MRT (4 floats) as output. Injection is measured for three different grid sizes, one being proportional to the volume side. Bottom: Total incremental voxelization times. Note that the performance of the optimized *Incremental Voxelization* is identical to that of the cleanup stage. . . . . . . . . . 150

74 Multi-channel voxelization performance for the Crytek Sponza II Atrium model, using 1 MRT (emitting 4 floating point values) and 4 MRTs (emitting 16 floating point values) in the GPU fragment shader stage. . . . . . . . . . . . . . . . . . . . 151

75 A series of voxelizations of the dragon model at $128^3$ resolution showing the normal vectors. The voxelization is *incrementally* updated and improved over several frames as the camera moves around the model. . . . . . . . . . . . . . . . . . . 154

76 The decreasing Hausdorff distance between the original dragon model and the computed *Incremental Voxelizations* of Figure 75. . . . . . . . . . . . . . . . . . 154

77 Correct indirect shadowing effects and color bleeding: Stale voxels from one view (behind the tank) are effectively invalidated in other views (reflective shadow map). 156

78 Scene with dynamic geometry, highlighting the shadowing effects of the tank model, as it moves towards the user, on the right wall of the tunnel. . . . . . . . . 156

79 Scene with dynamic lighting. Sequence of a side-by-side comparison of a single-frame image-based voxelization (left images) vs. *incremental* image-based voxelization (right-images). The curtains that are hidden behind the colonnade do not obstruct the light in the case of the single-frame voxelization. . . . . . . . . . . 158

Athanasios Gaitatzes

80 Diagram indicating the correlation of the *Discretization* methods used in this dissertation. The blue rectangles indicate the application domains. . . . . . . . . . . . 160

Athanasios Gaitatzes

# Chapter 1

## Introduction

This research has been conducted in the context of computer graphics, in particular in the field of realistic image synthesis. The goal of this research is to develop new algorithms that improve the quality of the illumination in large and fully dynamic complex environments or accelerate the expensive computation of existing algorithms, leading to as realistic illumination effects as possible in shorter computation time. The results of this work are applicable to real-time applications in order to produce as realistic illumination effects as possible.

### 1.1 Motivation

One track of computer graphics research is interested in creating images of arbitrary environments so that they are indistinguishable from photographs (i.e. photo-realistic rendering). The time required to generate one of these images is an important factor when we consider interactive three-dimensional worlds that are rendered in real-time like video-games or virtual reality reconstructions. The most involved part in this process is the simulation of light transport which adds important cues in the perception of virtual environments and is very time consuming.



Athanasios Gaitatzes

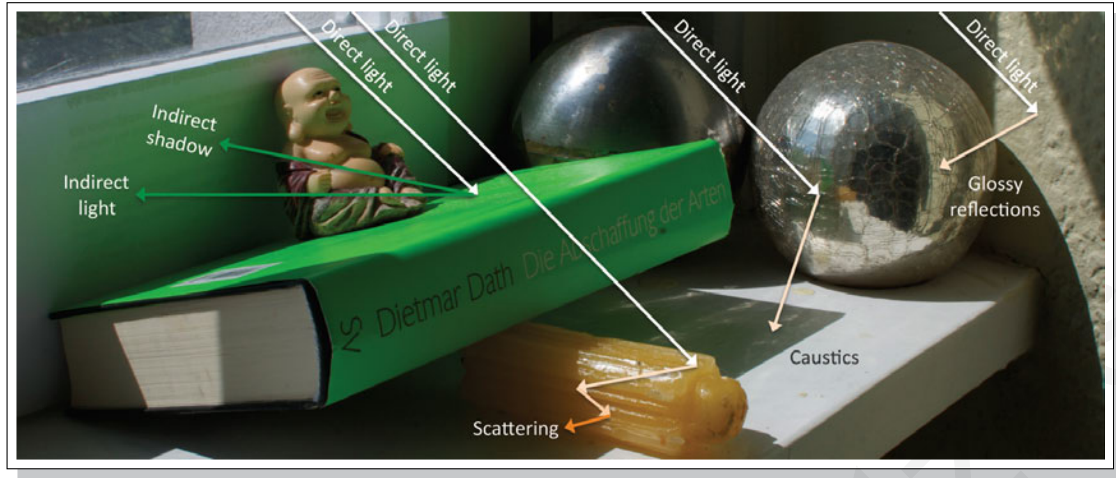



Figure 1: Photograph of a scene exhibiting global illumination effects like multiple diffuse and specular bounces, caustics and scattering. In this dissertation we are interested in simulating effectively diffuse illumination i.e. the green arrows. (Image courtesy of Tobias Ritschel).

Global Illumination is the light propagation through a 3D environment and its interaction with all the scene geometry. In contrast to direct or local illumination where only one bounce of light is considered, global illumination considers several bounces. Global illumination methods in interactive applications are still of limited use due to the overwhelming computational cost of the solution of the *Rendering equation* (see Equation 1). Although the equation is very general, it does not capture every aspect of light reflection. Effects like phosphorescence (i.e. the light is absorbed at one moment in time and emitted at a different time), fluorescence (i.e. the absorbed and emitted light have different wavelengths), interference (i.e. interaction of light sources) and subsurface scattering (i.e. the spatial locations for incoming and reflected light are different) can not be handled. The simulation of global illumination is a very costly process, as the interaction of light among all surfaces within a scene has to be taken into account several times, in order to achieve an accurate computation of direct and indirect illumination. In this interaction of light and

Athanasios Gaitatzes

geometry the key problem is computing the visibility or occlusion of one surface from another. This is an expensive computation which we will try to accelerate using discretization approaches.

Soft shadows, color bleeding, caustics, refractions and all other phenomena related to the interaction of light with the environment (see Figure 1) greatly enhance the visual perception of a scene. Determining the combined effect for several types of light transport (i.e. direct and indirect light, shadows and indirect shadows), has not been a real-time task so far. In addition when the environment is dynamic i.e. when the objects or the light sources in the environment are allowed to move, this task becomes even more difficult. Currently physically accurate global illumination can not be recomputed at real-time frame rates. The development of new techniques that accelerate the process of simulating global illumination effects can lead to higher realism in real-time or interactive applications, which require high frame rates.

With the rapid development of graphics hardware, global illumination has become increasingly attractive even for fully dynamic scenes. To this end we have developed a set of algorithms and techniques for rendering global illumination effects using graphics hardware. These algorithms not only result in real-time or interactive performance but also generate comparable quality to previous works.

### 1.2 Scope

The main objective of this work is to research methods and techniques that accelerate global illumination and produce visually plausible and realistic results for dynamic environments. The main issues that have been identified with existing work can be classified into:

- Performance of real-time algorithms
- Size of data sets
- Amount of pre-processing time (if-any)

Athanasios Gaitatzes

- Amount of extra storage (usually generated at pre-processing time) required during the execution of the real-time algorithm

To address these issues, this research will focus primarily on discretization techniques and how these can be further accelerated by the GPU in an attempt to improve and rectify problems of previous methods though acknowledging the risk of introducing extra errors. We will deal primarily with two genres of algorithms. First, we will look at geometry-based ambient occlusion and ray tracing methods. Under this category we are interested in discretization methods that accelerate the visibility function computation as it is one of the most expensive computation for solving the rendering equation. Then, we will address volume-based global illumination methods in an attempt to improve and rectify problems of previous methods. We will investigate discretization techniques that provide multi-valued surface voxelization data by either using geometry- or image-based voxelization, in order to accelerate arbitrarily complex illumination calculations such as ambient occlusion and diffuse indirect illumination from multiple light bounces.

In this investigation we will consider environments, including articulated objects and deformable geometry, that can change arbitrarily and dynamically in each frame. We will consider illumination that originates from a single point and spreads outward in all directions (point lights) in addition to illumination that originates from a single point and spreads outward in a coned direction (spot lights).

### 1.3 Contributions

In this research work we have made contributions in the following areas:

- **Discretization of Visibility** (i.e. Ambient Occlusion). The most expensive aspect in the computation of ambient occlusion is the calculation of the visibility function which is also a problem

Athanasios Gaitatzes

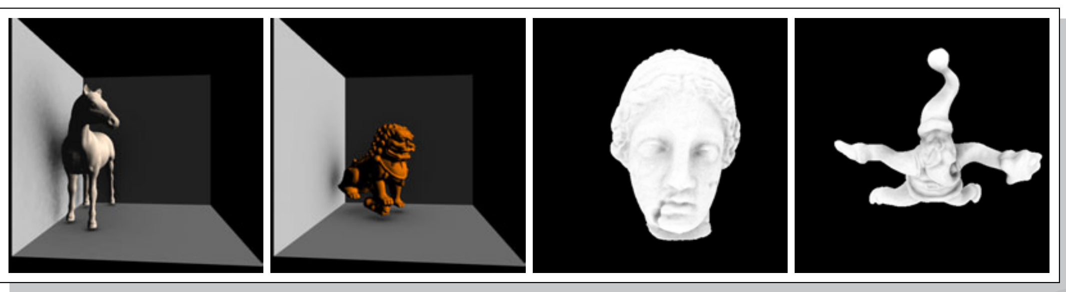


Figure 2: The *visibility fields* algorithm applied to several different types of models in order to compute inter-ambient occlusion ($1^{st}$ & $2^{nd}$ images) and intra-object ambient occlusion rendered on the GPU ($3^{rd}$ & $4^{th}$ images).

shared by other rendering methods. In order to accelerate the visibility function computation, we proposed a pre-computation, for each object in the scene, of the visibility information, as seen from the environment, onto the bounding sphere surrounding the object. The visibility function was encoded by a four-dimensional *visibility field* that described the distance of the object in each direction for all positional samples on a sphere around the object. Thus, we were able to speed up the calculation of most algorithms that trace visibility rays to real-time frame rates. The method has several advantages over the previous work. First the displacement maps are pre-calculated faster and stored as grayscale textures. Then, during the real-time simulation the time to access the displacement values is constant and in addition, the displacement maps contain information that is transformation invariant. Finally the method can handle several different cases like intra-object occlusion and inter-object occlusion but also shadow and reflection rays in the case of ray-tracing, cases which previous work [63], [70] could not handle (see Figure 2).

- **Discretization of Illumination** (i.e. Volume-based Global Illumination). In order to synthesize photo-realistic images we need to capture the complex interactions of light with the environment. Light follows many different paths distributing energy among the object surfaces. We proposed a

Athanasios Gaitatzes

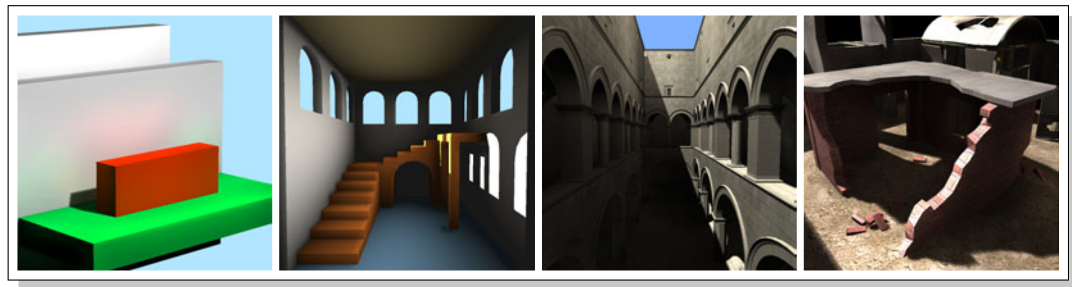


Figure 3: Final images with direct and indirect lighting.

real-time algorithm to compute the global illumination of dynamic scenes with complex dynamic illumination. We used a *Virtual Point Light* illumination model on the volume representation of the scene over *Light Propagation* methods. The method has several advantages over previous work [54]. It takes into account indirect occlusion (shadowing and masking) caused by the interference of geometry and is able to estimate diffuse inter-reflections from multiple light bounces thus producing more accurate illumination while always maintaining a high frame rate (see Figure 3).

- **Discretization of Geometry** (i.e. Voxelization). As an increasing number of rendering and geometry processing algorithms relies on volume data to calculate anything from effects like global illumination or visibility information, a fast and efficient computation of this volume in real-time is imperative. Voxelization was also one of the bottlenecks of the *Light Propagation Volumes* method. We propose two real-time and simple-to-implement surface voxelization algorithms and a volume data caching structure, the *Volume Buffer*, which encapsulates functionality, storage and access similar to a *frame buffer object*, but for three-dimensional scalar data. The *Volume Buffer* can rasterize primitives in 3D space and accumulate up to 1024 bits of arbitrary data per voxel, as required by the specific application. The method is much faster to compute than previous methods [11], [25] that perform rasterization-based voxelization by using the rendering pipeline (GPU). It

Athanasios Gaitatzes

also has the ability to store arbitrary data on each voxel (up to 1024 bits when using 8 MRT) (see Figure 4).

In addition, we introduced the concept of *Incremental Voxelization* for the multi-valued, scalar volume rasterization of fully dynamic scenes (geometry, materials and lighting) and demonstrated its application in the context of volume-based global illumination (see Figure 5). Where current image-based voxelization algorithms [55] repeatedly regenerate the volume using the deferred geometry image buffers of a single frame, we incrementally update the existing voxels using a depth-buffer re-projection scheme and therefore, produce a more complete voxelization of the scene. *Incremental Voxelization* can be used for multi-attribute volumes and complex dynamic scenes (see Figure 6).

The *Incremental Voxelization* framework will be publicly available from the website of the author under the appropriate paper.

(`http://www.virtuality.gr/gaitat/en/publications.html`).

## 1.4 Organization

In this section we outline the structure of this dissertation and summarize its contents. In Chapter 2 we give the theoretical background of global illumination and the *Rendering equation*. In addition, we present the evolution of programmable hardware. Then in Chapter 3 we present

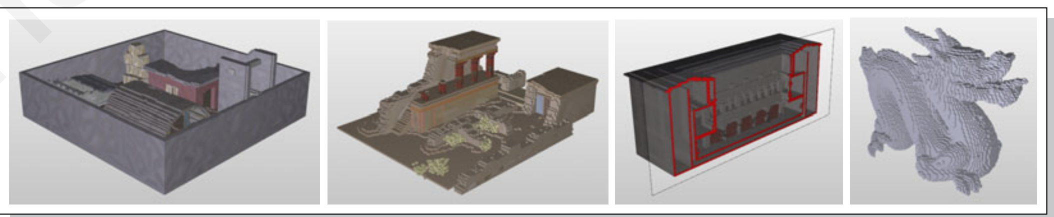


Figure 4: Multi-valued voxelization of several models showing the albedo channel.

Athanasios Gaitatzes

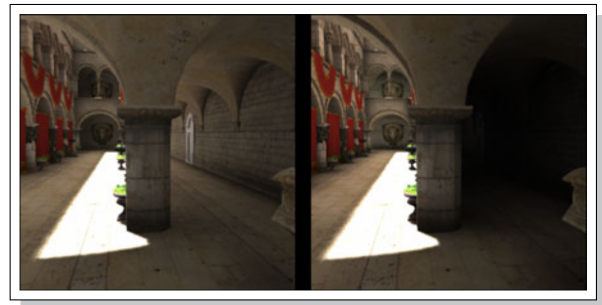


Figure 5: Side-by-side comparison of a single-frame image-based voxelization (left inset) vs. *incremental* image-based voxelization (right inset). The curtains that are hidden behind the colonnade do not obstruct the light in the case of the single-frame voxelization.

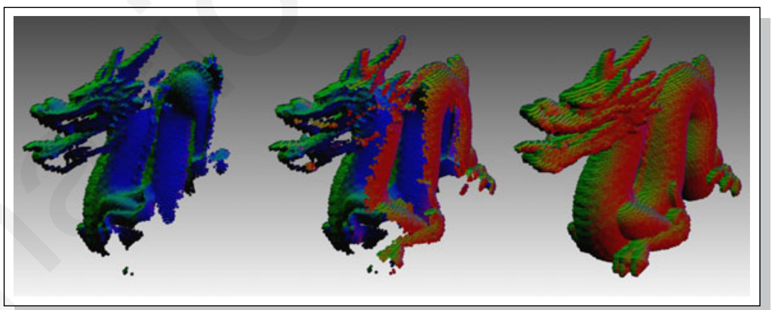


Figure 6: A series of voxelizations of the dragon model at $128^3$ resolution showing the normals channel. The voxelization is *incrementally* updated over several frames as the camera moves around the model.

Athanasios Gaitatzes

an overview of the existing related methods for solving the problem. In Chapter 4 we create a discretization of the visibility function by clustering together visibility rays, which is directly applicable to ambient occlusion and ray tracing calculations where exact ray hits are not critical, such as soft shadow rays (see [29] and [30]). In Chapter 5 we create a discretization of the light in the scene using *Virtual Point Light* methods by generating maps from the perspective of the light sources and injecting the samples of these maps into a volume while at the same time creating an additional volume that would hold the occlusion / visibility of the geometry in the scene (see [31] and [71]). Finally, in Chapters 6 and 7 we create a discretization of the scene geometry using an intermediate regular approximation in order to store lighting and geometry data and simulate indirect illumination (see [32], [33]). We conclude this dissertation and discuss some future work in Chapter 8.

Athanasios Gaitatzes

Athanasios Gaitatzes

# Part I

**– Background and Related Work –**

Athanasios Gaitatzes

Athanasios Gaitatzes

# Chapter 2

## Theoretical Background

One requirement for a scene to have any semblance of photorealism is the appearance of objects within a scene exhibiting influence over each other. Global illumination (GI) is an important factor in creating realistic scenes. A key role of Global Illumination is to provide visual cues about the geometric relationship of objects in an image. The goal in Global Illumination is to compute all possible light interactions in a given scene and thus obtain a truly photo-realistic image. All combinations of diffuse and specular reflections and transmissions must be accounted for. Hard to achieve effects such as color bleeding, caustics and refractions must be included in a global illumination simulation. However, global illumination is very costly and only recently has it become viable to render scenes with global illumination effects at interactive frame rates by exploiting the parallelism and programmability of modern Graphics Processing Units (GPU).

In order to understand how to simulate global illumination effects we must first understand the behavior of light. In addition, we must have a good comprehension of the operations of the hardware and the new programmable possibilities that each new generation has to offer.



Athanasios Gaitatzes

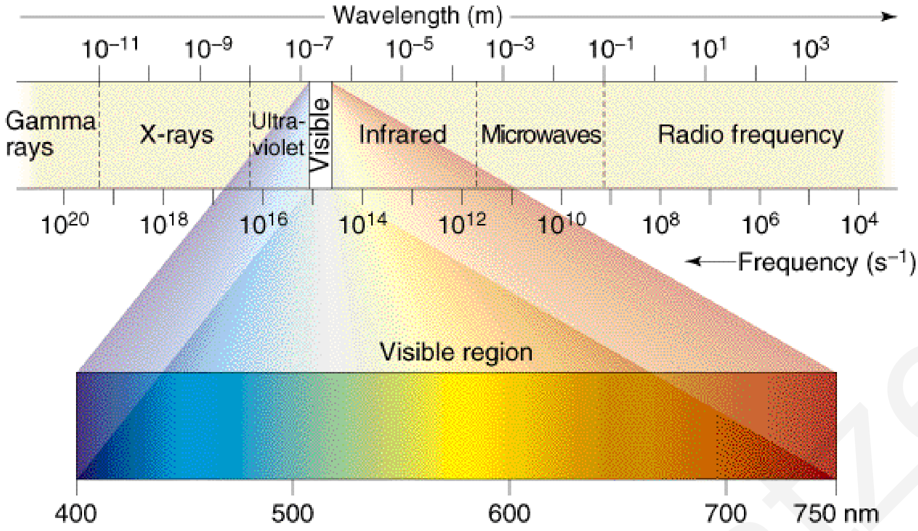



Figure 7: Visible spectrum

## 2.1 The Physics of Light Transport

Light is a form of energy manifesting itself as electromagnetic radiation and is closely related to other forms of electromagnetic radiation such as radio waves, radar, microwaves, infrared and ultraviolet radiation and X-rays.

The only difference between the several forms of radiation is in their wavelength. Radiation with a wavelength between 380 and 780 nanometers forms the visible part of the electromagnetic spectrum and is therefore referred to as light. The eye interprets the different wavelengths within this range as colors moving from red, through orange, green, blue to violet as wavelength decreases. Beyond red is infrared radiation, which is invisible to the eye but detected as heat. At wavelengths beyond the violet end of the visible spectrum (see Figure 7) there is ultraviolet radiation that is also invisible to the eye. White light is a mixture of visible wavelengths, as is demonstrated for example by a prism which breaks up white light into its constituent colors.

Under daylight conditions, the average normal sighted human eye is most sensitive at a wavelength of 555 nm, resulting in the fact that green light at this wavelength produces the impression

Athanasios Gaitatzes

of highest "brightness" when compared to light at other wavelengths. The spectral sensitivity function (see Figure 8) of the average human eye under daylight conditions (photopic vision) is defined by the CIE spectral luminous efficiency function $V(\lambda)$. Only in very rare cases, the spectral sensitivity of the human eye under dark adapted conditions (scotopic vision), defined by the spectral luminous efficiency function $V'(\lambda)$, becomes technically relevant. By convention, these sensitivity functions are normalized to a value of 1 in their maximum.

As an example, the photopic sensitivity of the human eye to monochromatic light at 490 nm amounts to 20% of its sensitivity at 555 nm. As a consequence, when a source of monochromatic light at 490 nm emits five times as much power (expressed in watts) than an otherwise identical source of monochromatic light at 555 nm, both sources produce the impression of same "brightness" to the human eye.

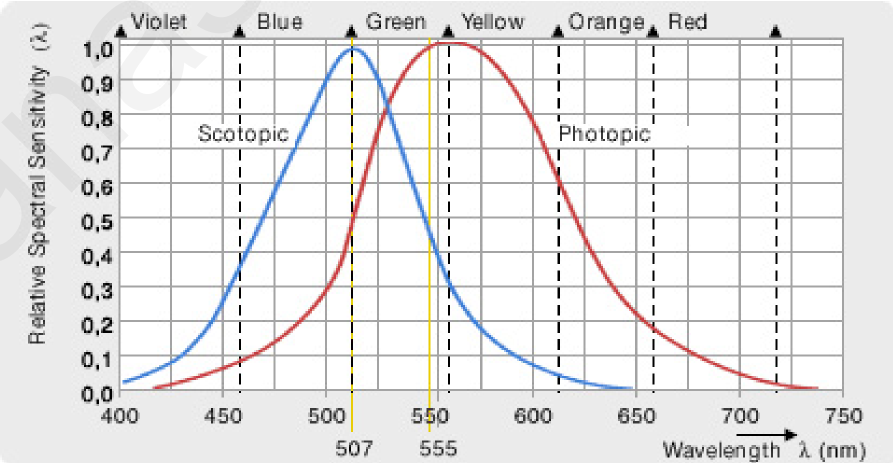



Figure 8: Spectral sensitivity curves $V(\lambda)$ and $V'(\lambda)$ for the human eye.

Athanasios Gaitatzes

## 2.2 Basic radiometric quantities

The whole discipline of optical measurement techniques can be roughly subdivided into the two areas of photometry and radiometry. Whereas the central problem of photometry is the determination of optical quantities closely related to the sensitivity of the human eye, radiometry deals with the measurement of energy per time emitted by light sources or impinging on a particular surface.

To compute the distribution of light energy in a scene we need to understand the physical quantities that represent light energy. Radiometry is the area of study involved in the physical measurement of light and provides a set of mathematical tools to describe light transport. All radiometric quantities are wavelength dependent but we will not make this dependency explicit.

*Radiant energy*, denoted by the symbol $Q$, is emitted from a light source or reflected from a surface and is transferred through space as photons. Radiant energy is the total energy emitted as radiation of all wavelengths in a defined period of time and is measured in *J* (*Joules*).

The fundamental radiometric quantity is *radiant power* (Figure 9), also called *flux*. It is denoted by the symbol $\Phi$ and is expressed in *Joules/sec* (*J/s*), or more commonly *Watts* (*W*). Radiant power expresses the total amount of energy that flows from/to/through a surface or region of space

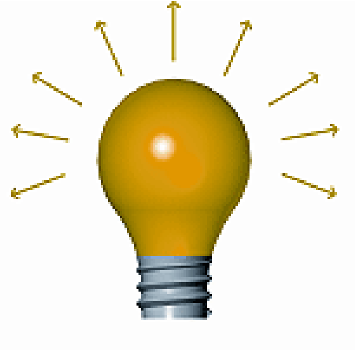


Figure 9: The *radiant power* $\Phi$ of a light source is given by its total emitted radiation.

Athanasios Gaitatzes

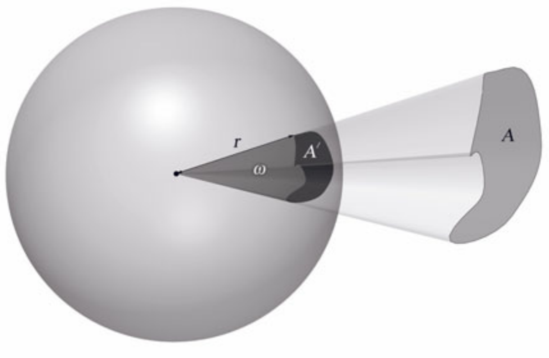



Figure 10: Definition of solid angle

per unit time.

$$\Phi = dQ/dt$$

For example, we can say that a light source emits 50 *Watt* radiant power, or that 20 *Watt* of radiant power is incident on a table.

Radiant power does not specify the size of the light source or the receiver surface, nor does it include a specification of the distance between the light source and the receiver. If a light source emits uniformly in all directions, it is called an isotropic light source. Total emission from light sources is generally described in terms of flux.

In order to define *intensity*, we first need to define the notion of a *solid angle* (Figure 10). Solid angles are just the extension of two-dimensional angles in a plane to an angle on a sphere. The *planar angle* is the total angle subtended by some object with respect to some position. Consider the unit circle; if we project an object onto the circle, some length of the circle will be covered by its projection. This arc length is the angle subtended by the object and is measured in *radians*. The solid angle extends the 2D unit circle to a 3D sphere. The solid angle $\omega$ subtended by a surface patch $A$ is defined as the area of the projection $A'$ of $A$ on the surface of a sphere of radius $r$,

Athanasios Gaitatzes

divided by $r^2$. Solid angles are measured in *steradians*. An entire sphere subtends a solid angle of $4\pi$ and a hemisphere subtends $2\pi$ *steradians*.

We can now define intensity. *Radiant intensity* (Figure 11), is denoted by the symbol $I$ and is expressed in *Watts/steradians*. It describes the radiant power density per unit of solid angle $\omega$ in a certain direction.

$$I = d\Phi / d\omega$$

The energy emitted or reflected from a point may be restricted to certain directions or it may be spreading equally in all directions. In general, radiant intensity depends on spatial direction.

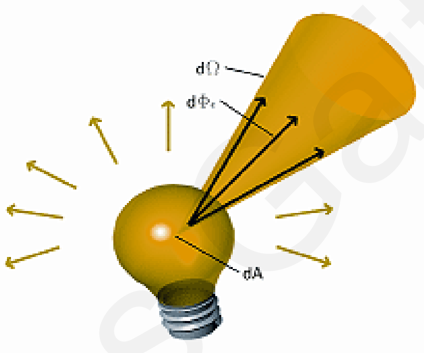



Figure 11: Typical directional distribution of *radiant intensity* for an incandescent bulb.

*Radiance* (Figure 12) is the most important quantity in global illumination algorithms because it is the quantity that captures the “appearance” of objects in a scene. It is denoted by the symbol $L$ and is expressed in *Watts/(steradians* $\cdot$ $m^2$*)*. Radiance is defined as the radiant power density per unit solid angle leaving or entering an infinitesimal area $dA$ from a certain direction, per unit projected surface area in that direction.

$$L = \frac{d\Phi}{d\omega dA^{\perp}} = \frac{d\Phi}{d\omega dA \cos\theta}$$

Thus, it is the limit of the measurement of incident light at the surface as a cone of incident directions of interest $d\omega$ becomes very small and as the local area of interest on the surface $dA$

Athanasios Gaitatzes

also becomes very small. Due to the solid angle, radiance is inversely proportional to the square of the distance from the light source.

*Irradiance* is denoted by the symbol *E* and is expressed in *Watt/*$m^2$. It describes the amount of radiant power impinging from all directions upon a surface per unit area. Irradiance arriving at a surface varies according to the cosine of the angle of incidence of illumination, since illumination is over a larger area and smaller incident angles (*Lambert's Law*).

$$E = d\Phi/dA$$

For example, if 50 *Watt* radiant power is incident on a surface which has an area of 1.25 $m^2$, the irradiance at each surface point is 40 *Watt/*$m^2$ (assuming the incident power is uniformly distributed over the surface).

Similarly *radiant exitance*, denoted by the symbol *M*, also called *radiosity*, denoted by the symbol *B*, is expressed in *Watt/*$m^2$. It describes the exitant radiant power per unit surface area.

$$M = B = d\Phi/dA$$

For example, consider a light source, of area 0.1 $m^2$, which emits 100 *Watts*. Assuming that the power is emitted uniformly over the area of the light source, the radiant exitance of the light is 1000 *Watt/*$m^2$ at each point of its surface.

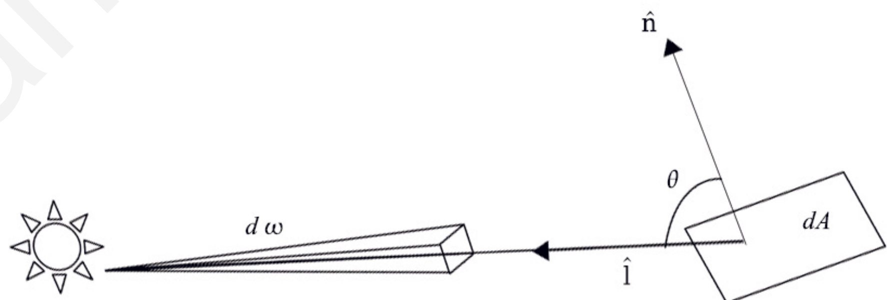



Figure 12: Definition of radiance

Athanasios Gaitatzes

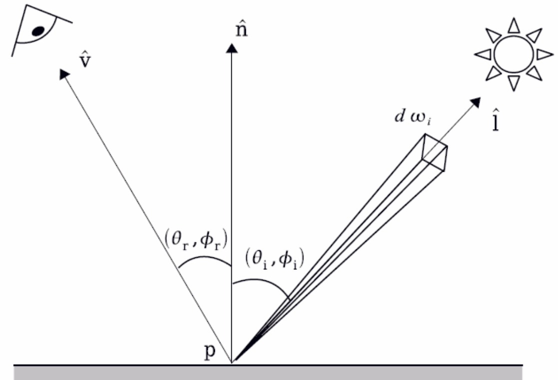



Figure 13: Determining the intensity at a point on a surface.

The interested reader can find more detailed information in [35], Illingworth [47], Dutre et al. [21], Pharr et al. [85] and Theoharis et al. [107].

### 2.3 Bidirectional Reflectance Distribution Function

We are interested in the relationship between the incident light from a certain direction onto a surface and the reflected light in another direction as well as the transmitted light through the object. This relationship is captured by the *Bidirectional Reflectance Distribution Function* (BRDF) [78]. The BRDF (Figure 13) depends on many parameters; lighting and observation directions, wavelength, shadow casting, the optical properties of the object, reflectivity, absorption, emission, etc. In practice, it can only be approximated and is also well known to the remote-sensing and modern painting communities. The BRDF associates the outgoing radiance $dL_r$ in direction $(\theta_r, \phi_r)$ to the irradiance $dE_i$ from the incident direction $(\theta_i, \phi_i)$

$$BRDF = \frac{dL_r(\theta_r, \phi_r)}{dE_i(\theta_i, \phi_i)}$$

Essentially the BRDF captures the fact that objects look differently when seen from different angles or when illuminated from different directions.

Athanasios Gaitatzes

## 2.4 The Rendering Equation

In order to accurately model light in an environment, the energy transfer has to be evaluated on each surface location. The *Rendering equation*, proposed by Kajiya [52], associates the outgoing radiance $L_o(\mathbf{x}, \vec{\omega}_\mathbf{o})$ from a surface point **x** along a particular viewing direction $\vec{\omega}_o$, with the intrinsic light emission $L_e(\mathbf{x}, \vec{\omega}_\mathbf{o})$ at **x** and the incident radiance from every direction $\omega_i$ in the hemisphere $\Omega$ centered above **x**, using a *Bidirectional Reflectance Distribution Function* (BRDF) [78] that depends only on the material properties and the wavelength of the incident light. The hemisphere-integral form of the *Rendering equation* can be written as:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + L_r(\mathbf{x}, \vec{\omega}_o)$$

$$L_r(\mathbf{x}, \vec{\omega}_r) = \int_\Omega L_i(\mathbf{x}, \vec{\omega}_i)\ f_r(\mathbf{x}, \vec{\omega}_i \to \vec{\omega}_r)\ (\vec{\omega}_i \cdot \vec{n})\ d\vec{\omega}_i$$

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int_\Omega L_i(\mathbf{x}, \vec{\omega}_i)\ f_r(\mathbf{x}, \vec{\omega}_i \to \vec{\omega}_o)\ (\vec{\omega}_i \cdot \vec{n})\ d\vec{\omega}_i \tag{1}$$

More specifically:

| | |
|---|---|
| $L_o(\mathbf{x}, \vec{\omega}_o)$ | is the outgoing radiance at position $\mathbf{x}$, along direction $\vec{\omega}_o$ |
| $L_e(\mathbf{x}, \vec{\omega}_o)$ | is the emitted radiance at position $\mathbf{x}$, along direction $\vec{\omega}_o$ |
| $L_r(\mathbf{x}, \vec{\omega}_o)$ | is the reflected radiance at position $\mathbf{x}$, along direction $\vec{\omega}_o$ |
| $L_i(\mathbf{x}, \vec{\omega}_i)$ | is the incoming radiance at position $\mathbf{x}$, along direction $\vec{\omega}_i$ |
| $f_r(\mathbf{x}, \vec{\omega}_i \to \vec{\omega}_o)$ | is the BRDF of the surface at point $\mathbf{x}$, expressing how much of the incoming light arriving at $\mathbf{x}$ along direction $\vec{\omega}_i$ is reflected along the outgoing direction $\vec{\omega}_o$ |
| $(\vec{\omega}_i \cdot \vec{n})$ | is the attenuation of incoming light due to incident angle |
| $\int_\Omega \ldots d\vec{\omega}_i$ | is an integral over the hemisphere $\Omega$ of incoming directions |
| $\Omega$ | is the hemisphere domain centered around position $\mathbf{x}$ |

Athanasios Gaitatzes

The above integral takes into account all of the incoming light and computes the reflected light. It takes into account all light paths such as those that contribute to caustics and global illumination. Solving Equation (1) for all surfaces simultaneously would result in the computation of the lighting in the observed environment. Of course, as Equation (1) implies, an infinite number of points and incident paths are required for its solution so the outgoing radiance is approximately estimated using numerical methods. All Global Illumination algorithms rely on this concept and can be thought of as special cases of the solution to the general *Rendering equation*.

Long before the actual mathematical formulation of the problem, many algorithms had started to appear in order to solve the problem. Two general sets of algorithms exist in order to simulate Global Illumination; light path tracing algorithms (to account for ray-tracing, forward ray-tracing, photon mapping, path and beam tracing etc.) and radiosity algorithms. Ambient occlusion methods simplify the above equation and only mimic global illumination effects. These techniques for accurately simulating Global Illumination generate realistic results but their rendering times are slow. Timings are often measured in minutes per frame. Their real-time counterparts should be able to produce Global Illumination effects in milliseconds.

A few years ago, dedicated graphics hardware solely excelled at drawing triangles with filled textures. But gaming has been the dominant force in pushing the advancement of computer graphics technology. This relationship is indirectly reciprocated when advanced graphics techniques work their way into the game production pipeline. With advancements in computer graphics and the introduction of programmable graphics boards, methods for simulating global illumination in real-time began to arise. It is now possible to execute custom pieces of code on graphics hardware, taking advantage of the parallelism of the current system architectures. These new architectures are being used today in order to advance the computations of Global Illumination into the real-time realm.

Athanasios Gaitatzes

### 2.5 Programmable Hardware Evolution

Graphics hardware acceleration is a rapidly evolving area (see Figure 16). In August 1999 the first Graphics Processing Unit (GPU) [116] was introduced to the consumer level hardware market. It integrated the entire graphics pipeline in one graphics chip and supported user programmability for some stages. In 2001, the first chip with custom programmable features was introduced but required developers to use assembly language. In addition, the programmable vertex and pixel shader pipelines used separate parts on the chip, creating mostly throughput problems. Then in 2006 a unified architecture model was introduced in addition to a C-like interface for programming. After this, the *programmable function pipeline* has been widely supported in graphics hardware to replace the previous *fixed function pipeline*. Currently the newest GPU hardware have become very flexible and easy to program via a graphics API, such as OpenGL® [96] (see Figure 15) or Direct3D [117]. The advancements of the GPU has been driven by the evolution of the so called *Shader Model* (SM), a set of features that must be supported by the programmable shader units of the GPU. The first "modern" graphics hardware supporting programmable shading was SM 1.0 (NVIDIA® GeForce 3 (NV20) [118], Direct3D 8.0 [117]). Programmable shading evolved with more and more flexibility up to todays SM 5.0 (NVIDIA® GeForce GTX480 [120], Direct3D 11.0 [5]). The interested reader can find more details on the evolution of these shader models and the commodity graphics hardware in general in Akenine-Moller et al. [1].

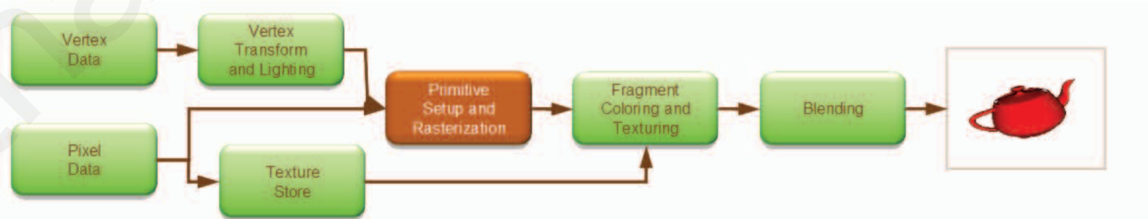



Figure 14: The OpenGL® 1.0 graphics rendering pipeline; no programmable stages.

Athanasios Gaitatzes

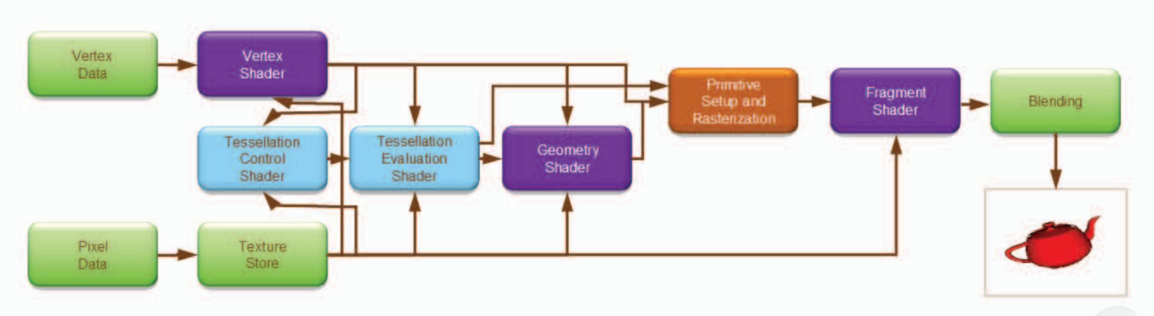


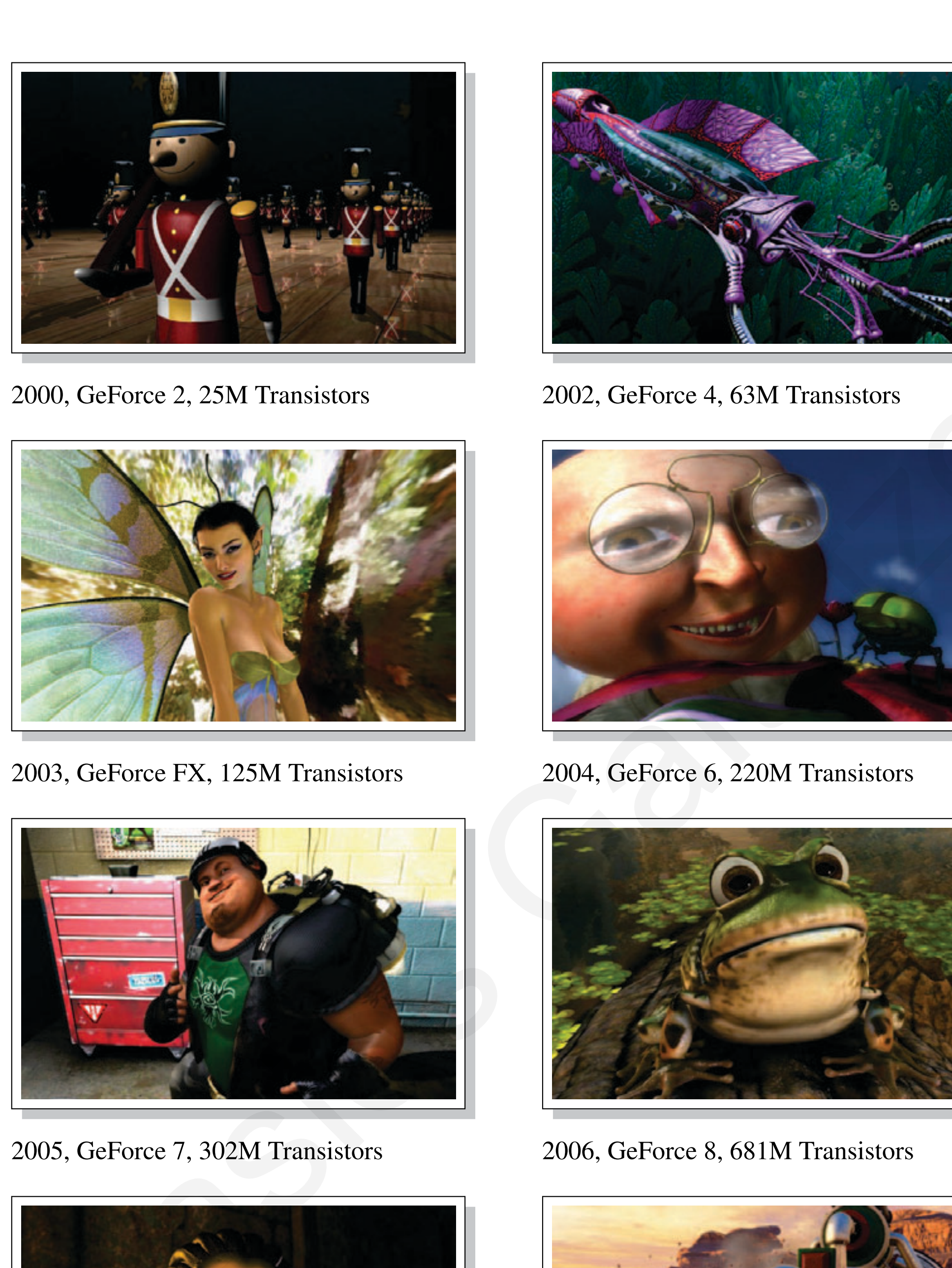

Figure 15: The OpenGL® 4.1 graphics rendering pipeline; the programmable stages are in purple.

In Figure 17, we see the increase on the number of shader processors each graphics card has. Simply put, shader processors are good at executing a set of the same instructions very fast. Thus, scalability is almost guaranteed when a card with more processors is used along with a sensible memory access scheme. In Figure 18 we see the GFLOPS performance chart as well as the Memory Bandwidth chart (Figure 19) for the NVIDIA GPUs compared to the Intel CPUs. It is clear that the Intel processor can not match the graphics chip's parallel processing performance.

### 2.5.1 Vertex Processing Unit

Vertex shaders (VS) are run once for each vertex given to the graphics processor, including adjacent vertices in input primitive topologies with adjacency. The purpose is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen (as well as a depth value for the Z-buffer). Vertex shaders can perform per-vertex operations such as transformations, skinning, morphing and per-vertex lighting but cannot create new vertices. Vertex shaders always operate on a single input vertex and produce a single output vertex. The vertex shader stage must always be active for the pipeline to execute. If no vertex modification or transformation is required, a pass-through vertex shader must be created. The output of the vertex shader goes to the next stage in the pipeline, which is either a geometry shader if present or the rasterizer otherwise.

Athanasios Gaitatzes



2000, GeForce 2, 25M Transistors

2002, GeForce 4, 63M Transistors

2003, GeForce FX, 125M Transistors

2004, GeForce 6, 220M Transistors

2005, GeForce 7, 302M Transistors

2006, GeForce 8, 681M Transistors

2008, GeForce GTX 280, 1.4B Transistors

2010, GeForce GTX 480, 3.0B Transistors

Figure 16: Images from demos showing the capabilities of the hardware which translates in better image quality; courtesy of NVIDIA®

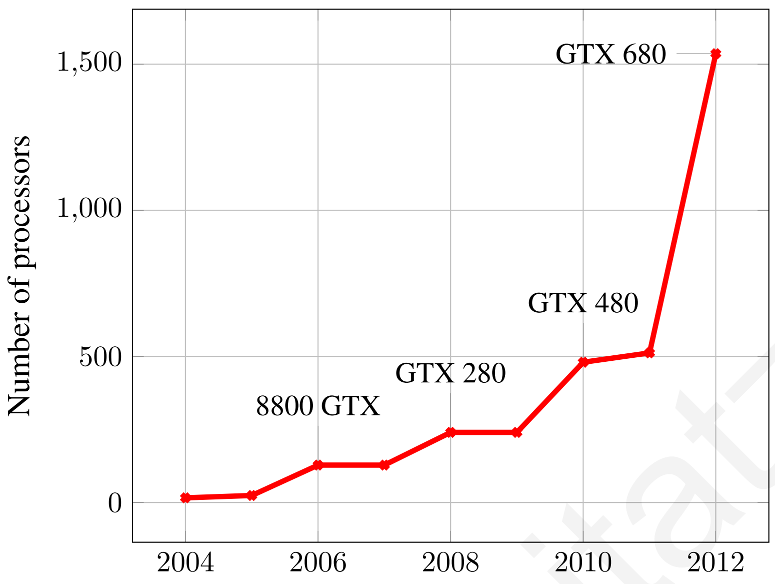



Figure 17: NVIDIA GPU Shader Processors

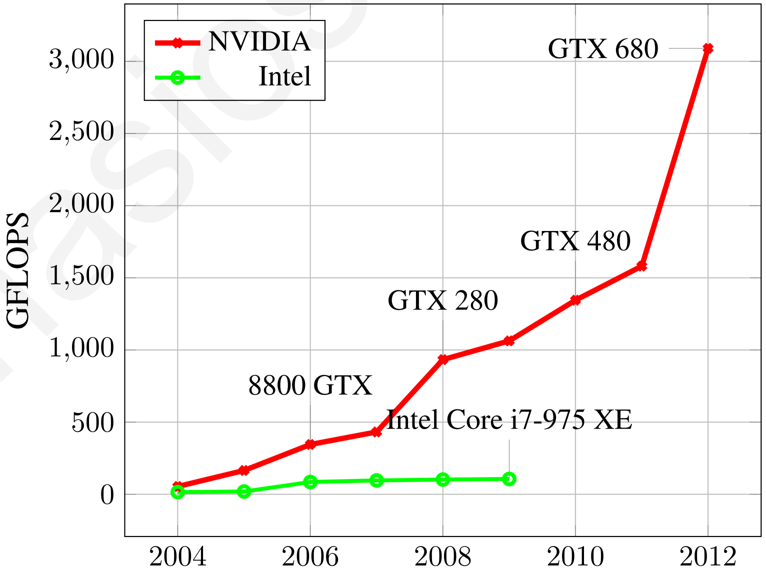



Figure 18: Peak GFLOPS Chart

Athanasios Gaitatzes

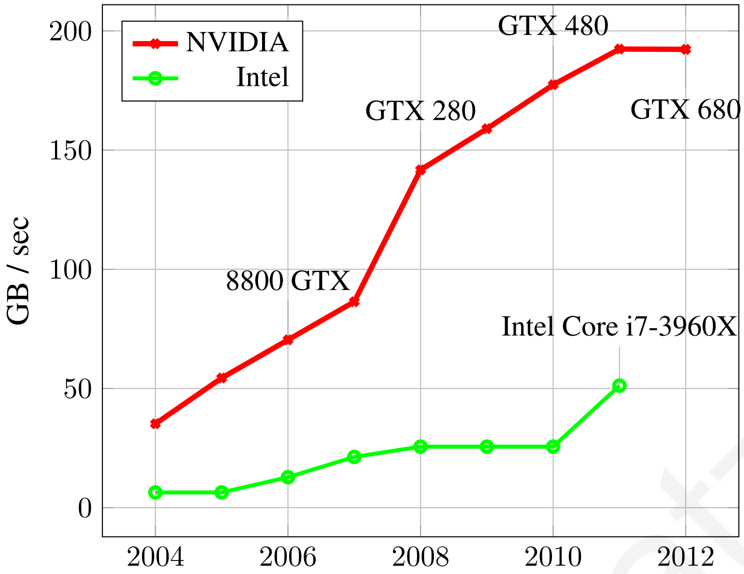



Figure 19: Memory Bandwidth Chart

### 2.5.2 Pixel Processing Unit

The pixel shader (PS), also known as fragment shader, enables rich shading techniques such as per-pixel lighting, post-processing effects and other phenomena. A pixel shader is a program that combines constant variables, texture data, interpolated per-vertex values and other data to produce per-pixel outputs. They can alter the depth of the pixel (for Z-buffering) or output more than one color if multiple render targets (MRT) are active. The rasterizer stage invokes a pixel shader once for each pixel covered by a primitive. Typically, a hardware implementation runs a pixel shader on multiple pixels (for example a 2x2 grid).

A pixel shader alone cannot produce very complex effects, because it operates only on a single pixel, without knowledge of a scene's geometry. Pixel shader input data includes vertex attributes of the primitive being rasterized, that can be interpolated (linearly, or with centroid sampling), or evaluated at pixel shader center locations with or without perspective correction, or can be treated as per-primitive constants.

Athanasios Gaitatzes

### 2.5.3 Geometry Processing Unit

The geometry shader (GS) is an optional shader stage which was introduced in Direct3D 10 and OpenGL® 3.2; formerly available in OpenGL® 2.0+ with the use of extensions. Unlike vertex shaders, which operate on a single vertex, this type of shader operates on a single primitive at a time and emits one or more output primitives, all of the same type, which are then processed like an equivalent primitive specified by the application. The geometry shader takes as input the transformed attributes of all the vertices that belong to the primitive, possibly with adjacency information. The shader can then emit a new set of transformed vertices, arranged into primitives, which are rasterized and their fragments ultimately passed to a pixel shader. The original primitive is discarded after geometry shader execution. Geometry shader programs are executed after vertex shaders. Typical uses of this shader include point sprite generation, geometry tessellation, shadow volume extrusion and single pass rendering to a cube map.

## 2.6 Deferred Shading

Standard rendering in a GPU starts from the input geometry and passes through the whole pipeline in order to generate the final result. We call such a rendering process *forward rendering*. In forward rendering, the shading computation in a pixel shader will be executed for all the pixels from rasterization. However, only the visible pixels will appear in the final result. Hence, the shading computations for the discarded pixels is wasteful. Motivated by this, a *deferred shading* technique has been proposed. Deferred shading postpones shading calculations for a pixel until the visibility of that pixel is completely determined. As such, only pixels that contribute to the final image are shaded.

Athanasios Gaitatzes

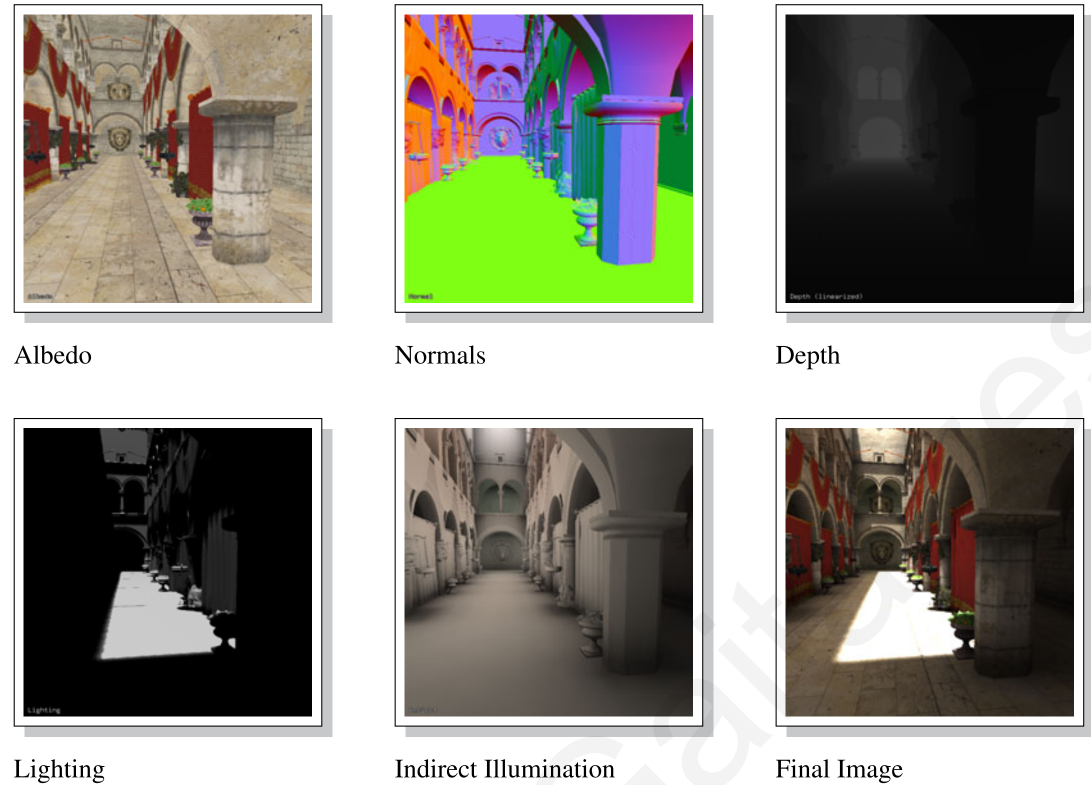



Figure 20: G-buffers used for Deferred Shading.

The usual implementation of deferred shading contains two passes. In the first pass, the scene geometry properties, such as position, normal, material, etc, are rendered into intermediate buffer storage to be combined later. These buffers are called *geometry buffers* (g-buffers), as shown in Figure 20. When generating the g-buffers in modern GPUs, we rely on the *multiple render targets* (MRT) technique to avoid redundant vertex transformations. In the second pass, a simple full-screen quad is rendered to invoke the shading computation in a pixel shader. All the g-buffers can be read by the pixel shader that computes the final shading in the image.

Deferred shading can achieve high performance by saving unnecessary shading computations. It can also provide simpler management of complex lighting resources, ease of managing other complex shader resources and the simplification of the software rendering pipeline. As a result, deferred shading has been widely used in video games. Because of the use of MRTs with floating

Athanasios Gaitatzes

point format, when generating g-buffers the memory bandwidth of deferred shading is higher than that of forward rendering. We refer the interested reader to Policarpo [86] for more details.

Athanasios Gaitatzes

# Chapter 3

## Related Work

There are several ways to approach the problem of simulating the global light in an environment. First, we can pre-compute the approximate lighting in an environment. Second, we can simplify the lighting equation itself thus reducing the computational complexity or simplify our scene geometry or lights in order to reduce the required computations. Finally, we can use brute force methods that in the past were extremely slow but with the hardware of today can parallelize quite easy.

We give emphasis on the related work regarding the lighting equation simplification which is pertinent to our work; namely two areas that both share the computation of the visibility function; the acceleration of stochastic ray-tracing algorithms and the acceleration of the computation of ambient occlusion. We present related work for both non-GPU methods and GPU methods. In addition we present various field computations around an object that are used for accelerating different types of algorithms.

In addition we explore geometry simplification type of algorithms like volume-based global illumination methods and present the founding work on which we base our research. We focus



Athanasios Gaitatzes

on geometry- and image-based voxelization methods that simplify considerably the environment geometry thus speeding-up the computation of global illumination.

Related to Deferred rendering [94], a screen-space, or image-space technique, only takes information into account which can be obtained from a rasterization pass that takes place before computing the global illumination approximation. As such, the information consists of fragment depths, positions, normals, tangent spaces etc. Based only on screen-space data, such techniques then can compute a variety of illumination effects. In most cases, these algorithms do not only consider a single pixel, but also the neighborhood of a pixel to compute its shading and suffer from view-dependent artifacts.

## 3.1 Methods that use Pre-computations

In order to pre-compute the approximate lighting in an environment several schemes can be used. Kavan et al. [59] use vertex baking. During pre-processing they pre-compute the lighting at each vertex and then at runtime interpolate the values of nearby vertices using continuous least-squares.

### 3.1.1 Lightmaps

Similarly, light-maps (first used by John Carmack's Quake engine, see Figure 21) allows lighting information to be pre-calculated and stored into a map. Thus, complex shading calculations are reduced to simple texture lookups. Any Global Illumination method, including complex view-independent lighting models, can be used to generate the maps. After the light-maps have been generated, the texture to be lit and the light-map are blended together when applied to the polygon to produce the final effect. Among the positive aspects of the method are the very good image quality that it produces, since it supports multiple bounces. In addition the method has relatively

Athanasios Gaitatzes

moderate memory budget requirements since the light-maps can be stored at a lower resolution, because view-independent lighting, changes more slowly than texture detail. On the negative side light-maps do not support dynamic lighting conditions or dynamic objects unless the maps are recomputed.

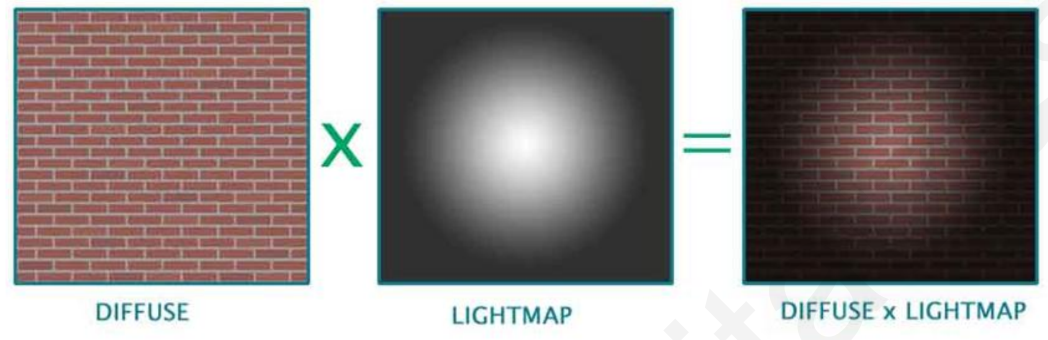



Figure 21: Light mapping example.

### 3.1.2 Precomputed Radiance Transfer

Usually any pre-computation method does not support dynamic lighting conditions or dynamic objects. On the contrary, the precomputed radiance transfer method, introduced by Peter-Pike Sloan et al. [100], supports dynamic distant lighting at the expense of storage and the method of

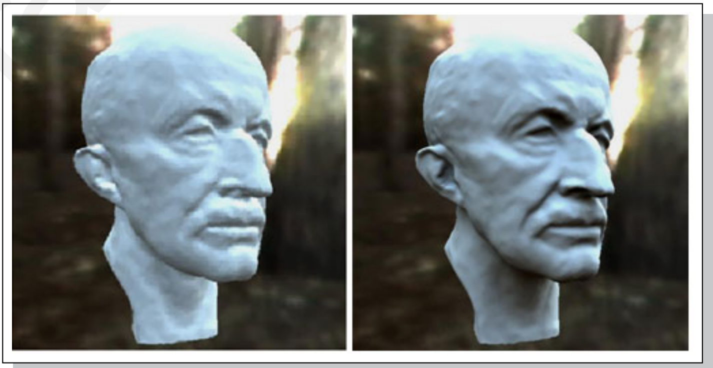


Figure 22: An object under area lighting, without self-shadowing and with self-shadowing. Image courtesy of Peter-Pike Sloan.

Athanasios Gaitatzes

Kristensen et al. [64] which supports local lighting. A preprocessing step is performed over an object's surface to create a set of radiance transfer functions, which represent transfer of arbitrary, low-frequency incident lighting into exiting radiance. These transfer functions can include effects from shadows and inter-reflections from the object onto itself and are stored at each vertex of an object. At runtime, these preprocessed transfer functions are applied to the incident lighting of the scene to produce the final, outgoing radiance at every vertex (see Figure 22). Because both the lighting and the transfer functions are expressed using a set of orthonormal basis functions, the whole shading integral becomes reduced into a dot product of vectors for diffuse surface objects, or into a matrix multiplication and dot product afterwards for glossy surface objects. Among the positive aspects of PRT is that it supports dynamic lighting environments but at the expense of very large auxiliary data requirements. On the negative side it only supports static objects.

### 3.2 Methods that Simplify the Lighting Equation

Assuming that a surface is lit by a uniform lighting environment beyond the influence of the geometry, is one way to simplify the *Rendering equation* that leads to the Ambient Occlusion genre of algorithms.

#### 3.2.1 Ambient Occlusion

Simplifying the *Rendering equation* by assuming that a surface is lit by a uniform lighting environment beyond the influence of the geometry in a hemisphere above point $\mathbf{x}$ leads to:

$$A(\mathbf{x}, \vec{n}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{x}, \vec{\omega}_o)(\vec{\omega}_o \cdot \vec{n}) d\vec{\omega}_o \tag{2}$$

where $V(\mathbf{x}, \vec{\omega}_o)$ is an empirical function that maps distance from surface point $\mathbf{x}$ to the closest surface along direction $\vec{\omega}_o$ to visibility values between 0 (no visibility) and 1.

Athanasios Gaitatzes

Ambient Occlusion (AO) (see Figure 23) was first introduced in 1998 by Zhukov and Iones et al. [124] [48]. AO is the attenuation of ambient light due to the occlusion of nearby geometry i.e. it computes how a surface point is actually exposed to this indirect diffuse light and declares the brightness of a surface point to be functionally dependent on the amount of surrounding geometry occluding its visible hemisphere. It imitates global illumination effects regarding the occlusion of surface points by the surrounding scene-geometry and does not try to simulate the interplay of incident and reflected light. The algorithm, depending on the size of the scene, could run in real-time producing adequate results. For off-line rendering, ambient occlusion is usually pre-computed at each vertex of the model and stored either as vertex information or into a texture. For real-time rendering, recent work by Kontkanen et al. [63] suggests storing ambient occlusion as a field around moving objects and projecting it onto the scene as the object moves. The interactions of multiple dynamically moving rigid objects can be combined in real-time. Zhou et al. [123] approximate the ambient occlusion by computing a field around an object that describes the shadowing effects of the model at points around it. The field is represented by Haar Wavelets or Spherical Harmonics making it more accurate than the method of Kontkanen et al. but also more expensive to calculate.

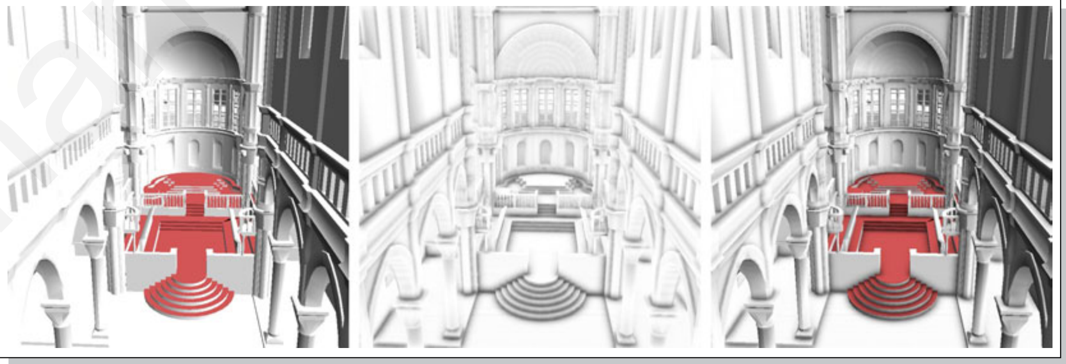


Figure 23: Ambient occlusion example courtesy of Morgan McGuire.

Athanasios Gaitatzes

Malmer et al. [70] surround the object with a regular 3D grid, pre-computing ambient occlusion at the center of each grid cell with high memory costs for moderately complex scenes.

Among the positive aspects of AO is that it is cheaper to compute than Global Illumination, it produces good results and it can be computed in multiple spaces. On the negative side it is still not cheap to compute well and it is not very well suited for high frequency lighting.

#### 3.2.1.1 Ambient Occlusion on the GPU - Screen Space AO

Ambient occlusion (AO) computation on the GPU was first used by Bunnell [6], who approximates the AO by modelling the receiver surface as disk-based occluders (i.e. surface discretization) and evaluates the ambient occlusion caused by the disks using an analytic method. He uses a heuristic method to combine the shadows cast from multiple disks into a noise free image but requires high tessellation of scene geometry and a big pre-computation step.

Screen Space Ambient Occlusion (SSAO) represents a class of algorithms, which compute occlusion in image-space (screen-space). SSAO is always computed after the visibility (z-buffer) test has been done and the actual visible pixels have been determined. This makes SSAO very efficient but also limits the occlusion to surface points that can be seen by the camera. The algorithm is implemented as a pixel shader, analyzing the scene depth buffer which is stored in a texture. For every pixel on the screen, the pixel shader samples the depth values around the current pixel and tries to compute the amount of occlusion from each of the sampled points. In its simplest implementation, the occlusion factor depends only on the depth difference between sampled point and current point.

Shanmugam et al. [97] compute ambient occlusion as a post-processing pass, based on a depth buffer from the eyes point of view. They split the AO computation into two phases, one for high frequency near detail and another phase for low frequency detail with a wider search. The second

Athanasios Gaitatzes

phase allows large objects to inter-occlude as they pass next to each other. Their approach requires no scene-dependent pre-computations. On the downside, over occlusion artifacts might show up when multiple neighboring spheres contribute occlusion to the same pixel.

Mittring [75] does a full screen post-processing pass where z-buffer data is sampled around each pixel and an AO value is computed based on depth differences. Sampling occurs randomly in a sphere around each pixel and AO is proportional to the number of sampled occluders. This view-dependent approach is fast, requires minimal or no pre-calculation, but cannot model AO correctly, because depth discontinuities, such as object edges and buffer boundaries, produce popping effects.

The *Horizon-based AO* by Bavoil et al. [4] uses the surfaces angle of elevation to approximate AO. They compute this angle by summing up the tangent and the horizon angle in view space for a predefined set of screen directions and choose the most representative one. The method is an improved form of SSAO as it produces better quality results but in terms of peformance its costs more than SSAO.

*Ambient Occlusion Volumes* by McGuire [73] compute analytic occlusion per polygon for dynamic geometry. Because some polygons may be double counted, the method approximates the aggregate occlusion using a compensation map.

*Volumetric Obscurance* by Loos et al. [67] improves upon the SSAO technique by making better use of each depth buffer sample; instead of treating them as point samples (with a simple binary comparison between the depth buffer and the sampled depth), each sample is treated as a line sample (taking full account of the difference between the two values). It is similar to a concurrently developed method of *Volumetric Ambient Occlusion* by Szirmay-Kalos et al. [106]. Both techniques can be applied to most SSAO implementations to improve quality or increase performance.

Athanasios Gaitatzes

Among the positive aspects of Screen Space approaches is that it requires no preprocessing or pre-computation and that it easily integrates into existing rendering pipelines. In addition all computations are independent of the scene polygon count thus generating dynamic real-time occlusion effects. Among its negative aspects are the viewpoint dependency, as objects outside the view frustum are not considered and the noisy computed occlusion. In addition the computational effort is dependent on the screen resolution.

#### 3.2.1.2 Field Computations around an Object

The work of Avneesh Sud et al. [103] [104] for computing the discretized 3D Euclidian distance to the surface of a primitive is used for speeding up interactive collision and distance queries types of algorithms. In the work of Huang et al. [46] in a pre-computation stage the object is separated into convex segments each one surrounded by an oriented bounding box (OBB). The OBB is split into cells, each one recording a reference to the primitive that is intersected by a ray through this cell (traversal field). The multiple OBBs are needed in order to allow inter-reflections. Due to the fact that the number of OBBs and their corresponding traversal fields depends on the complexity of the original model, memory consumption may rise significantly.

## 3.3 Methods that Discretize the Scene Geometry

The use of a regular or hierarchical space discretization in global illumination is not new. Several non-real-time algorithms in the past have utilized volume-based acceleration methods and volume data caching to increase their performance. In the past two years, both the porting of global illumination algorithms to the GPU and the inception of new, real-time methods for approximating indirect lighting have gained significant interest from the research community and the game industry.

Athanasios Gaitatzes

Figure 24: Radiosity examples from Cornell University.

In order to avoid using all the tiny triangles in a scene we can simplify the environment geometry or the light representation in order to reduce the number of required computations for simulating Global Illumination. Radiosity and Voxelization are two methods that provide geometric scene simplification.

Radiosity based methods in voxel space have addressed the illumination problem, like Greger et al. [40] and Chatelier et al. [10] but their results were not computed in real-time and had large storage requirements. Modern advances of the same approach, Kaplanyan [54], yielded much faster results than before, but ignored indirect occlusion and secondary light bounces.

### 3.3.1 Radiosity

In 1984 Goral et al. [38] introduced the radiosity algorithm (see Figure 24) as a solution to the Global Illumination problem. Radiosity handles a scene by splitting up geometry into a series of sub-patches. It then uses linear equations to calculate how illumination travels from a light source and across these patches. Radiosity though, just like ray-tracing, only simulated part of the Global Illumination equation and did not account for effects like specular and mirror reflections and transmission. It was therefore often complemented by a ray-tracing method to fill in some of the specular effects.

Athanasios Gaitatzes

Radiosity represents the rate at which energy leaves a surface i.e. the total power emitted in every direction from a surface area. It is therefore ideally suited to model the (uniformly) diffused outgoing light from a small surface area. Radiosity methods create a closed energy system where every polygon emits and/or bounces some light at every other polygon. It calculates how light energy spreads through the system. The solution is found by solving a linear system for radiosity of each "surface", dependent on the emissive properties of the surface and the relation to other surfaces (form factors). The final output is a polygon mesh with pre-calculated colors for each vertex or even light maps (texture atlases, see 3.1.1). Among its positive aspects is the view-independent real-time display after initial calculation and inter-object interaction effects like soft shadows, indirect lighting and color bleeding. Among its negative aspects are the large computational and storage costs and the difficulty to represent non-diffuse light like mirrors and shiny objects as these effects tend to be "blurry" i.e. not sharp without good surface subdivision. In addition, if anything moves in the scene then the form factors need to re-computed.

### 3.3.2 Voxelization

A voxel represents a single sample, or data point, on a regularly-spaced, three dimensional grid. This data point can consist of a single piece of data, such as opacity, or multiple pieces of data, such as surface normal vector and color. Work has been presented on binary voxelization by Eisemann et al. [22] and Forest et al. [27], on solid voxelization by Eisemann et al. [23] and Schwarz et al. [95] and on surface voxelization by Crassin et al. [16] and Schwarz et al. [95] among others.

Athanasios Gaitatzes

#### 3.3.2.1 Geometry-based Surface Voxelization

In the domain of geometry-based voxelization many algorithms with various properties have been devised. Among the most relevant real-time approaches are variations of the XOR slicing method that was first presented by Chen et al. [11] and Fang et al. [25]. The algorithm renders the geometry once for each slice of the volume grid, each time restricting the clipping volume to this slice. It requires watertight models and multiple passes over the geometry, once for each texture slice per sweep axis in order to correctly assign the geometry into voxels.

Dong et al. [20] encode binary voxels in separate bits of multiple multi-channel render targets, allowing to treat many slices in a single rendering pass. A fragment's depth is used to derive the voxel and its bit is set via additive alpha blending. They also consider all three volume axes as sweep directions and each triangle is submitted for rendering in one of the directional passes, according to the dominant direction of its normal. Unfortunately, for dynamic scenes the performance is influenced by the required triangle pre-sorting. Eisemann et al. [22] presented an extension to this approach achieving higher performance. Their method, taking advantage of the same efficient encoding, uses the RGBA-channels of a texture as a binary mask to encode the boundary of the scene geometry. The depth of a fragment is used as an indicator as to which bit in the mask has to be set by using the more robust bitwise or-blending. The resulting representation though, frequently exhibits holes as only one viewing direction is considered in the original implementation.

Forest et al. [27] suggest a hierarchical volumetric representation by offering an extension to Dong et al. [20] method. Furthermore, Zhang et al. [122] proposed to use a conservative rasterization approach to capture more details of the scene geometry. Another method based on slicing was presented by Crane et al. [15]. They used the geometry shader to intersect all triangles of the scene

Athanasios Gaitatzes

with each plane of the three dimensional grid to successively fill each layer. Schwarz et al. [95] directly build a hierarchical volume representation using a GPGPU triangle processing algorithm. It can achieve sparse, high resolution voxelization but it is complex, requires GPGPU architectures and GPU context switching. Pantaleoni [80] presents a programmable pipeline to perform geometry-based triangle voxelization. He also uses a GPGPU triangle processing algorithm and requires *Shader Model 5* GPUs in order to execute.

#### 3.3.2.2 Image-based Surface Voxelization

In addition to the fact that performance of geometry-based voxelization algorithms depends heavily on the size of the geometry, several approaches do not allow for the storage of multi-channel scalar data at the location of each voxel (i.e. Dong et al. [20] and Eisemann et al. [22]). On the other hand, image-based techniques guarantee constant, low running time but sacrifice voxelization quality in terms of completeness of volume set, manifested as partial voxelization or inconsistent sample density.

One of the first depth-buffer-based voxelization method by Karabassi et al. [57], performed a fast volume rasterization of arbitrary geometry but could not voxelize correctly the cavities of objects. Passalis et al. [83] proposed a depth-peeling multi-directional generalization of the above technique, lifting its concavity restrictions. Unfortunately, their algorithm requires a number of depth layers equal to the scene depth complexity in each sweeping direction, rendering it practical mostly for single object voxelization.

Thiedemann et al. [108] introduced an interactive volume-based global illumination method, where the spatial occupancy and color data are generated by injecting a geometry texture atlas containing point samples of the polygonal geometry. The method relies on the generation of geometry images of the objects prior to voxelization and produces fast and view-independent voxelization

Athanasios Gaitatzes

but requires model preprocessing and extra storage for the texture atlas and is sensitive to the point sampling rate and surface deformations.

Mavridis et al. [72] adopted a similar volume population approach, where occupancy and direct illumination of the geometry are injected into the volume as vertices of the tessellated geometry. Supplementary points are generated by injecting the view camera and RSM G-buffers into the volume.

Similarly, Kaplanyan et al. [55] populate their voxelization by injecting the view camera and RSM G-buffers into the volume. In their method geometry significant to the visibility determination, is ignored as it resides outside the view frustum.

Athanasios Gaitatzes

### 3.4 Methods that Discretize the Light Representation

In 1996 Henrik Wann Jensen [50] proposed the photon mapping method (see Figure 25) that decoupled the geometry from the illumination solution which was represented in a spatial data structure called the photon map. This decoupling proved to be quite powerful as the *Rendering equation's* terms could be calculated separately and stored into separate photon maps. It was also the reason that photon mapping was very flexible as parts of the *Rendering equation* could be

Figure 25: Photon mapping examples courtesy of Henrik Wann Jensen.

solved using other techniques. Photon mapping has also been extended to account for participating media effects such as sub-surface scattering and volume caustics.

The algorithm worked in two passes; the first pass would emit from the light sources a user-specified amount of photons into the scene. These photons would bounce for a pre-determined number of times, until they were absorbed or until they exited the scene. Each time a photon (i.e. a ray carrying a predetermined amount of radiant power) intersected a surface, the incident power was recorded and the reflected (attenuated) power was retransmitted to the environment using a stochastic path tracing scheme. The course of each photon was terminated when the photon no longer represented substantial power or when a user-defined termination criterion was met. The spectral attributes of the transmitted photon energy were modulated as the photon interacted with the wavelength-dependent material attributes of the surfaces (e.g. iridescence, color bleeding). The second pass would basically sample each pixel in the scene, with a ray tracer and calculate how much each photon hit contributed to that pixel's color and illumination. Among its positive aspects is that the preprocessing step is view-independent, so it only needs to be re-calculated if the lighting or positions of objects in the scene change. Among its negative aspects are its high computational cost and the re-evaluation of the photon map once changes are made to the scene. In addition, phenomena such as diffraction, interference and polarization still can not be accurately simulated.

McGuire et al. [74] computed the first bounce of the photons using rasterization on the GPU, continues the photon tracing on the CPU for the rest of the bounces and finally scatters the illumination from the photons using the GPU. Since part of the photon tracing still runs on the CPU, a large number of parallel cores are required to achieve interactive frame-rates.

The *Irradiance Volume*, which was first introduced by Greger et al. [40], regards a set of single irradiance samples, parameterized by direction, storing incoming light for a particular point in

Athanasios Gaitatzes

space (i.e. the light that flowed through that point). The method had large storage requirements as neither an environment map nor spherical harmonics were used for the irradiance storage. With a set of samples they approximated the irradiance of a volume, which was generally time-consuming to compute but trivial to access afterwards. With an irradiance volume they efficiently estimated the global illumination of a scene.

The idea of *Instant Radiosity*, introduced by Keller [60], approximates the indirect illumination of a scene using a set of *Virtual Point Lights* (VPLs). VPLs are points in space that act as light sources and encapsulate light reflected off a surface at a given location. A number of photons are traced from the scene light sources into the scene and VPLs are created at surface hit points, then the scene is rendered, as lit by each VPL. The major cost of the original method is the calculation of shadows from a potentially large number of point lights but since it does not require any complex data structures it is a very good candidate for a GPU implementation. Lightcuts by Walter at al. [112] reduce the number of the required shadow queries by clustering the VPLs in groups and using one shadow query per cluster, but the performance is still far from real-time. As instant radiosity algorithms are only used to compute the indirect illumination of a scene they have to be combined with the direct illumination in order to create a high quality lighting environment.

*Reflective Shadow maps* (RSMs), introduced by Dachsbacher et al. [18], is a collection of maps that capture information of surfaces visible from a light source. The RSM is then sampled to choose surfaces that will be used as VPLs. Normal vectors, position and outgoing flux of the lit surface samples are stored in an RSM along with the standard depth image of a conventional shadow map. In the original method, indirect light at a surface point is estimated by projecting it on the RSM of each light source and sampling the VPL data in image-space in a disk centered at the projected coordinate pair. The cumulative contribution of the VPLs is measured, but without taking scene occlusion into account. To achieve interactive frame rates, screen-space interpolation

Athanasios Gaitatzes

is required and the method is limited to the first bounce of indirect illumination. Many variations of the RSM algorithm have been proposed such as the VPL splatting version by the same authors [19] or its multiresolutional extension by Nichols et al. [77]. *Imperfect Shadow Maps* algorithm by Ritschel et al. [91] use a point based representation of the scene to efficiently render extremely rough approximations of the shadow maps for all the VPLs in one pass. They achieve interactive frame rates but indirect shadows are smoothed out considerably by the imperfections and the low resolution of the shadow maps.

Ritchell [92] extends previous methods for screen-space ambient occlusion calculation by Shanmugam et al. [97] and introduces a method to approximate the first indirect diffuse bounce of the light by only using information in the 2D frame buffer. This method has a very low computational cost but the resulting illumination is hardly accurate since it depends on the projection of the (visible only) objects on the screen.

The concept of interpolating indirect illumination from a cache was introduced by Ward et al. [114]. Accurate irradiance estimates are computed using ray-tracing on a few surface points (irradiance sample points) and for the remaining surface points fast interpolation is used. Wang [113] presents a method to calculate the irradiance sample points in advance and implements the algorithm on the GPU. The method is accurate but it achieves interactive frame rates only in very simple scenes.

Nijasure [79] uses spherical harmonics to store the incoming radiance of the scene in a uniform grid structure. The surfaces are rendered by interpolating the radiance from the closest grid points. This method supports multiple bounces and indirect occlusion but it’s very expensive because it requires the complete scene to be rendered in a cube map for the radiance estimation on each grid point.

Athanasios Gaitatzes

In the *Light Propagation Volumes* method (LPV) [54] and its multi-scale extension by Kaplanyan et al. the *Cascaded Light Propagation Volumes* [55] (CLPV), RSM-generated virtual point lights are "injected" into a volume texture. A set of points corresponding to a sparse sampling of the RSM depth image is transformed to world space and finally to volume-clip space of a texture-encoded volume buffer. Rendering of the transformed points results in the rasterization of the VPLs in the appropriate locations in this volume buffer. Subsequently, the method uses an iterative propagation scheme to transfer energy from voxel to voxel, taking into account occlusion caused by blocking voxels (CLPV). Blocking voxels are marked by storing any available depth information from the RSMs as well as from the camera depth map into a separate occlusion (geometry) volume. The method achieves high performance for a relatively small number of propagation iterations with respect to the volume size, but suffers from popping artifacts due to view-dependent occlusion information.

Thiedemann et al. [108] using the injection method mentioned above, propose an optimized ray marching scheme for intersecting the gathering rays with the volume data. At each hit voxel, the RSMs are looked up to determine its visibility from the source and subsequently its normal and the incident light. The method can be used to perform near-field single-bounce indirect light estimation and VPL-based computation of indirect illumination.

Mavridis et al. [72] after the volume population (as mentioned before), use ray marching at a voxel level to simulate global illumination with multiple bounces at a cost proportional to the volume size and the sampling distance.

Athanasios Gaitatzes

## 3.5 Brute Force Methods

As a brute-force method, ray-tracing has been too slow to consider for real-time applications. As the compute power of hardware increases and programmable architectures are introduced, new algorithms are being developed in order to bring ray-tracing to real-time.

### 3.5.1 Ray-Tracing

In 1980 Whitted [115] described a ray-tracing algorithm (see Figure 26) which facilitated for shadows, reflections and refractions. Ray tracing works by shooting a ray from the viewer's eye into the scene, where it either leaves that scene or hits another object. If a ray is intercepted by an object, it generates up to three new types of rays: reflection, refraction and shadow. A reflected ray continues on in the mirror-reflection direction from the surface. It is then intersected with objects in the scene; the closest object it intersects is what will be seen in the reflection. Refraction rays traveling through transparent material work similarly, with the addition that a refractive ray could be entering or exiting a material. To further avoid tracing all rays in a scene, a shadow ray is used to test if a surface is visible to a light. If a ray hits a surface at some point which faces a light, a ray is traced between this intersection point and the light. If any opaque object is found in between the surface and the light, the surface is in shadow and so the light does not contribute to its shade.

Ray-tracing can accurately represent specular global highlights in a scene by tracing light rays from the eye to the pixel and into the scene. Rays recursively bounce off objects in the scene and accumulate a color for a specific pixel. Among its positive aspects are the inter-object light interactions like shadows, reflections and refractions (light through glass, etc.). But the lighting effects tend to be abnormally sharp, without soft edges, unless more advanced (sampling) techniques are used.

Athanasios Gaitatzes

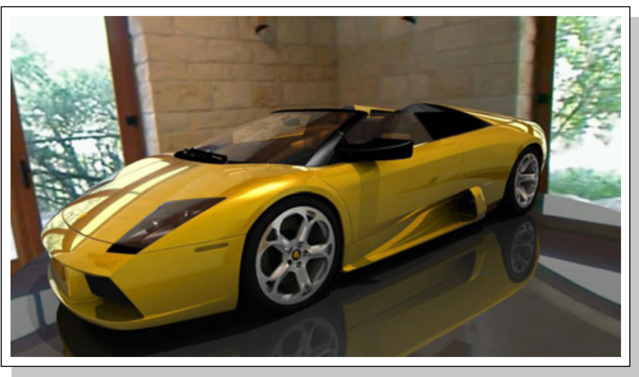


Figure 26: Ray-tracing example.

Despite all the benefits of ray-tracing, it is still far from flawless. One problem associated with ray-tracing and computer graphics in general, is aliasing. This is due to the fact ray-tracing samples a scene at regularly spaced intervals and ignores everything in between. The problem with sampling is that each pixel of the display represents one single light ray. This creates aliasing and unnaturally sharp images. The solution is to send multiple rays through each "pixel" and average the returned colors together. Several methods exist to this affect like "direct super-sampling" where each pixel is split into a grid and rays are send through each grid point, "adaptive super-sampling" (by Whitted [115]) where each pixel is split only if it's significantly different from its neighbors and "jittering" or "stochastic sampling" (by Cook [14]) where rays are send through randomly selected points within the pixel and the extremely jarring effect of aliasing is reduced to noise, which is easily tolerated by the human eye.

Despite adding a whole new repertoire of effects, stochastic sampling does not fix ray-tracing's most glaring drawback as a solution to Global Illumination. Ray tracing only solved part of the problem, in that it only simulates direct illumination and backward specular transmission (i.e. reflection and refraction from the camera) and completely ignores scattering, diffusion and light convergence (caustics).

Athanasios Gaitatzes

Traditionally ray-scene intersection is accelerated through the use of hierarchical data structures. Bounding Volume Hierarchies [37] [93] [13], Voxel Grids [101] [28], Hierarchical Grids [62] [9] [51], Octrees [36], Binary Space Partitioning Trees [105], kd-Trees [44] [76] [68] [3] [49] are just a few.

Recently, a new set of algorithms have been developed for interactive ray-tracing and ray-tracing of dynamic scenes. The work of Wald et al. [110] and [111] demonstrates real-time ray-tracing for small scenes using in-expensive off-the-shelf PCs with SIMD floating point extensions and cluster architectures. Parker et al. [82] demonstrates real-time ray-tracing for larger scenes on shared memory multiprocessor machines. The main issue of these algorithms that accelerate spatially coherent rays, is that their speedup on secondary ray intersection tests is limited.

### 3.5.2 Real-time Ray-Tracing on the GPU

Most GPU ray-tracing methods accelerate already established mechanisms for limiting the number of intersection tests.

Carr et al. [7], Purcell et al. [89], [88], Karlsson et al. [58] and Christen et al. [12] implemented a streaming ray-triangle kernel on the GPU, fed by buckets of coherent rays and proximate geometry organized by a CPU process. However, there was a frequent communication of results from the GPU to the CPU over a narrow bus, negating much of the performance gained from the GPU kernel. Streaming geometry to the GPU became quickly the bottleneck.

To improve the performance of the GPU ray-tracing, different acceleration structures have been widely adopted, such as the incorporation of kd-trees by Havran [43] and Ernst et al. [24]. However, these approaches had limited performance; by far not reaching the frame rates of the CPU based ray tracers. The main problem was the limited GPU architecture. Only small kernels

Athanasios Gaitatzes

without branching were supported. In addition a stack was usually required, which was poorly supported on GPUs. Foley et al. [26] presented two implementations of a stack-less kd-tree traversal algorithm for the GPU, namely kd-restart by Kaplan [53] and kd-backtrack. Foley showed, that on graphics hardware, there are scenes for which a kd-tree yields far better performance than a uniform grid. Although better suited for the GPU, the high number of redundant traversal steps led to relative low performance.

Besides grids and kd-trees there are also several other approaches that use a BVH as an acceleration structure on the GPU. Carr et al. [8] implemented a limited ray tracer on the GPU that was based on geometry images but it required careful parameterization of the geometry. It could only support a single triangle mesh without sharp edges. The acceleration structure was a predefined bounding volume hierarchy which could not adapt to the topology of the object. To alleviate the need for a stack Thrane et al. [109] presented stack-less traversal algorithms for a BVH. They conclude that on the GPU, the bounding volume hierarchy traversal method is up to 9 times faster than that of a uniform grid and a kd-tree. Also, the technique proves the simplest to implement and the most memory efficient.

Horn et al. [45] reduced the number of redundant traversal steps of kd-restart by adding a short stack. With their implementation on modern GPU hardware they achieved a high performance of 1518M rays/s for moderately complex scenes. At the same time, Popov et al. [87] presented a parallel, stack-less kd-tree traversal algorithm without the redundant traversal steps of kd-restart but with a poor GPU utilization of below 33%. With over 16M rays/s, their GPU ray tracer achieved similar performance as CPU based ray tracers. However, both GPU ray-tracing implementations demonstrated only medium-sized, static scenes. Gunther et al. [41] presented a BVH based GPU ray-tracing method for large models achieving close to real-time rates using hard shadows.

Athanasios Gaitatzes

Athanasios Gaitatzes

# Part II

**– Discretization of Visibility –**

**Ambient Occlusion and Secondary Light Bounces**

Athanasios Gaitatzes

Athanasios Gaitatzes

# Chapter 4

## Fast approximate Visibility using pre-computed 4D Visibility Fields

### 4.1 Motivation

One of the most intensive parts for the calculation of the *Rendering equation* (see Equation 1) is the computation of the visibility term. Ray based solutions to the rendering problem have been popular for over two decades. An enormous amount of work has been done by researchers in order to accelerate the tracing of rays, especially through the use of spatial acceleration structures and is still a very active field of research. However, such methods typically have a non-constant cost for ray-intersections. Ambient occlusion computation and real-time ray tracing are just two of the fields where the fast computation of the visibility queries is very important.

### 4.2 Overview

We will create a discretization of the visibility function by clustering together visibility rays thus accelerating the computation of visibility in dynamic scenes. This approximation is directly applicable to secondary diffuse illumination (i.e. ambient occlusion) and ray tracing calculations where exact ray hits are not critical (i.e. soft shadow rays). The main idea of the technique is to



Athanasios Gaitatzes

pre-compute for each object in the scene its associated four-dimensional field that describes the visibility in each direction for all positional samples on a sphere around the object, we call this a displacement field. We are able to speed up the calculation of algorithms that trace visibility rays to near real time frame rates. The storage requirements of the technique, amounts from one byte to one bit per ray direction making it particularly attractive to scenes with multiple instances of the same object, as the same cached data can be reused, regardless of the geometric transformation applied to each instance.

### 4.3 Introduction

Ambient occlusion is defined as the attenuation of ambient light due to the occlusion of nearby geometry. It gives perceptual clues of depth, curvature and spatial proximity and thus is important for realistic rendering. It is a technique that approximates the effect of indirect global illumination and does not yet try to simulate the interplay of incident and reflected light. In Ambient occlusion the indirect component can be computed as:

$$A(\mathbf{x}, \vec{n}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{x}, \vec{\omega}_o)(\vec{\omega}_o \cdot \vec{n}) d\vec{\omega}_o \tag{3}$$

where $V(\mathbf{x}, \vec{\omega}_o)$ is an empirical function that maps distance from surface point $\mathbf{x}$ to the closest surface along direction $\vec{\omega}_o$ to visibility values between 0 (no visibility) and 1. By tracing rays outward from a given surface point $\mathbf{x}$ over the hemisphere around the normal $\vec{n}$, ambient occlusion measures the amount that a point is obscured from light. This average occlusion factor is used to simulate soft-shadowing.

Ray tracing is a general and versatile algorithm that performs image synthesis by shooting rays through each pixel, finding the closest intersection with the scene geometric entities. The generic

Athanasios Gaitatzes

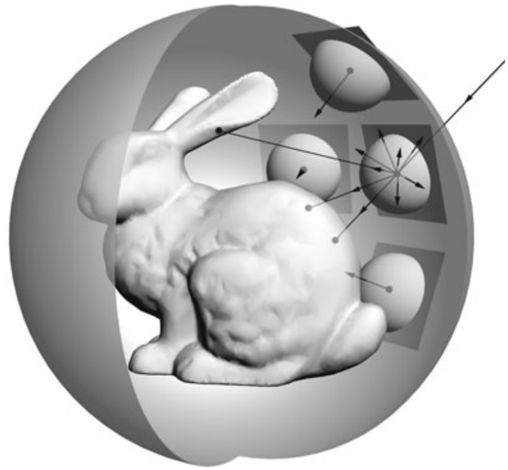


Figure 27: A hemisphere of rays emanating from the bounding sphere towards the object is precomputed for a large number of sample points on the sphere.

backwards ray-tracing algorithm is capable of capturing both local illumination and basic indirect specular effects such as mirror-like reflections and refraction.

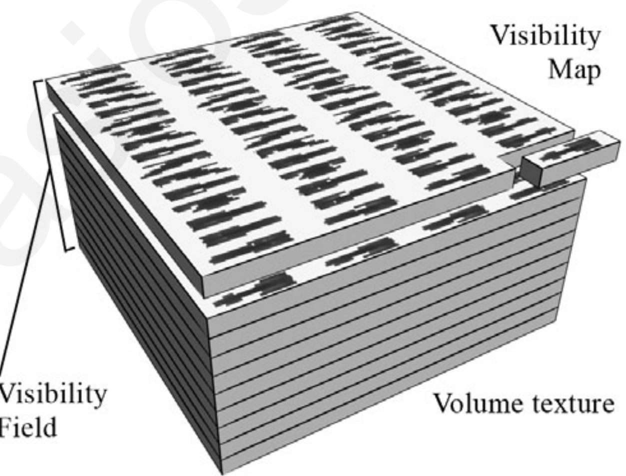



Figure 28: Volume texture of a *visibility field*. Row by row each map is placed into a slice of the volume texture thus minimizing the volume space requirements. As a result a $512^3$ volume will hold four $256^2$ maps per slice.

Athanasios Gaitatzes

We propose a method for the discrete representation, pre-calculation and storage of distance measurements on rigid objects. The encoded distances can be queried at run time to efficiently calculate accurate ambient occlusion and soft shadowing effects. It accelerates the ray-object intersection test and in turn the computations of the visibility function of the lighting equation, by separating the task in two subtasks. First, at preprocessing time, we construct on the CPU the *visibility field* (Figure 27). It stores the intersection distances of a hemisphere of rays originating from sample points on the bounding sphere of an object and directed towards the model itself. We construct one map for each sample point (Algorithm 1). Then, at run time, when a ray from the environment towards an object (or vise versa) intersects its bounding sphere, we perform a simple ray-sphere intersection test and recover from the pre-computed maps the rest of the ray distance for the ray-object intersection test.

In the GPU implementation version, after the construction of the visibility maps, we compactly fit them in one volume texture for easy access on the GPU (Figure 28). In addition, all mesh information (i.e. coordinates, normals and materials) are stored in maps and passed on to the GPU. Taking advantage of the shader units parallelism we demonstrate significant performance gains.

The advantage of the method is that the bulk of the computation is moved to a preprocessing stage. The results are stored in compact gray-scale textures; 1 byte per ray direction for the computation of ambient occlusion and soft shadows and 4 bytes per ray direction for the computation of reflection in ray-tracing, providing for each object a constant size of additional information, independent of the complexity of the original model. Then the real-time algorithm performs a simple intersection test with the bounding sphere of the object and a constant-time map lookup (see Section 4.4.2).

Athanasios Gaitatzes

**Algorithm 1:** Pseudo-code for *Visibility Fields* computation at preprocessing time.

```
generate bounding sphere sample points;
generate samples of hemisphere of rays;
for all bounding sphere sample points (u, v) do
    align hemisphere of rays to normal at (u, v);
    for all rays (θ, φ) do
        if ray intersects the object then
            normalize the distance; // i.e. divide by 2 * R
            record distance in visibility map;
        else
            record distance in visibility map as 2 * R;
```

We show that, in applications such as ambient occlusion, maps that use 1-byte of storage per ray give almost the same result as maps that use 4-bytes of storage space. If the model changes level of detail the same maps can still be used. In addition, the visibility maps contain information that is transformation invariant. As such, no additional information has to be computed when the rigid object is geometrically transformed in the environment. For dynamic scenes with rigidly moving objects, *visibility fields* accelerate the computation of the approximation of the indirect lighting term of the *Rendering equation* to real-time frame rates as well as the computation of soft shadows and reflection in ray-tracing. The performance of this approach does not depend on the polygon count to a large extent; instead, it is directly related to the number of visible pixels shaded by the GPU. This is a significant advantage over existing approaches. In addition, our acceleration structure is flat by nature and thus more suited to the GPU architecture.

### 4.4 Overview of the Visibility Fields

In this section we describe the general idea of *visibility fields*, while in the next section we show their application for ambient occlusion.

Our method bears some similarity to the parameterization of Huang et al. [46] where each ray was described as a vector of the parametric incident location (u, v) on the bounding volume and its

Athanasios Gaitatzes

**Algorithm 2:** Pseudo-code for Ambient Occlusion rendering using *Visibility Fields* during real-time processing.

```
generate hemisphere of ray samples;
for each occlusion receiver object do
    for all points x on the occlusion receiver surface do
        for all emanating rays do
            if ray intersects bounding sphere of occluder object then
                discretize intersection point (u, v);
                discretize ray (θ, φ);
                access distance in visibility map;
            use distance for occlusion approximation;
        compute occlusion at x;
```

corresponding incoming direction $(\theta, \phi)$. However, we introduce our *visibility field* encoding pre-computation where using a similar parameterization, we store the distance from the entry point on the bounding volume to the surface of the object. We further discuss the sampling techniques used and the storage requirements of our method along with the compression scheme.

### 4.4.1 Visibility Field Computation

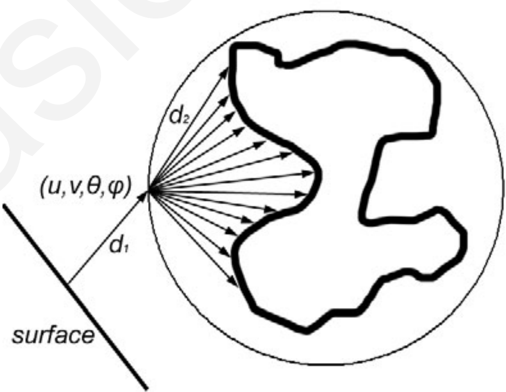



Figure 29: The distance from the surface to the object is split into distance $d_1$ (i.e. surface to object bounding-sphere ray intersection) and distance $d_2$ (bounding-sphere to object ray intersection, retrieved from the appropriate *visibility map*).

Athanasios Gaitatzes

The main idea of encoding *visibility fields* into maps is as follows (Algorithm 1). Consider a rigid object possibly moving through a scene. At a preprocessing step, from a discreet set of sample points on the bounding sphere, described as spherical coordinates (u, v), a hemisphere of rays is cast around the inward normal direction (Figure 29). For each ray $(u, v, \theta, \phi)$, the closest distance between the bounding volume and the model surface is found and recorded as a compact integer value after being normalized by twice the sphere radius. Thus, for each sample point (u, v) a visibility gray-scale map is obtained (Figure 30) that represents the distance traveled along the ray in the direction $(\theta, \phi)$ before hitting the model surface. We define the *visibility field* of the object to be the collection of all visibility maps generated from all sample points on the bounding sphere of the object.

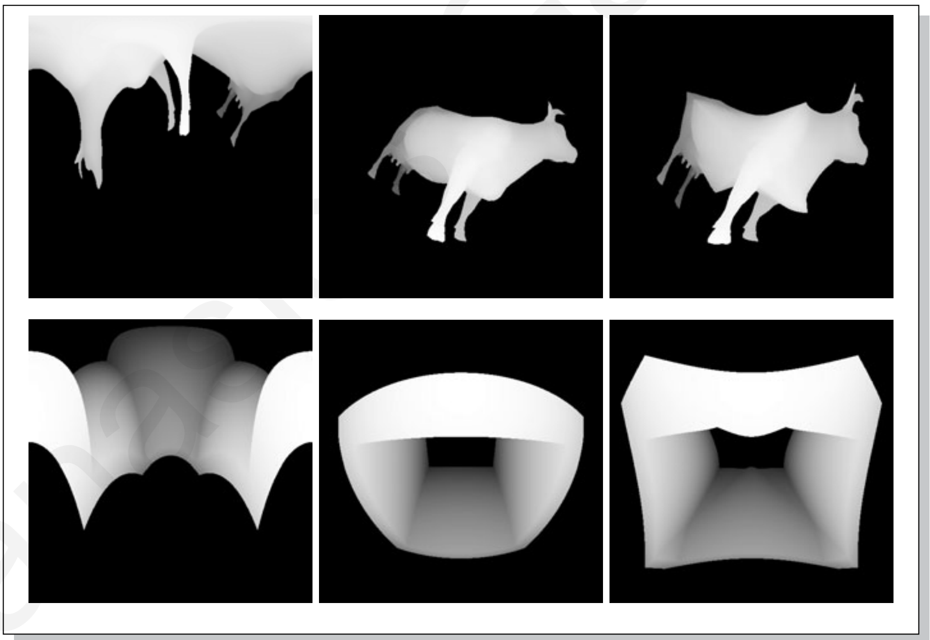


Figure 30: 512x512 visibility maps of a model of a cow (top) and a cube with a hole in it (bottom). Using uniform Sampling of rays (left), Rejection Sampling (middle), Concentric Map Sampling (right). Different $(\theta, \phi)$ to $(s, t)$ mappings, produce different visibility maps.

Athanasios Gaitatzes

### 4.4.2 Visibility Field Indexing

During the real-time part of the execution (Algorithm 2) an incident ray to the object, intersects its bounding sphere and the distance between the ray origin and the intersection point is recorded. The intersection point q is transformed into the object coordinate system: $\mathbf{q}' = \mathbf{M}^{-1} \cdot \mathbf{q}$, where M is the transformation matrix with respect to the reference frame of the ray. Depending on the sampling on the surface of the sphere (see Section 4.4.3), the inverse function is applied to $\mathbf{q}'$ in order to get the closest corresponding point $(u, v)$ on the sphere for which we have a visibility map and therefore the index of the corresponding visibility map. Next we need to find the corresponding $(\theta, \phi)$ of the incident ray. Depending on the ray sampling method (see Section 4.4.4), the appropriate inverse function is applied to the ray, thus recovering the $(\theta, \phi)$ values of the ray. We can now index into the *visibility field* for the given ray $(u, v, \theta, \phi)$ and extract the distance information which is then added to the intersection distance above and this is our approximated distance value of the ray origin from the object's surface.

A special case arises when the rays originate from the object being queried for visibility. As we can see in Figure 31, when a ray originates on the object at point $p_0$, the distance $d_1$ in direction $p_0\overrightarrow{p_1}$ is computed and compared to distance $d_2$ in direction $p_1\overrightarrow{p_0}$ which is extracted from the visibility map at point $p_1$. If $d_1$ is greater than $d_2$ then point $p_0$ is occluded.

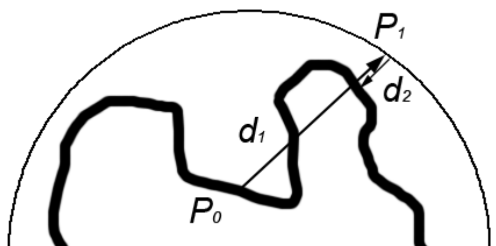



Figure 31: Diagram of visibility computation for intra-object occlusion.

Athanasios Gaitatzes

### 4.4.3 Selecting Samples around the Object

We need to sample entry points on the surface of the bounding volume of the object from where the rays originate in order to generate the visibility maps. The method selected must also have a quick inverse function that can convert an intersection point into the nearest sample. In addition it should distribute the samples over the bounding volume as evenly as possible.

A fairly straightforward choice are the spherical coordinates which have a fairly easy to compute inverse function. However, the samples in this method are concentrated more towards the poles of the sphere.

A common bounding shape that is used to sample the contained geometry is a axis-aligned bounding box (AABB). During the real-time simulation we would perform fast ray-box intersections. Special care though is needed as the AABBs are not transformation invariant and their oriented bounding boxes (OBB) counterparts require more operations.

As most sampling methods deal with sampling over a sphere, if the same methods were used to sample over a cube there would be a high concentration of samples near the vertices of the cube.

We opted for Slater's [99] method, which generates uniformly distributed points on a hemisphere using the triangle subdivision method. The same can be used to cover the full sphere as well. At the same time he suggests a constant time inverse function so, when an environment ray intersects the bounding sphere of the object, we can immediately associate this intersection point with one of the pre-generated visibility maps, in order to retrieve the angle and distance information.

### 4.4.4 Sampling a Hemisphere of Directions

There are several methods that deal with the uniform sampling of rays distributed over a hemisphere. The method selected must be able to uniquely discretize its samples so that they can be

Athanasios Gaitatzes

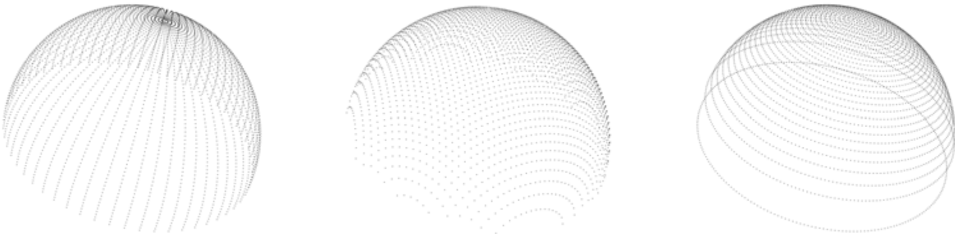


Figure 32: Sampling a hemisphere of rays. (a) Polar Mapping of rays, (b) Rejection Sampling, (c) Concentric Map Sampling.

stored in the visibility maps. In addition there must exist an inverse function that converts the visibility map entries back into sample space.

One method is to use spherical coordinates where a direction in the hemisphere is given by two angles $(\theta, \phi)$. But as can be seen in Figure 32a the rays generated are concentrated towards the cap of the hemisphere producing a good cosine term (close to 1.0) but they are not equally spaced.

In the rejection sampling method (Figure 32b) uniformly distributed points are selected inside a unit disk by selecting points inside the $[\;\;1, 1]^2$ square and rejecting the points that fall outside the unit disk. Using Malley's method [69] the samples are projected on the disk up to the hemisphere above it, producing a cosine distribution of rays. Using this method, about 21.5% of the samples are rejected and so the corresponding space in the visibility map remains unused.

Shirley et al. [98] suggest a concentric map (Figure 32c) sampling method that maps samples in the square $[\;\;1, 1]^2$ to the unit disk $\{(x, y) \mid x^2 + y^2 \leq 1\}$ by mapping concentric squares to concentric circles. The map preserves fractional area, it is bi-continuous and has low distortion. Combined with Malley's method where samples on the unit hemisphere have density proportional to the cosine term, it provides the best solution.

Athanasios Gaitatzes

## 4.5 Visibility Fields on the GPU

### 4.5.1 Ambient Occlusion

Directional ray samples on a reference hemisphere aligned with the z-axis are pre-computed and stored in a texture for passing to the GPU. In the fragment shader (Algorithm 3), the pre-computed ray directions are transformed according to the local normal vector and intersected with the bounding sphere of each occluder. We are able to handle both rays originating outside and inside the bounding sphere for inter-object and intra-object occlusion respectively. The only difference in the computation is the respective step to compute the final ray-object intersection distance at line 3 of Algorithm 3.

The indexing of the *visibility fields* is executed entirely on the GPU as is the Monte Carlo ray casting to evaluate the resulting ambient occlusion. The visibility maps are compacted and stored into a single 3D texture as slices, as shown in Figure 28. As the number of positional samples (i.e. visibility maps) can exceed the maximum volume texture dimension supported by the hardware, we compact as many visibility maps on each 2D slice of the volume as the texture hardware permits.

**Algorithm 3:** Fragment shader pseudo-code for Ambient Occlusion rendering, using *Visibility Fields*.

```
for all emanating rays do
    if ray intersects bounding sphere of occluder object then
        discretize intersection point (u, v);
        discretize ray (θ, φ);
        access distance in visibility field volume;
    use distance for occlusion approximation;
compute occlusion at pixel x;
```

Athanasios Gaitatzes

### 4.5.2 Ray tracing

For our proof-of-concept case study, we wanted to further improve ray-tracing timings of an already fast ray tracer. We used the method of Amit Ben-David et al. [2] that implemented both a CPU and a fast GPU ray tracer by exploiting a BVH acceleration structure that has been proven to work better in some cases [41] and is better suited for dynamic scenes. We did not replace the primary ray intersection tests because the regularity of the ray distribution emphasized the sampling pattern on the bounding sphere. Furthermore GPU rasterization provides better timings for the primary rays pass. In conjunction with the fact that for complex (and therefore time consuming) scenes with elaborate materials, most time is spend on secondary rays, we applied the *visibility fields* method only to secondary rays, including shadow rays. To capture the intricate reflection effects of non-perfect reflection surfaces and to highlight the advantage of our method when intersection tests increase significantly, we extended the implementation to stochastic ray-tracing.

As in the case of the ambient occlusion computation, the rays are stored in a 2D map but this time are re-computed for each running pass. For the ray-object intersection the visibility maps are used in a fragment shader on the GPU (similar to Section 4.5.1) along with the additional pre-computed maps of normals. The generated fragments correspond to intersection test results and the fragment shader returns the intersection point and distance to the actual surface as extracted from the *visibility field*. These results are used for shading or for spawning secondary rays for the next ray-tracing iteration.

Athanasios Gaitatzes

### 4.6 Implementation & Evaluation on the CPU

We have implemented the *visibility fields* algorithm on an Intel Pentium 4 desktop PC running at 3.4GHz with 1GB RAM and an NVIDIA® Quadro FX5500 GPU with 1GB video memory (machine type 1) and an Intel dual Xeon running at 3.0GHz with 4GB RAM and the same graphics board (machine type 2).

The implementation does not utilize the GPU for the indexing calculations. The method is a generic ray casting implementation, used in this case for ambient occlusion and as such can not be compared with other specialized GPU implementations.

#### 4.6.1 Ambient Occlusion

##### 4.6.1.1 Storage and Error Considerations

A 256x256 map stores the distance to the object for 65536 ray directions emanating from one sample. If that map was to store the values as floats it would require 262144 bytes of storage space while storing them as unsigned chars it would require 65536 bytes. In addition, if lossless

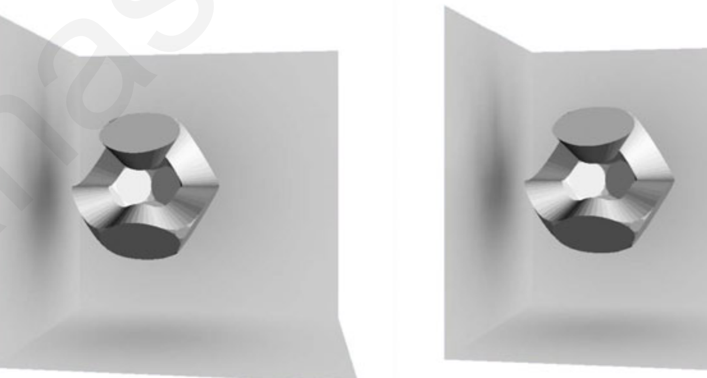


Figure 33: 4 bytes per ray for storage (left), 1 byte per ray for storage (right). There are no obvious visible differences, when the *visibility fields* are used for ambient occlusion.

Athanasios Gaitatzes

compression is used (e.g. run length encoding) then on average less storage would be required. In application areas where integral calculations are performed over the samples or accuracy is not imperative, lossy compression could be used to further reduce the storage requirements. Given that it is essential to keep the storage requirements to a minimum we opted to use unsigned chars for storage as it was found in ambient occlusion approximations that there was little to no gain in visual quality from using floats as can be seen in Figure 33. If higher accuracy is desired one can consider storing more bytes per sample.

#### 4.6.1.2 Using the 8-bit maps

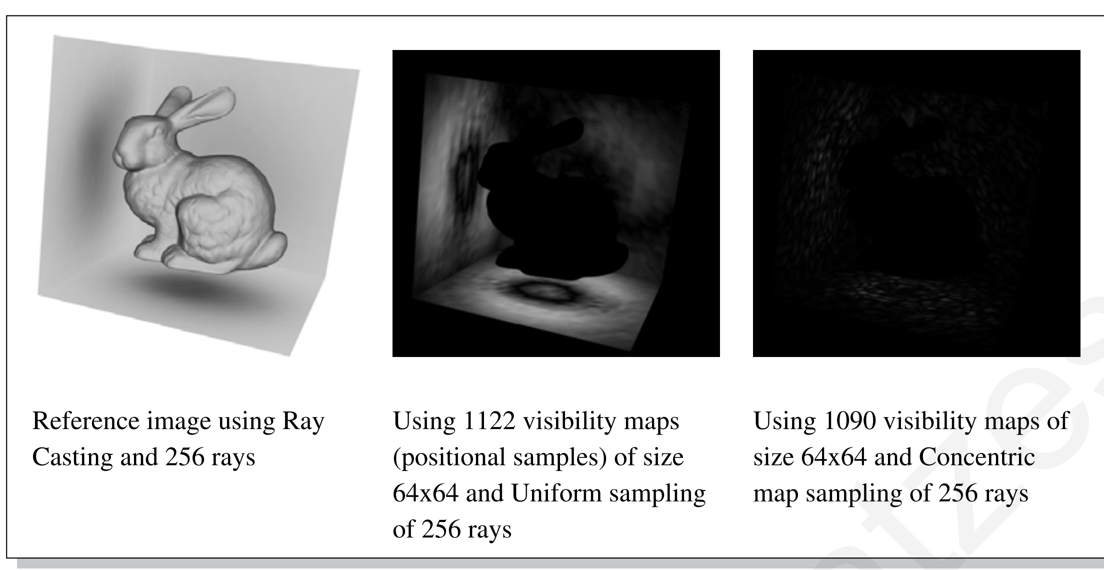
In our examples we used a large number of cast rays per vertex (256) and achieved interactive results that would otherwise be impossible (Figure 1). The complexity of the *visibility field* method is O $(N_r)$ where $N_r$ is the number of rays.

We have run several experiments in order to evaluate the method and establish which of the sampling method is the preferred.

As we can see in Figure 34 choosing a concentric map sampling distribution for the rays produces much better results than the uniform sampling of rays. As such, we opted to use this method in producing the rest of the results.

In Figure 35 we see images of the ambient occlusion solution produced using 4 different resolutions for the concentric map sampling for the ray directions and 3 resolutions for the positional samples on the bounding sphere around the object. Here, as expected, we see that cost of computing the *visibility field* is a function of the sampling resolutions while the cost of using it for ambient occlusion is almost independent of the object complexity. By looking at the root mean square (RMS) error, we observe that it drops quickly as we increase the positional samples. In addition, we observe that as we increase the directional samples, the error does not decrease

Athanasios Gaitatzes



Reference image using Ray Casting and 256 rays

Using 1122 visibility maps (positional samples) of size 64x64 and Uniform sampling of 256 rays

Using 1090 visibility maps of size 64x64 and Concentric map sampling of 256 rays

Figure 34: The image differences between the Reference image and the visibility map methods, show that using Concentric map sampling produces much better quality results as compared to Uniform sampling. Image differences are exaggerated by a factor of 5.

significantly. So a good compromise between memory use and accuracy would be to use the 4226 / 32x32 maps.

In Figure 1, we see the method applied to different types of models. We observe that the cost of using the visibility maps increases very little as we go to higher complexity models. The only exception is the multiple model case, where we have inter-object interactions. The bunny is a caster, the corner is a receiver and the other two objects are both casters and receivers. So the results are justified by the increase of rays cast by about 30 times.

#### 4.6.1.3 Further Memory Optimization

When objects are further away from the viewer, the approximate ambient occlusion calculated previously, can be further optimized in terms of texture space required. Instead of storing into the map the distance between the bounding sphere and the object, we can store only the visibility

Athanasios Gaitatzes

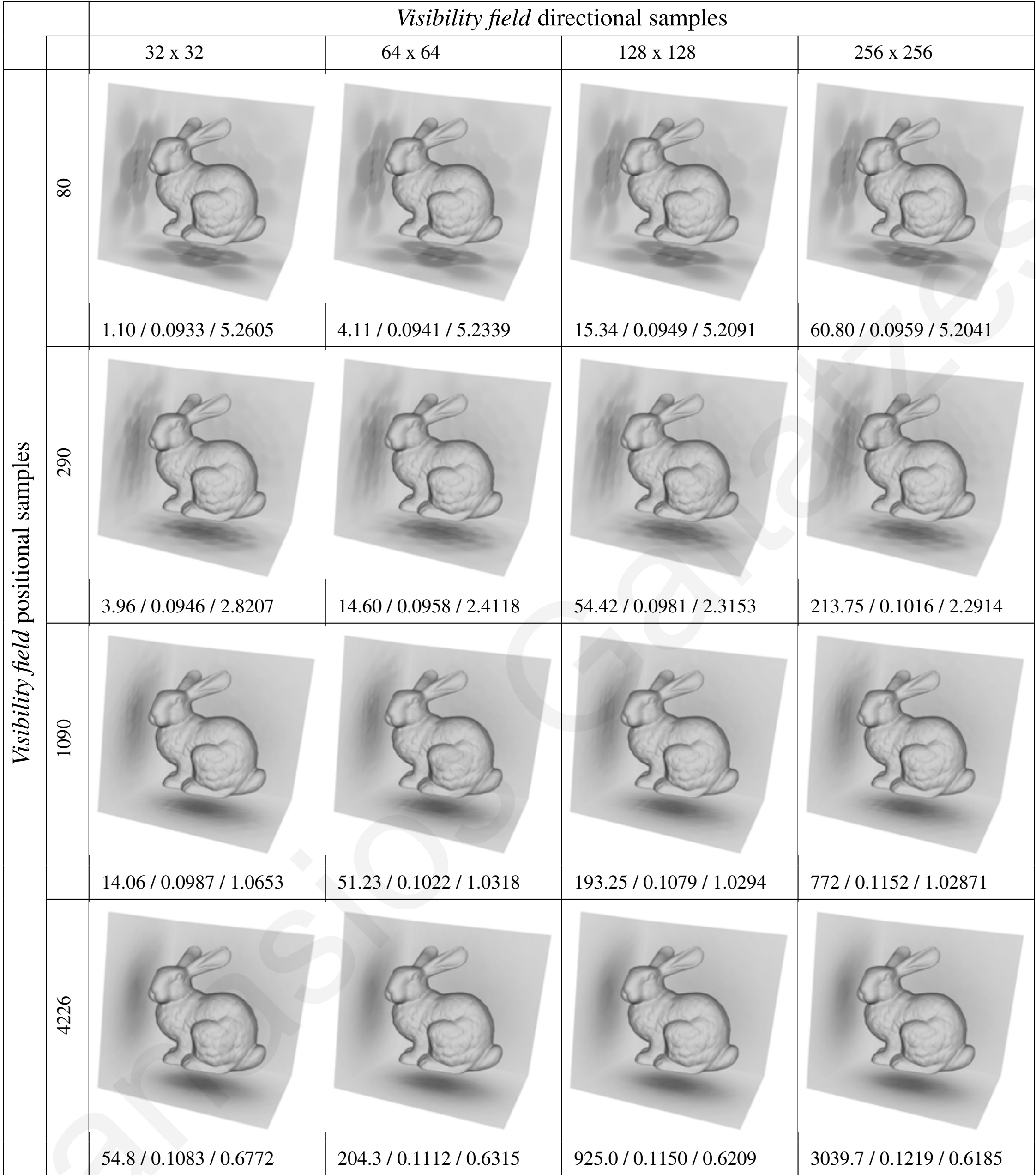



Figure 35: Cumulative table using 256 sample rays from each vertex of the tessellated corner (3x33x33) with a concentric map sampling distribution. The numbers under the images correspond to the preprocessing time, the run-time ambient occlusion computation on the CPU in seconds (using machine type 2) and the RMS error compared to the Reference image of Figure 34.

Athanasios Galatzes

| | Lemon Tree | Bunny | Igea | Multiple Objects |
|---|---|---|---|---|
| **Model** | | | | |
| Triangles | 26.3K | 39K | 67.2K | 142.3K |
| Rays cast | 836,352 | 836,352 | 836,352 | 26,444,800 |
| Ray casting time | 196.2 s | 331.1 s | 616.35 s | 4286.8 s |
| Pre-processing | 99.54 s | 54.79 s | 243.76 s | 334.6 s |
| AO calculation | 0.24 s | 0.228 s | 0.204 s | 4.692 s |

Table 1: The *visibility fields* algorithm applied to several different types of models and their respective CPU timings (using machine type 1). In the above images we used 256 sample rays with a concentric map sampling distribution. The ambient occlusion computation is done using the 4226 / 32x32 maps.

of the geometry in the given ray direction. This is a binary value, thus the method saves about 87.5% in texture space. The distance used in this case is the average distance of the sample points towards the object in the direction of the normal at the given sample point. In Figure 36 we can see the comparable results.

### 4.7 Implementation & Evaluation on the GPU

We implemented the real-time part of the above algorithm using the OpenGL® Shading Language (GLSL) [61] on a 32bit Intel Core 2 Quad Q6600 at 2.4GHz CPU and 4GB of main memory equipped with a GeForce GTS8800 GPU with 512MB of video memory. The window size was set to 512x512 for a total of 262144 pixels.

Athanasios Gaitatzes

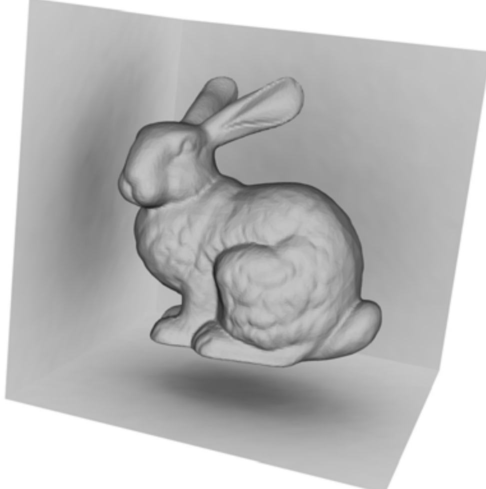


Figure 36: Using the 1 bit per direction optimization method with 4226 occlusion maps (positional samples) of size 64x64 and Concentric map sampling of 256 rays we get results which are comparable with the corresponding image from Figure 35, giving an RMS error of 4.4030.

#### 4.7.1 Ambient Occlusion

For most of the test runs the active pixels were about 200000 as only 75% of the window was rendered (the rest being black).

To acquire a reference image against which to compare our acceleration method in speed but mainly in image quality, we implemented ambient occlusion on the GPU using the uniform grid acceleration structure (see Figure 37 bottom-right).

We observe (in Figure 37) that the RMS error of the images compared to the reference image of the bunny, is very low and the achievable draw time, even for large models, is real-time. Based on the RMS error using 4226 64x64, visibility maps gives the same results as using maps of size 16642 32x32. We also infer from Figure 38 that the draw time is unaffected by the number of maps used thus the space required for the visibility maps depends only on the image quality that we would like to achieve.

Athanasios Gaitatzes

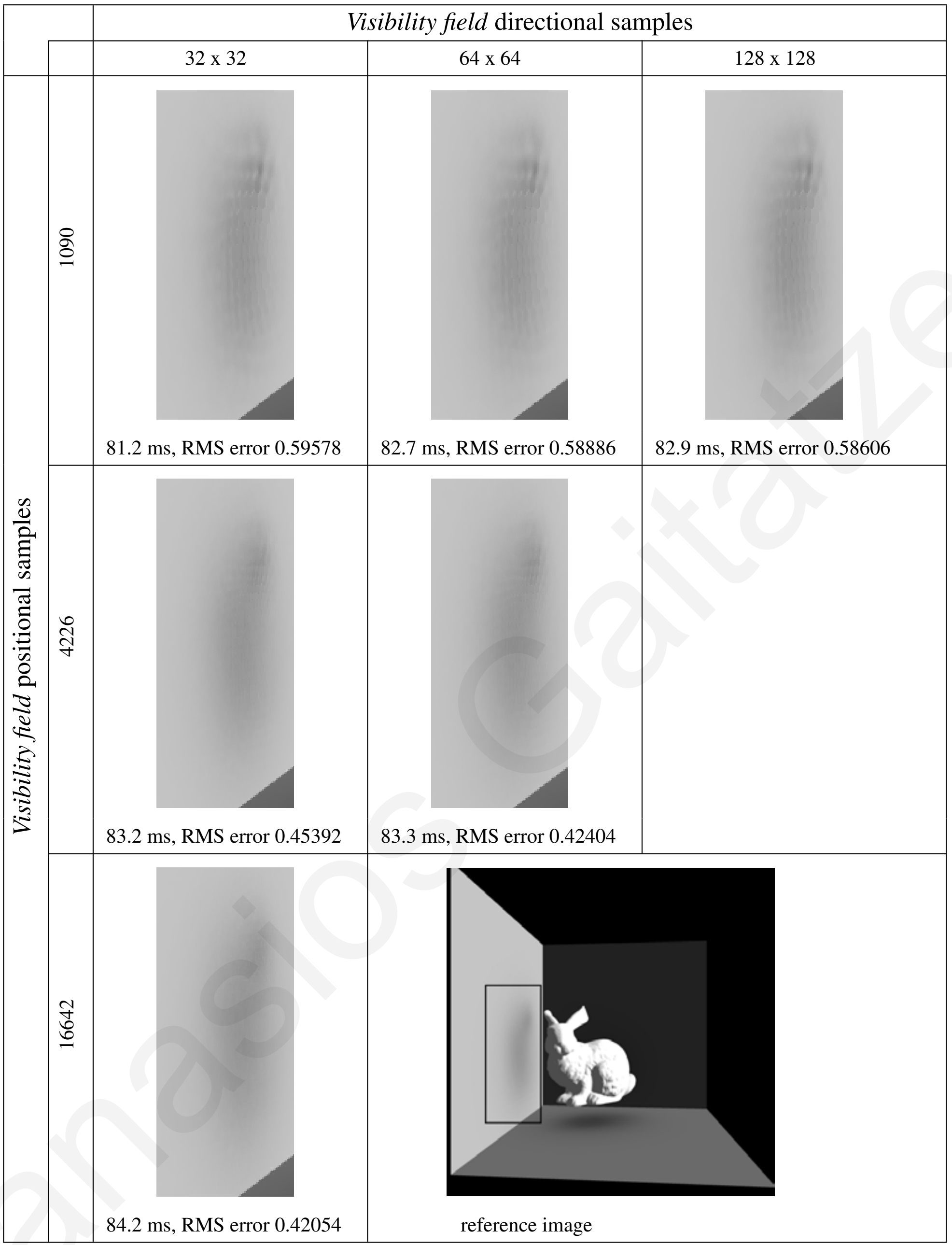



Figure 37: Inter-object ambient occlusion (close-up) of a bunny model using the *visibility fields* method with 256 rays per pixel implemented on the GPU. We report the draw time and the RMS error. On the bottom right the reference image rendered on the GPU using 256 rays per pixel in 7126 ms. The model itself is rendered using fixed-pipeline direct rendering.

Athanasios Gaitatzes

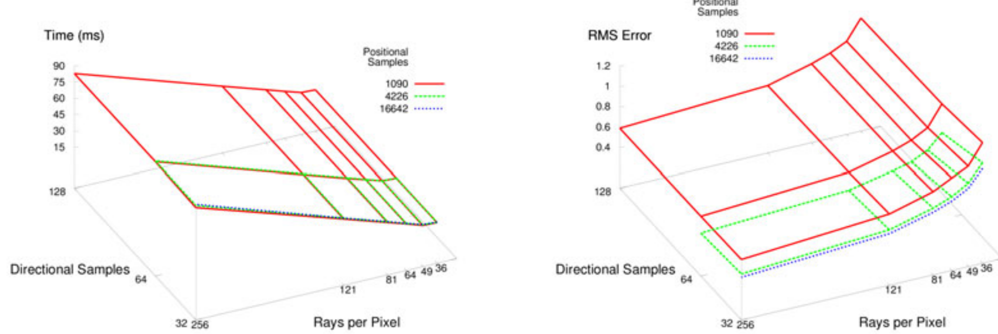
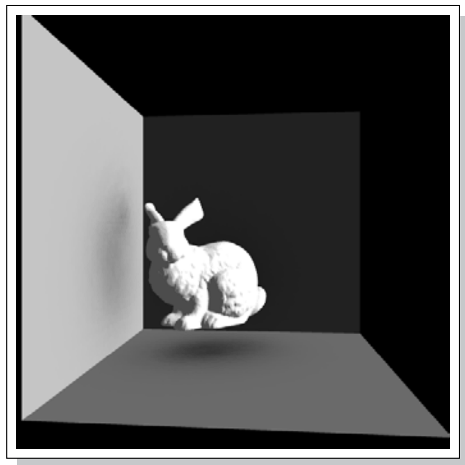



Figure 38: The draw time (left) and the RMS error (right) of the bunny model (39000 tris) plotted against different rays/pixel versus the size of the *visibility maps*. We observe that the frame draw time is not dependant on the number of *visibility maps* used or their size but rather on the amount of rays used.

In Figure 39 we show the application of our algorithm for the generation of inter-occlusion for several large models while maintaining the interactive nature of the algorithm. We achieve between 260 and 600 million rays per second, which is well above todays maximum performance reported by Horn et al. [45].

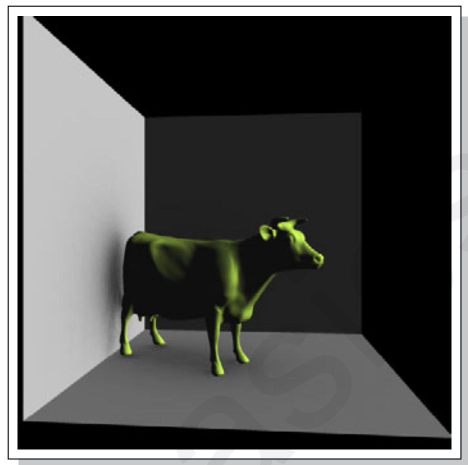
In Figure 40 the *visibility fields* were used for the generation of intra-object occlusion but because the ray sphere intersection algorithm always succeeds at finding an intersection (worst case since we are inside the bounding sphere of the object) the rendering times are up to 4 times slower than the inter-object occlusion case. Still the performance rate is above the one reported by Horn et al. [45]. We also observe that more visibility maps are required in this case in order to render a believable image. We attribute this to the fact that multiple rays, with small angular differentiation, originating on close points on the object, hit the same sample point on the objects bounding sphere. Thus, the same visibility map is used and the occlusion result looks grainy. When more maps are used the problem is alleviated.

Athanasios Gaitatzes

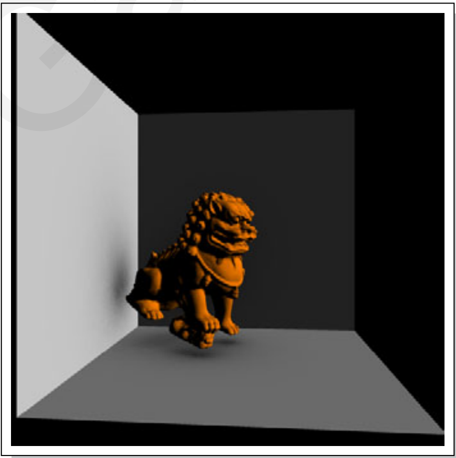


Bunny: 38889 triangles
Preprocessing time: 0.28 hours
Draw time: 39.4 ms
Rate: 614.21 M rays/s

Horse: 96966 triangles
Preprocessing time: 1.58 hours
Draw time: 73.3 ms
Rate: 330.15 M rays/s



Cow: 92864 triangles
Preprocessing time: 2.69 hours
Draw time: 59.3 ms
Rate: 408.09 M rays/s



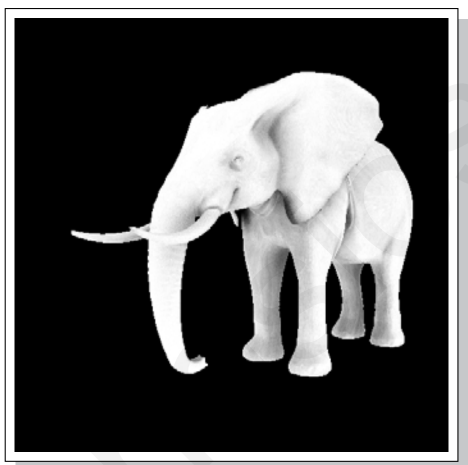
Dragon: 255138 triangles
Preprocessing time: 15.91 hours
Draw time: 93.3 ms
Rate: 259.37 M rays/s

Figure 39: The GPU *visibility fields* algorithm applied to several different types of models in order to compute inter-ambient occlusion. We used 4226 64x64 visibility maps, requiring 16.5 MB of space and 121 rays per pixel. The models themselves are rendered using fixed-pipeline direct rendering.

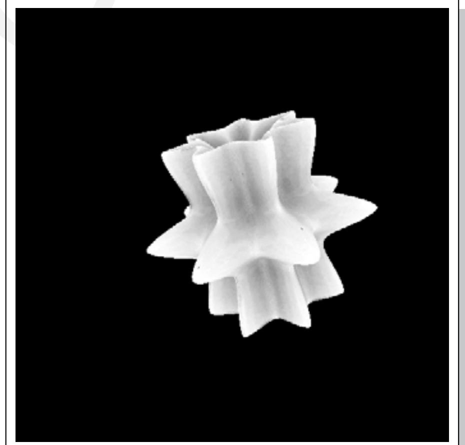
Athanasios Gaitatzes

Igea: 67170 triangles
Preprocessing time: 2.87 hours
Draw time: 202 ms
Rate: 119.8 M rays/s

Santa: 75777 triangles
Preprocessing time: 1.06 hours
Draw time: 183 ms
Rate: 132.2 M rays/s



Elephant: 157160 triangles
Preprocessing time: 21.02 hours
Draw time: 400 ms
Rate: 60.5 M rays/s



Super Shape: 261120 triangles
Preprocessing time: 52.37 hours
Draw time: 330 ms
Rate: 73.3 M rays/s

Figure 40: Intra-object ambient occlusion rendered on the GPU using 16642 64x64 visibility maps requiring 65 MB of space and 121 rays per pixel.

Athanasios Gaitatzes

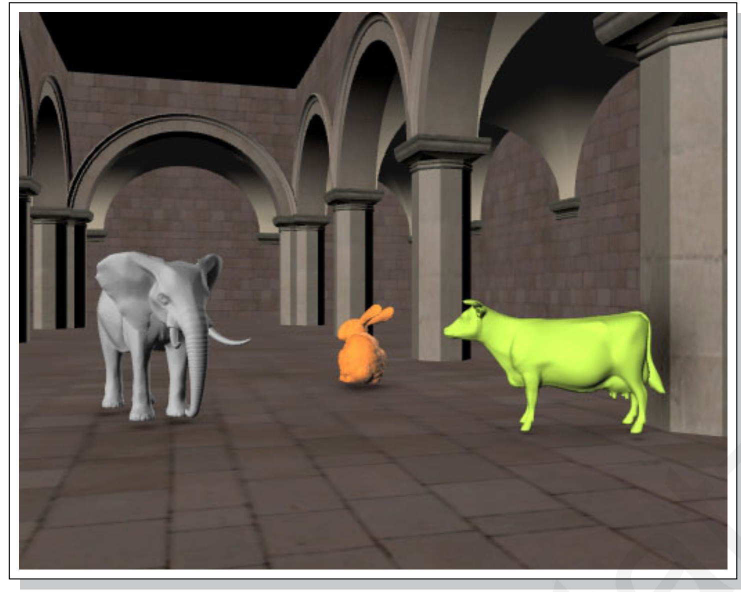


Figure 41: A scene of the Sponza Atrium with a bunny (38889 tris), a cow (92864 tris) and an elephant (157160 tris) rendered in three passes (one per object) with the *visibility fields* algorithm using 4226x64x64 maps and rendering in 2.5 frames per second.

In Figure 41 we show the Sponza Atrium rendered with several large polygon models inside it. The resulting draw time is contributed to the rendering method that uses one pass for each caster model. Just before each caster model is drawn, we enable subtractive blending (with OpenGL® blend equation GL_FUNC_REVERSE_SUBTRACT), in effect, removing colour from the image. The poor draw time is also attributed to the fact that non-visible pixels (the Sponza Atrium has a lot of non-visible geometry) are not culled before the fragment shader is executed on the GPU.

Even though the *visibility fields* method is only an approximation, it does a very good job at preserving image quality given the low memory requirements and achieved draw time.

### 4.7.2 Ray tracing

In Figures 42 and 43 we show a close-up of the bunny ears of using the *visibility fields* method. We show that very good results of soft shadows can be achieved while using 20 shadow ray

Athanasios Gaitatzes

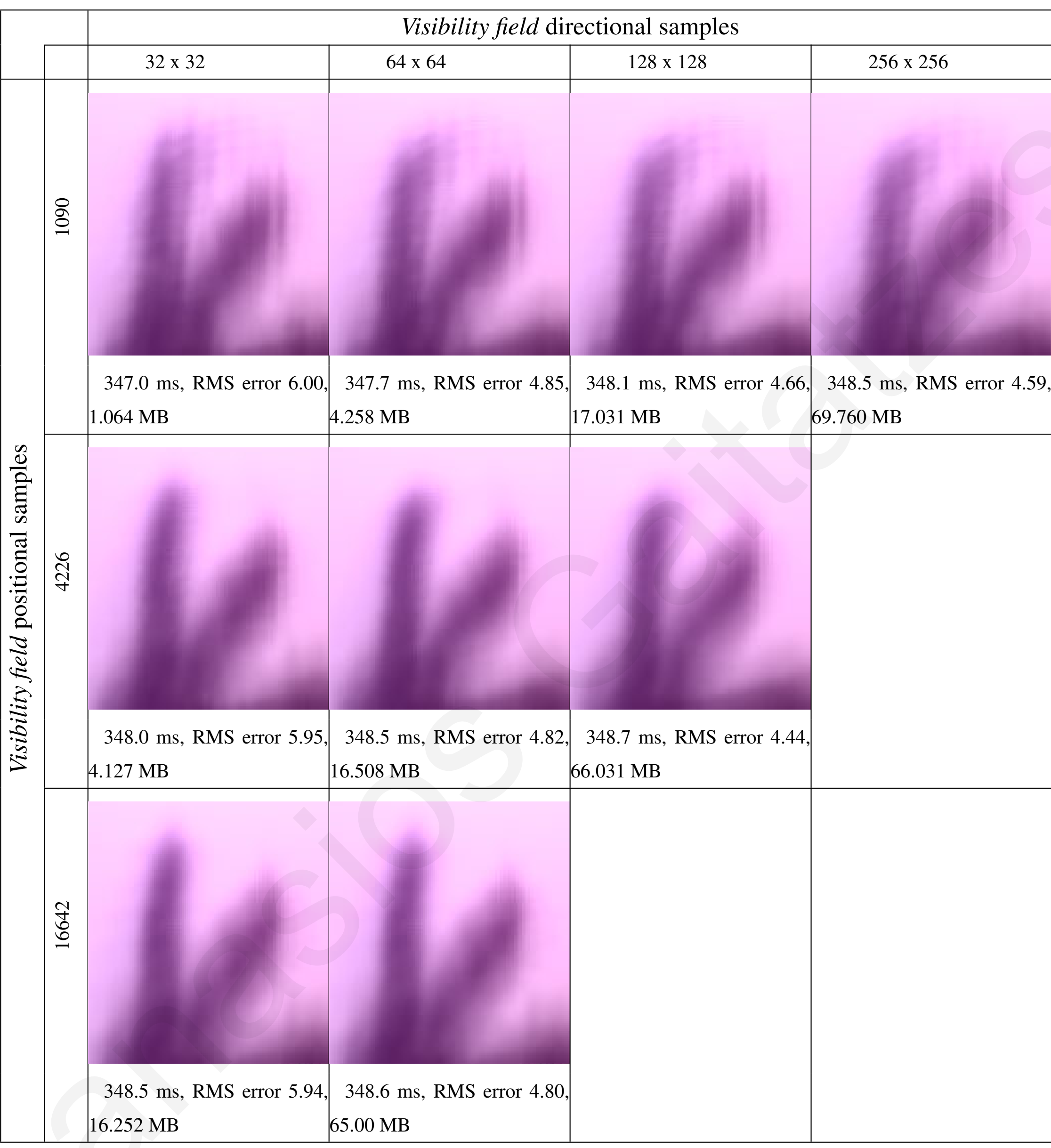
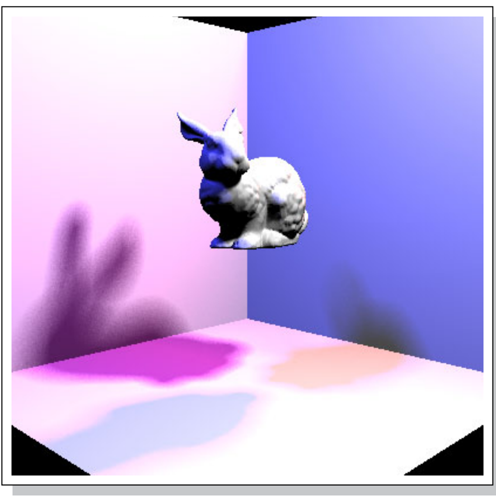



Figure 42: Close-up of the bunny ears rendered using the *visibility fields* method for the generation of soft shadows using 3 lights and 20 shadow ray samples on the GPU. We report the required time, the RMS error and the total space requirements.

Athanasios Gaitatzes



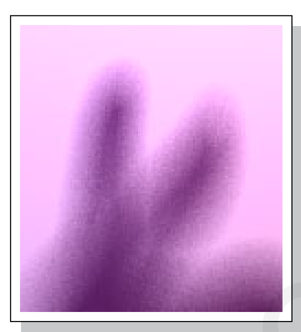


Figure 43: Reference image of the bunny, rendered using the BVH method with 3 lights and 256 rays per pixel taking 913,210 ms on the GPU. On the right, close-up of the ears.

samples along with 4226 64x64 visibility maps (i.e. 16.51MB of memory) as the visual quality is much better than the 1090 maps.

In Figure 44 we show a soft shadows rendering of a horse using various types of visibility maps to realise that their RMS error is similar. We can conclude that using the 1090 64x64 low-resolution maps is sufficient to get a very good approximation of the result. Comparing the timings with those of the bunny model we observe that the cost of using the visibility maps does not depend on the underlying geometry of the objects.

In Figure 45 we render a slightly more complex scene using 3 light sources of radius 2. As in the previous cases, the rendering time is almost completely affected by the primary rays which perform triangle intersection tests. Our method completes the rendering in 3268 ms, of which 70% is for the shadow rays. It produces a very good approximation of soft shadows using 20 shadow rays per pixel. For the total of 11,838,600 shadow rays, this corresponds to 1.9323 $10^{4}$ ms per shadow ray which is a very encouraging result. In the corresponding BVH GPU method to produce sharp shadows using just 1 shadow ray per pixel, the draw time is 48047 ms to compute

Athanasios Gaitatzes

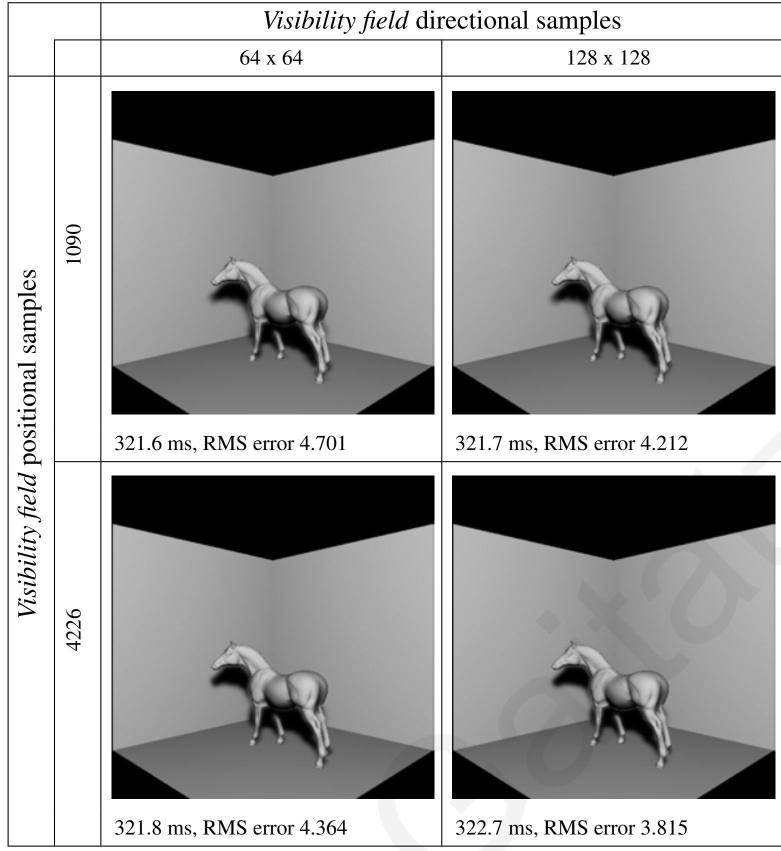



Figure 44: Soft shadow of the horse (96966 tris) using 1 light rendered using the visibility fields and 20 shadow ray samples. In contrast the reference image, using the BVH method and 256 shadow ray samples, on the CPU required 760012 ms and on the GPU 322100 ms.

the final image. Of that time 70% is used for the 591930 shadow rays yielding 5.682 $10^{2}$ ms per shadow ray.

In Figures 46 and 47 we use the *visibility fields* algorithm to render non-perfect-mirror reflections. The polished reference image is rendered with 4 rays for each reflective pixel leading to slower rendering times. However, we notice from the images and the RMS factor that the reflected sub region of our method is much closer to the result of the brushed metal reference image than the perfect mirror reference image. This strengthens our position that the proposed method is suitable for stochastic ray-tracing, as the quality of the rendered image is comparable to the reference

Athanasios Gaitatzes

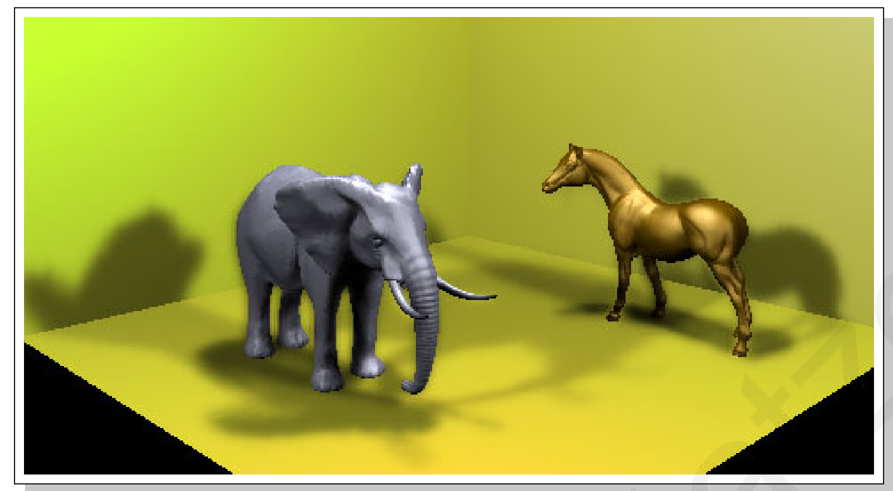
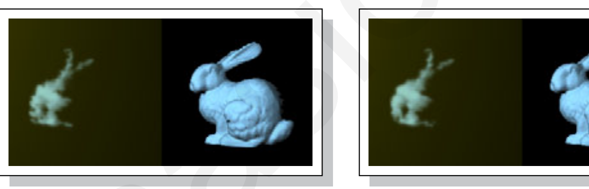
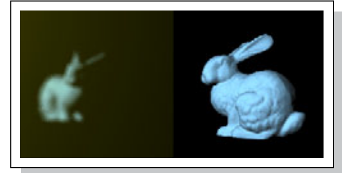


Figure 45: Close-up of a more complex scene using 3 point lights and 20 shadow ray samples rendered in 3268 ms using the *visibility fields* method. The BVH GPU method for sharp shadows takes 48047 ms.





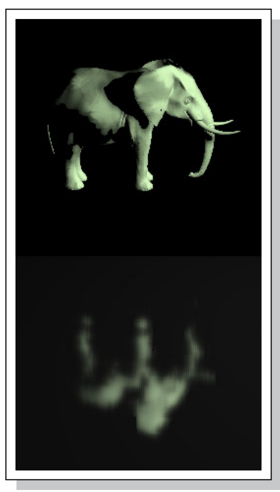
4226 32x32 maps, 440 ms,
4.555 RMS error, 16.508 MB

4226 64x64 maps, 441 ms,
4.480 RMS error, 66.031 MB

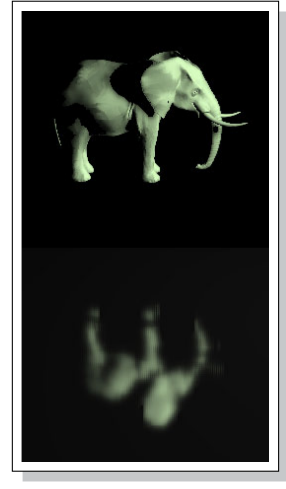
reference image: 5530 ms

Figure 46: Polished reflection of the bunny (39000 tris) using 4 rays per reflective pixel. From left to right: close-up views of our *visibility fields* GPU method where we report the draw time, the RMS error and the space requirements. Last is the reference image using the BVH method and its draw time.

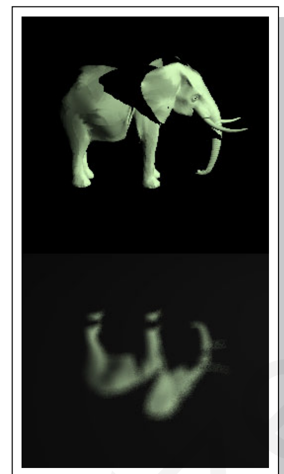
Athanasios Gaitatzes



4226 32x32 maps, 1897 ms,
9.392 RMS error, 16.508 MB



4226 64x64 maps, 1900 ms,
8.137 RMS error, 66.031 MB



reference image: 112910 ms

Figure 47: Polished reflection of an elephant (157160 tris) using 4 rays per reflective pixel. From left to right: close-up views of our *visibility fields* GPU method where we report the draw time, the RMS error and the space requirements. Last is the reference image using the BVH method and its draw time.

image. Furthermore, the rendering time, even using 4 rays per reflective pixel, is very close to ray-casting without secondary rays.

## 4.8 Limitations

The *visibility fields* method is not very well suited for elongated models (see Figure 48). The occlusion produced, even when using 16642 maps is pretty grainy. In addition models that are highly concave would fail to produce accurate visibility maps as it would not be possible to record all of the tight concavities of the model. Surrounding the model with more spheres could be a solution to this limitation even though this would require additional texture space.

Athanasios Gaitatzes

Figure 48: Inter-ambient occlusion of a cane (elongated object) rendered on the GPU using 121 rays per pixel. On the left using 4226 128x128 maps and on the right using 16642 128x128 maps. (The image brightness is doubled for clarity)

### 4.9 Summary

We have presented the *visibility fields*, a discretization of the visibility around an object, that can be used for secondary diffuse illumination (i.e. ambient occlusion) and ray tracing calculations where exact ray hits are not critical (i.e. soft shadow rays). We have shown how it can be used for an interactive inter-object ambient occlusion approximation computation. For the intra-object occlusion case the number of required maps is large and the draw time needs improvement when the model covers a lot of pixels on the screen. In a game environment though, where several models exist on the screen and their coverage is not very big, the intra-object occlusion method can be used even for high triangle count models.

The method especially favors large model data sets, where we maintain a constant computation time, independent of the model complexity. Our method is robust, has a relatively small memory footprint against comparable existing methods and the time required to generate the visibility maps depends only on the complexity of the occluder geometry. In addition, the number and resolution

Athanasios Gaitatzes

of the maps used in the *visibility fields* can be adjusted depending on the required accuracy and the available memory. The same maps can be used for both inter- and intra-object ambient occlusion computation.

Furthermore, our algorithm can be applied to ray-tracing calculations where exact ray hits are not critical, for example for shadow and secondary ray intersection tests, such as soft shadow rays and Monte Carlo ray-tracing.

We have shown that in the above mentioned cases the production of the desired image is accelerated while the results remain close to the reference images. The hybrid method we propose favors large model data sets as in ambient occlusion. This result is expected as all triangle intersection tests for shadow and secondary rays are replaced with constant time operations. In this way rendering time is affected mostly by the primary rays that give us the visibility of the scene.

In the next chapter we will discuss the discretization of the illumination in an environment into a volume that will assist us in generating in real-time diffuse global illumination.

Athanasios Gaitatzes

# Part III

## – Discretization of Illumination –

## Virtual Point Light Methods

Athanasios Gaitatzes

Athanasios Gaitatzes

# Chapter 5

## Interactive Volume-based Indirect Illumination of Dynamic Scenes

### 5.1 Motivation

In order to synthesize photo-realistic images we need to capture the complex interactions of light with the environment. Light follows many different paths distributing energy among the object surfaces. This interplay between light and object surfaces can be classified as local illumination and global illumination. Local illumination algorithms take into account only the light which arrives at an object directly from a light source. Global illumination algorithms, on the other hand, take into account the entire scene, where the light rays can bounce off the different objects in the environment or be obstructed and absorbed.

### 5.2 Overview

We present a real-time algorithm to compute the global illumination of dynamic scenes with complex dynamic illumination. We will create a discretization of the light in the scene using a *Virtual Point Light* (VPL) illumination model on the volume representation of the scene by generating maps from the perspective of the light sources and injecting the samples of these maps



Athanasios Gaitatzes

into a volume while at the same time creating an additional volume that would hold the occlusion / visibility of the geometry in the scene. Unlike other dynamic VPL-based real-time approaches, our method handles occlusion (shadowing and masking) caused by the interference of geometry and is able to estimate diffuse inter-reflections from multiple light bounces.

### 5.3 Introduction

In the previous chapter we presented a discretization of the visibility function by clustering together intersection rays. In this chapter we will discretize the illumination of the scene using a *Virtual Point Light* method.

We propose a method that produces photo-realistic images of diffuse, dynamic environments in real-time, by estimating the slowly varying global illumination at discrete locations in the environment and applying the results on the scene geometry. This way, we can capture shadowing effects as well as diffuse inter-reflections from multiple secondary light bounces. The method we propose uses a uniform discretization of the scene, incorporating geometry information in the discretization structure. Instead of using the shadow map data as virtual point lights (VPLs) [18] [19] [54], our method performs a complete scene voxelization and is thus able to include occlusion information along with any direct, indirect and self-emitted illumination. Furthermore, it is capable of calculating global illumination from multiple light bounces and include energy from emissive materials in the process.

Scene voxelization describes the process of turning a scene representation consisting of discrete geometric entities (e.g. triangles) into a three-dimensional regular spaced grid. Each cell of the grid encodes specific information about the scene. Depending on the type of voxelization, this information can be different. In the case of a binary voxelization, a cell stores whether geometry

Athanasios Gaitatzes

is present in this cell or not. The cells can be represented by single bits in a bitmask. In a multi-valued voxelization, the cell can also store information like material or normals. Furthermore, voxelization can be divided into boundary or solid voxelization. Boundary voxelization encodes the object surfaces only, whereas solid voxelization captures the interior of a model as well.

Both full occlusion and emissive materials were not handled by previous methods. Also, the most successful approaches either could not handle secondary bounces or could do so at a great penalty in terms of computation cost and/or severe irradiance storage interference.

## 5.4 Mathematical Background

### 5.4.1 Review of Spherical Harmonics

The spherical harmonics (SH) are a set of orthonormal basis functions defined over a sphere, in the same manner that the Fourier series is defined over an $N$-dimensional periodical signal. The *Spherical Harmonic* (SH) functions in general are defined on imaginary numbers, but since in graphics we are interested in approximating real functions over the sphere (i.e. light intensity fields), we use the real spherical harmonics basis. Given the standard parameterization of points on the surface of the unit sphere into spherical coordinates $(\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta) \to (x, y, z)$ the real SH basis functions of order $l$ is defined as:

$$Y_l^m(\theta,\phi) = \begin{cases} \sqrt{2}K_l^m \cos(m\phi)P_l^m(\cos\theta) & m > 0 \\ \sqrt{2}K_l^m \sin(\;\; m\phi)P_l^{\;\; m}(\cos\theta) & m < 0 \\ K_l^0 P_l^0(\cos\theta) & m = 0 \end{cases} \tag{4}$$

where $l \in \mathbf{R}^+, \quad l \le m \le l$, $P_l^m$ is the *associated Legendre polynomial* and $K_l^m$ is the scaling factor to normalize the functions which is defined as:

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l \quad |m|)!}{(l+|m|)!}} \tag{5}$$

Athanasios Gaitatzes

The spherical harmonics possess several important properties, such as rotation invariance. This means that the SH approximation $f$ of a spherical function $f$ rotated by some rotation operator R is the same regardless of the order of application of the rotation: rotating the SH projection of $f$ gives the same approximation as rotating $f$ and then projecting it. This prevents aliasing artifacts from occurring while rotating functions and means that we can simply rotate the projection of a function instead of re-projecting a rotated function.

Similar to the Fourier series expansion, a function on the sphere $f(\theta, \phi)$ can be represented in terms of *spherical harmonics coefficients* $f_{l,m}$ as:

$$f(\theta, \phi) = \sum_{l=0}^{\infty} \sum_{m=\ \ l}^{l} f_{l,m} Y_l^m(\theta, \phi) \tag{6}$$

A signal over a sphere is approximately reconstructed using a a truncated SH series, by projecting the initial function $f$ onto a finite set of SH coefficients up to order $l = n, l \in \mathbf{N}$. Typically, in computer graphics a maximum order of 6 is used.

Additionally, because of orthnormality, the integral of two reconstructed spherical functions that have been projected in the SH basis is reduced to the inner product of the vectors of their SH coefficients. For band-limited SH functions of order $n$, the integral becomes:

$$\int \tilde{f}(\theta, \phi)\tilde{g}(\theta, \phi) = \sum_{l=0}^{n} \sum_{m=\ \ l}^{l} f_{l,m} g_{l,m} \tag{7}$$

### 5.4.2 Radiance Transfer

In order to accurately model light in an environment, the energy transfer has to be evaluated on each surface location. The *Rendering equation* (see Equation 1), proposed by Kajiya [52], associates the outgoing radiance $L_o(\mathbf{x}, \vec{\omega}_\mathbf{o})$ from a surface point $\mathbf{x}$ along a particular viewing direction $\vec{\omega}_o$, with the intrinsic light emission $L_e(\mathbf{x}, \vec{\omega}_\mathbf{o})$ at $\mathbf{x}$ and the incident radiance from every direction $\omega_i$ in the hemisphere $\Omega$ above $\mathbf{x}$, using a BRDF that depends only on the material properties and

Athanasios Gaitatzes

the wavelength of the incident light. The hemisphere-integral form of the *Rendering equation* can be written as:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int\limits_{\Omega} L_i(\mathbf{x}, \vec{\omega}_i) f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o) cos(\theta) d\vec{\omega}_i \tag{8}$$

where $f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o)$ is the bidirectional reflectivity distribution function of the surface at point **x**, expressing how much of the incoming light arriving at **x** along direction $\vec{\omega}_i$ is reflected along the outgoing direction $\vec{\omega}_o$. $L_i(\mathbf{x}, \vec{\omega}_i)$ is the light arriving along direction $\vec{\omega}_i$.

If we group the cosine term and $f_r$ into a single *transfer function T*, (Sloan et al. [100]), which expresses how point **x** responds to incoming illumination, then Equation 8 becomes:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + L_r(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int\limits_{\Omega} L_i(\mathbf{x}, \vec{\omega}_i) T(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o) d\vec{\omega}_i \tag{9}$$

The above generic energy transfer equation can be used in fact to model any variation of the *Rendering equation*. In our case, where a point in space (voxel center) is illuminated, it is more convenient to consider an integral over the entire sphere surrounding the voxel center. Furthermore, the voxel center can behave as a spherical particle, receiving energy with maximum flow from every direction. Therefore, the cosine term is dropped as the projected solid angle towards the emitting location along $\vec{\omega}_i$ always equals $d\vec{\omega}_i$.

Using the orthonormality property of the SH function basis (Equation 7) and considering for the moment only the reflected radiance, the integral in Equation 9 can be approximated with a finite set of terms as:

$$L_r(\mathbf{x}, \vec{\omega}_o) \approx \sum_{l=0}^{n} \sum_{m=\ l}^{l} L_{l,m} T_{l,m} \tag{10}$$

## 5.5 Method Overview

We have extended the work of Kaplanyan [54], in order to take into account occlusion in the light transfer process and secondary light bounces. Our method is based on the full voxelization

Athanasios Gaitatzes

of the geometry instead of the injection of only the relfection shadow map points (VPLs) in a volume grid. This way, the presence of geometry that is unlit by the direct illumination is also known and light interception and reflection is possible. The voxelization records — among others — direct illumination and scalar occupancy data, thus enabling the indirect illumination from emissive materials and the transmission through transparent elements.

Our method consists of three stages. First the scene (or a user-centered part of it) is discretized to a voxel representation. Next, the radiance of each voxel is iteratively propagated in the volume and finally, during image rendering, the irradiance of each surface point is calculated by sampling the radiance from the nearest voxels.

To this effect, we use several 3D volume buffers. An *accumulation volume buffer* is used for the storage of radiance samples when light bounces off occupied voxels. This buffer is sampled during the final rendering pass to reconstruct the indirect illumination. For each color band it stores a spherical harmonic representation of the radiance of the corresponding scene location (4 coefficients encoded as RGBA float values). It is initialized with zero radiance. For the iterative radiance distribution, a *propagation volume buffer* is used (see Section 5.5.2). The propagation buffer stores a spherical harmonic representation of the radiance to be propagated in the next propagation iteration. It is initialized with the radiance from the first bounce VPLs (direct illumination). Both the accumulation and the propagation buffers are read and write buffers. Our algorithm also reads information from one more read-only volume buffer that contains information about the scene normals and surface albedo (see Figure 49 (b) and (c)). Average normals and space occupation (scalar voxelization value, also accounting for transparency) are compacted into a single voxel value.

To discretize our scene in real-time, we create a uniform spatial partitioning structure (voxel grid - see Section 5.5.1) on the GPU, where we store the geometry and radiance samples. We

Athanasios Gaitatzes

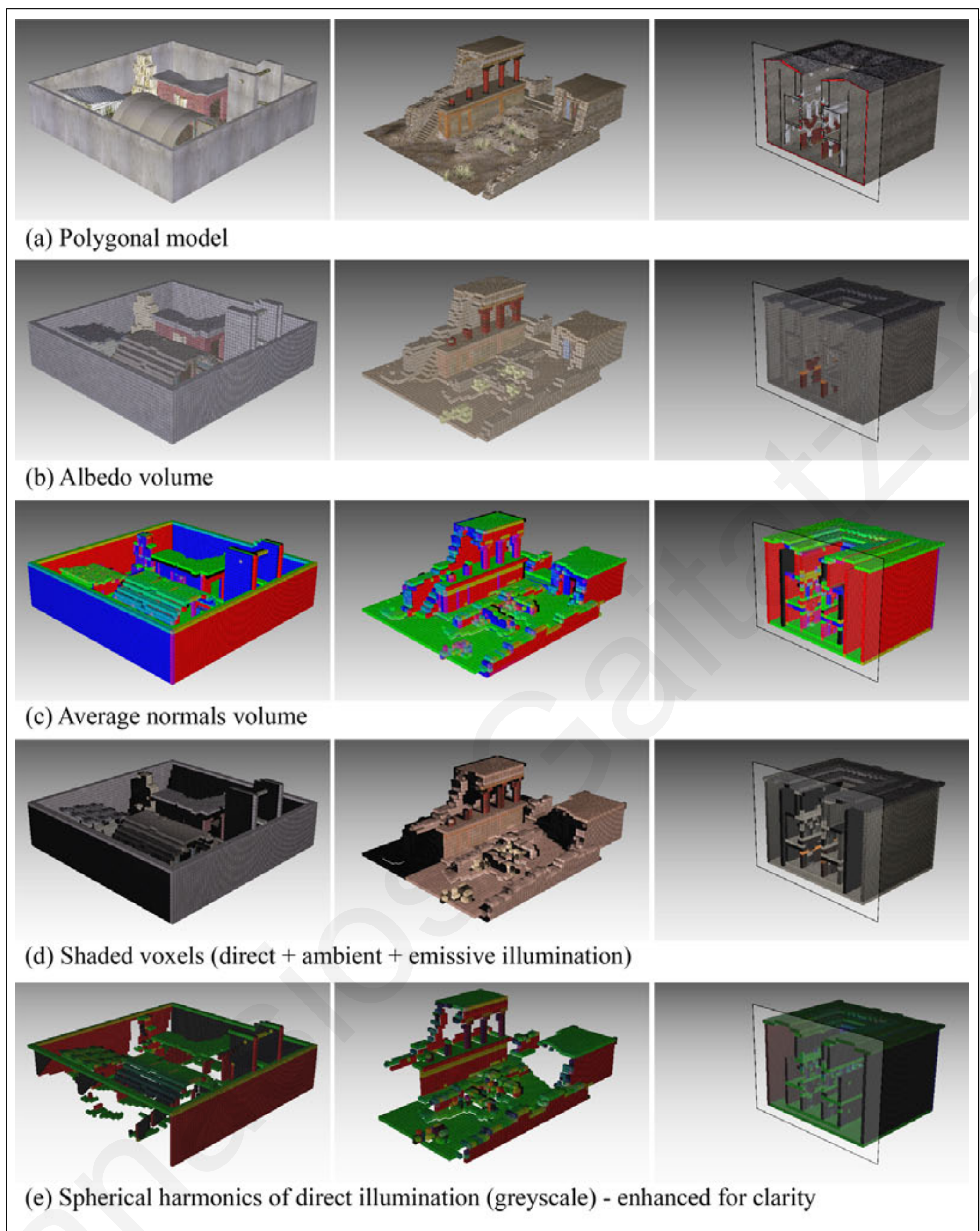



Figure 49: Several environments voxelized into a $64^3$ grid. Column 1: a model of 10,220 triangles (Arena). Column 2: a model of 109,170 triangles (Knossos). Column 3: the Sponza II Atrium of 135,320 triangles (cross section depicted). All buffers are floating point and have values $\in$ [-1,1]. As such when viewing the buffers some black voxels may appear (see the normals map) indicating negative values.

Athanasios Gaitatzes

inject VPLs in our voxel space, which are essentially hemispherical lights with a cosine falloff. The VPLs are then approximated by a low-order spherical harmonic coefficients representation (see Figure 49 (d) and (e) respectively). Similar to Kaplanyan [54], we use an iterative diffusion approach on the GPU to propagate energy within space. In contrast to [54] though, since we obtain the space occupation information from the voxelization (not-just VPLs), energy is propagated only in void space, from one voxel boundary to the next. The propagated radiance is *reflected* on occupied (voxelized) volume grid points and accumulated at these locations in the accumulation buffer (see Section 5.5.2). The new propagation direction is determined by the stored average voxel normal of the occupied voxel.

During the rendering pass (see Section 5.5.3), for each fragment, the volume is queried as a texture and the closest texels (accumulated irradiance) are used to estimate the global illumination at that point in the scene.

### 5.5.1 Real-Time Voxelization

Instead of applying one of the fast binary GPU voxelization methods, such as Eisemann's et al. [23], we use a variant of Chen's et al. algorithm [11] because we need to store multi-channel scalar data in each voxel. More specifically, we use the same steps as Chen's algorithm, for the slicing of the volume. The main difference is that we do not use the originally proposed XOR operation because in practice, very few models are watertight and many volume attributes cannot be defined for interior voxels. Therefore, our method produces only volume shells.

In brief, for every volume slice (see Algorithm 4), a conventional scan-conversion of the scene geometry takes place and the generated fragments correspond to the voxels of this slice (Figure 50). Rasterization is incremental and requires that the slope of the dependent variable on the increment is less than 1. As far as the scan conversion of a polygon onto the (slice-oriented) view

Athanasios Gaitatzes

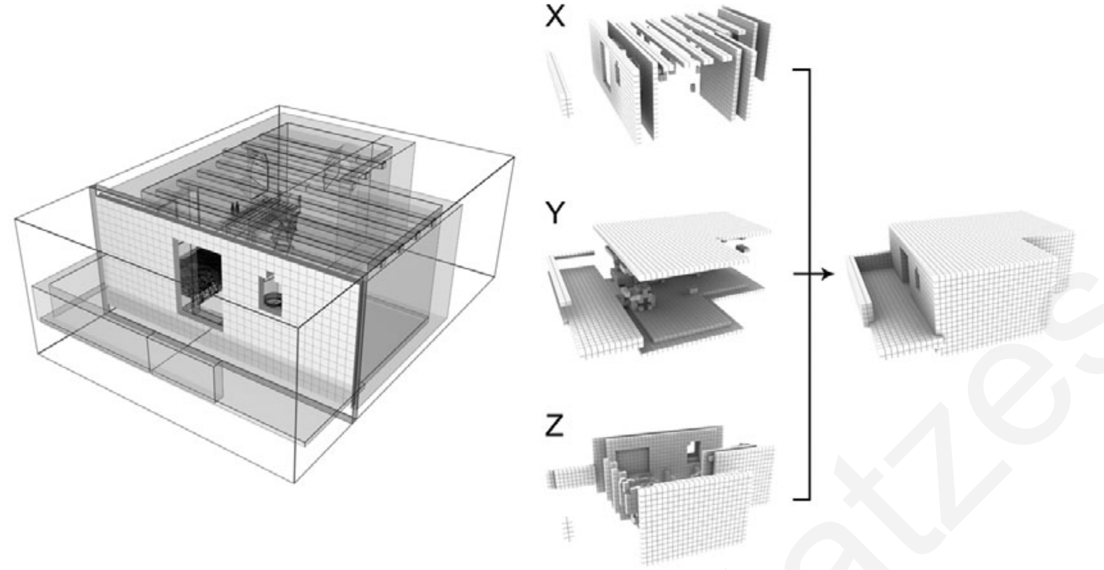



Figure 50: Slice-based voxelization (left) and composition of the three sub-volume passes into one voxelized volume (right).

plane is concerned, there are no holes generated but when the fragments are stored as voxels, discontinuities along the Z-axis occur (slicing direction). The XOR operation indirectly solved this problem (by filling in the missing fragments) but in our case this is not an option. The problem is solved by repeating the scan-conversion process 3 times, once for each primary axis. This way, we ensure that the depth-discontinuity in one orientation of the view plane will be remedied in one of the two others (see Figure 50 (right)).

**Algorithm 4:** Pseudo-code for Scene Voxelization

generate a bounding box of the scene;

**for** $i \leftarrow 1$ **to** *N volume slices* **do**
  define a voxel-deep, thin orthogonal viewing frustum along X-axis;
  execute 2D scan conversion for all object faces;
  store result in slice $i$ of volume buffer-X;

repeat above loop for Y and Z axes;

combine the three temporary volumes, buffer[XYZ], into one final volume keeping the MAX value for each corresponding cell in all three volumes;

Athanasios Gaitatzes

During the above 3-way voxelization, the radiosity of each grid cell is computed using direct illumination (complete with shadows and emissive illumination). Three buffers on the GPU are needed to store the temporary volume results of the three slicing procedures (one for each different orientation of the object). Finally, those three volumes are combined into one buffer, using the MAX frame buffer blending operation. The maximum radiocity of each cell is stored as a spherical harmonic representation. These values will be used as the initial radiance distribution in the propagation buffer for the iterative radiance distribution.

### 5.5.2 Iterative Radiance Distribution

Once we have injected the VPLs into the initial 3D volume, we need to propagate their initial radiance to their neighboring voxels. The propagation stage consists of several sequential iterations performed entirely on the GPU. Each iteration represents one discrete step of radiance propagation in the (empty) 3D volume. We effectively perform radiance shooting at each volume location by gathering radiance instead from each one of the voxel's neighbors (see Figure 51) and interpolating the weighted sum of the corresponding directional contributions on the GPU.

Similar to [54], we split the integral of radiance gathering (Equation 9 for spherical integration domain) into six sub-domains corresponding to the six sides of the receiving voxel. Instead of considering only unobstructed propagation, though, the transfer function $T_{j \to i}$, between the neighboring voxel $j$ and the current voxel $i$, is split into a geometric (transfer) term $T_{cone(j)}$ and a reflective term $T_{r(j)}$. The four $T_{cone(j)}$ coefficients are pre-computed from the six rotated spherical harmonic functions of a 90-degree cone. $T_r(j)$ is used for the deflection of the incident radiance.

When voxel $i$ corresponds to void space, radiance is propagated in the direction from voxel $j$ to $i$. This means that when no obstacle is encountered, $T_r(j)$ coefficients are equal to 1. On the other hand, when voxel $i$ is occupied, the spherical harmonic function of the incident radiance

Athanasios Gaitatzes

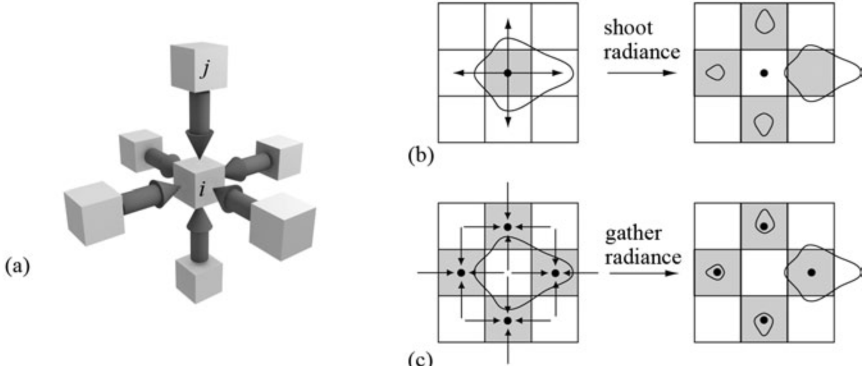



Figure 51: Radiance Gathering Illustration (a). The radiance for the center voxel is gathered from the values stored at the voxels of the surrounding cells. Radiance shooting (b) in the radiance propagation procedure is equivalent to radiance gathering (c).

from voxel $j$ should be mirrored with respect to the plane that is perpendicular to the plane of reflection and parallel to the average normal direction stored in the volume buffer. This requires two SH rotations and a mirroring operation. See [65] for details on the rotation of real spherical harmonics. For speed and simplicity though, this operation is replaced by a mirror reflection on the voxel boundary, i.e. along one of the three primary axes. Therefore, the four coefficients of $T_{r(j)}$ are 1 except the one corresponding to the mirror direction. Taking into account the above factors, the gathering operation becomes:

$$\int_S L_i(\mathbf{x}, \vec{\omega}_i) T(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o) d\vec{\omega}_i = \sum_{j=1}^{6} \sum_{l=0}^{1} \sum_{m=\ l}^{l} L_{(j),l,m} T_{cone(j),l,m} T_{r(j),l,m} \quad (11)$$

Figure 52 demonstrates the propagation and radiance accumulation process.

### 5.5.3 Final Illumination Reconstruction

During the final rendering pass, the irradiance at each surface point is computed from the incident radiance $L_i$ that is stored in our uniform grid structure (accumulation buffer). The irradiance

Athanasios Gaitatzes

$E$ at point **x** can be derived by integrating the definition of incident radiance over the hemisphere above **x**:

$$E(\mathbf{x}) = \int_{\Omega} L_i(\mathbf{x}, \vec{\omega}_i) cos\theta d\vec{\omega}_i \tag{12}$$

where $\theta$ is the angle between the surface normal and the incident radiance direction $\omega_i$. The integration domain is the hemisphere $\Omega$ defined by the surface normal $\mathbf{n_x}$ at point **x**.

In order to include the color filtering at the final gathering stage as well as the material emission, we can estimate the radiosity $B(\mathbf{x})$. For diffuse surfaces, $B(\mathbf{x})$ is given by the following hemisphere-integral equation, after multiplying Equation 8 with $\pi$ (and hense $L_i$):

$$B(\mathbf{x}) = B_e(\mathbf{x}) + \frac{\rho(\mathbf{x})}{\pi} \int_{\Omega} B_i(\mathbf{x}, \vec{\omega}_i) cos\theta d\vec{\omega}_i \tag{13}$$

where $\rho(\mathbf{x})$ is the albedo of the surface. If we change the integration domain to the full sphere $\Omega'$, the previous equation can be rewritten as follows:

$$B(\mathbf{x}) = B_e(\mathbf{x}) + \frac{\rho(\mathbf{x})}{\pi} \int_{S} B_i(\mathbf{x}, \vec{\omega}_i) T(\mathbf{n_x}, \vec{\omega}_i) \omega_i \tag{14}$$

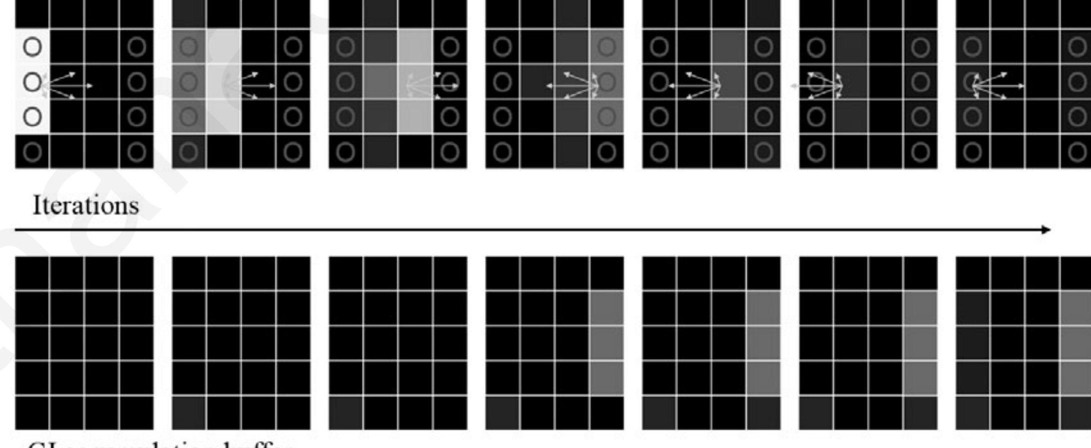



Figure 52: Simplified example of the propagation and light reflection process.

Athanasios Gaitatzes

where the function $T$ is defined as follows:

$$T(\mathbf{n_x}, \vec{\omega}_i) = \begin{cases} \cos\theta, & \theta < \pi/2 \\ 0, & \theta > \pi/2 \end{cases} \tag{15}$$

This change of the integration domain is necessary because we are going to use spherical harmonics, which are defined over the sphere and not the hemisphere.

Equation 14 is directly evaluated per pixel to give the final indirect lighting. In our algorithm the radiance $L$ is tri-linearly interpolated from the stored values in the uniform grid structure. From the eight closest grid points only the ones corresponding to occupied voxels are considered for interpolation. $L_i$ is already stored and interpolated in spherical harmonic representation. We also build a spherical harmonic representation for the function $T$, as described in [54] and the integral is calculated per pixel as a simple dot product, as shown in Equation 7.

### 5.6 Implementation & Evaluation

We have integrated the above algorithm in a real-time deferred renderer using OpenGL® and the OpenGL® Shading Language (GLSL) [61]. Our proof of concept implementation uses a $2^{nd}$ order spherical harmonic representation, since the four SH coefficients, map very well to the four component buffers supported by the graphics hardware. All results were rendered on an Intel Core i7 860 at 2.8GHz with 8GB of RAM and equipped with an NVIDIA® GeForce GTX285 GPU with 1GB of video memory at 512x512 pixels with a $32^3$ grid size. It should be noted here that, excluding the final interpolation stage, the performance of the indirect lighting computation in our method does not depend on the final screen resolution, but only on the voxel grid size and the number of propagation steps. This is a big advantage over instant radiosity methods, like imperfect shadow maps.

Athanasios Gaitatzes

| | Triangles | Grid size | Iterations | Voxelization (ms) | Propagation (ms) | Total (ms) |
|---|---|---|---|---|---|---|
| test scene | 48 | $32^3$ | 11 | 10 | 12 | 22 |
| room | 704 | $32^3$ | 64 | 3 | 61 | 69 |
| arena | 10219 | $32^3$ | 12 | 3 | 13 | 21 |
| sponza | 66454 | $32^3$ | 11 | 10 | 11 | 28 |

Table 2: Time measurements of our test scenes in milliseconds. Only the voxelization and propagation times are relevant to our work. The total rendering time includes the direct lighting computation and other effects and is given as a reference. Note that higher grid sizes are prohibitive using the current hardware.

Table 2 shows comprehensive time measurements for all the scenes detailed below. All scenes are considered to have fully dynamic geometry and lighting conditions. In all cases our algorithm achieves real-time frame rates and sufficient accuracy in the reproduction of the indirect diffuse illumination, even though our implementation is not optimized in any way.

We have found that the propagation stage of our method is limited by the available memory bandwidth and not the computational speed. This is to be expected, since the propagation kernel requires 52 floating point reads and 8 floating point writes per color band. To save memory bandwidth we do not store the diffuse color of the surfaces in the voxel structure, but after the first light bounce we consider it constant and equal to 0.5 for each color band.

Figure 53 shows a direct comparison of our algorithm with a reference solution on a simple test scene. We can see that our method reproduces the shape and the properties of the indirect illumination in the reference solution.

Figure 54 shows a room lit through a set of open windows. This is a challenging scene for global illumination algorithms, because only a small region on the left wall is directly lit by the

Athanasios Gaitatzes

sun and the majority of the lighting in the room is indirect. We can see that the simple light propagation method completely fails to reproduce the indirect lighting in the room, since it is not taking into account secondary light bounces and occlusion. At least two bounces of indirect lighting are required to get meaningful results in this case. In our method, when a grid of size $N$ is used, the distance between the walls of the room is also $N$, so $kN$ propagation steps are required to compute $k$ bounces of indirect illumination. This is a worst case scenario, as in typical scenes light interaction from one end of the scene to the other is not required. In this particular case we have used 64 propagation steps to simulate two bounces of light on a $32^3$ grid. The resulting illumination is visually pleasing, giving high contrast on the edge of the walls and the staircase. Since our algorithm takes indirect occlusion in consideration, the area below the staircase is correctly shadowed. We observe some artifacts below the windows, due to the imprecision of the spherical harmonics and the fact that the grid cell on this area covers both the edge of the wall and the empty space inside the window. Even with a number of propagation steps this high, our

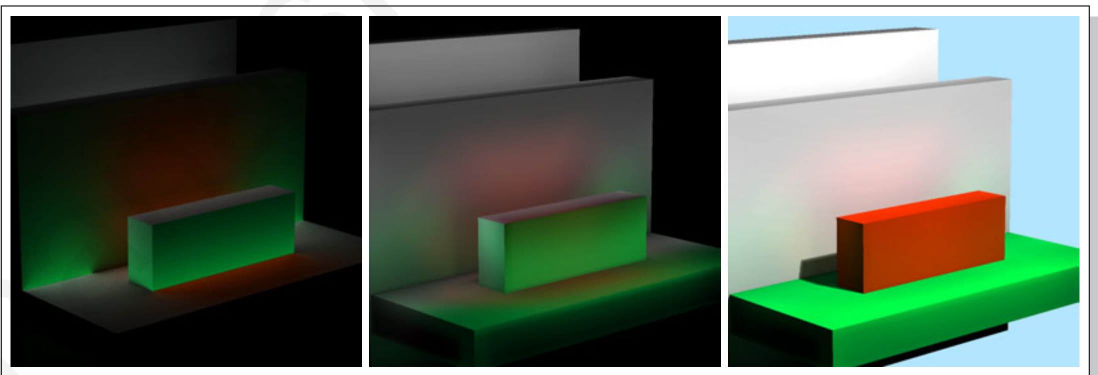


Figure 53: Test scene solution. From left to right: reference solution computed with ray-tracing (indirect illumination only), our solution (indirect illumination only) and final image with direct and indirect lighting.

Athanasios Gaitatzes

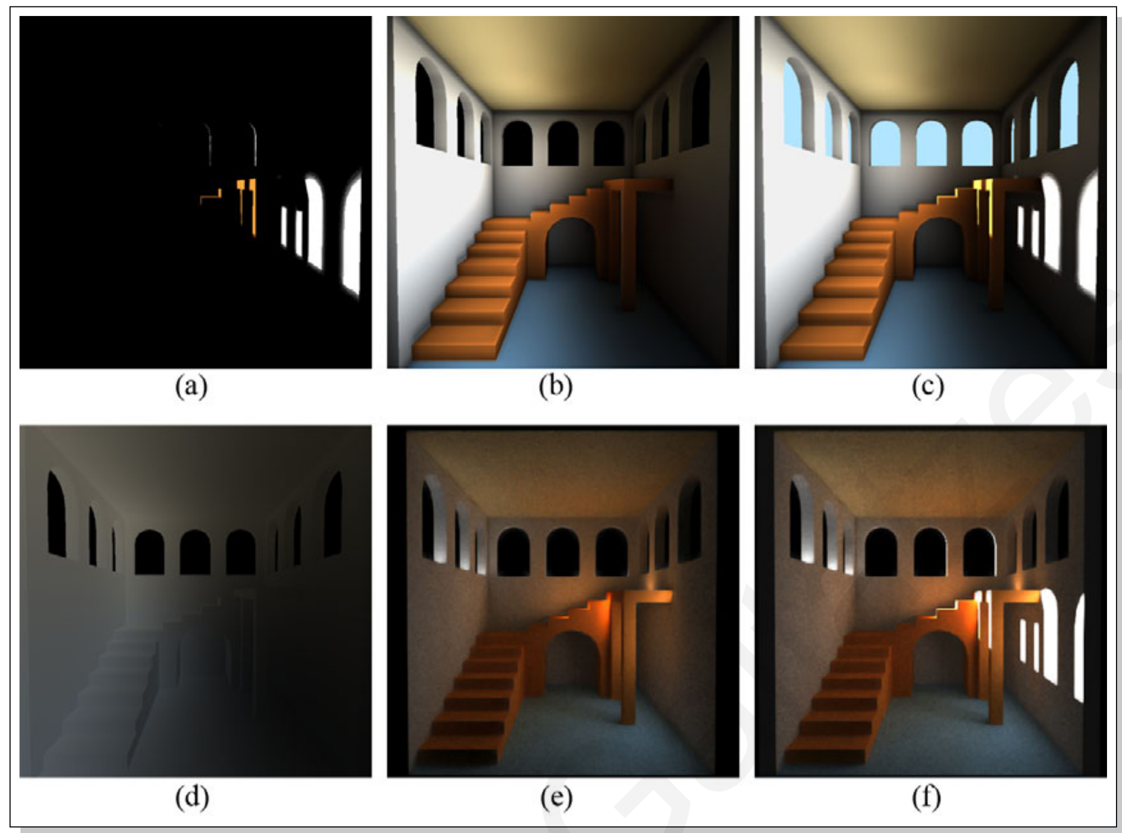



Figure 54: Room scene solution; (a) lit with Direct lighting only. (b) Radiosity with 64 iterations. (c) Direct and indirect illumination using our method. (d) The indirect illumination using light propagation volumes [54]. (e) Reference radiosity using 2-bounce path tracing. (f) Reference final image using path tracing.

method maintains easily an interactive frame-rate since the propagation stage takes only 61 ms to complete.

A nice characteristic of our method is the predictable performance of the propagation stage. We can easily calculate the time for the propagation step for each individual voxel. This time is constant and independent from the scene complexity. It should be noted of course that the performance may be constant and predictable, but the same is not true for the accuracy and the quality of the resulting illumination.

Athanasios Gaitatzes

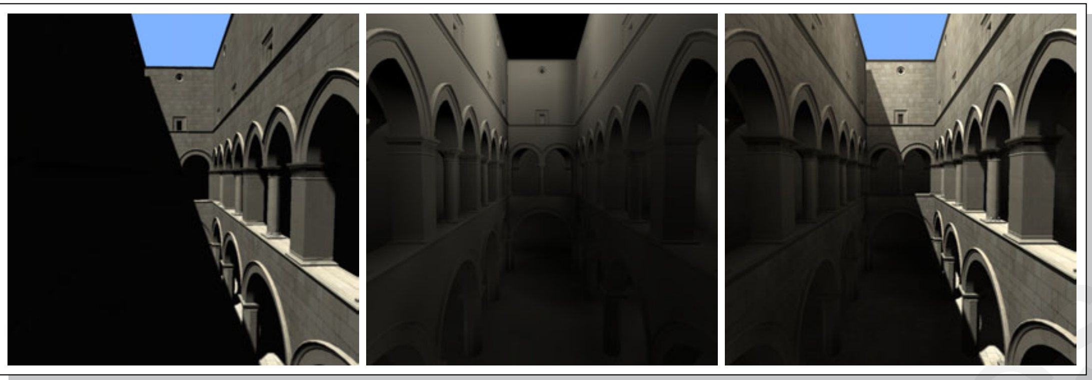


Figure 55: Sponza Atrium II scene solution. From left to right: direct lighting, indirect illumination only and final image with direct and indirect lighting.

Figure 55 shows the Sponza Atrium II, a typical scene in the global illumination literature. The right part of the scene is directly lit by the sun, the left one is lit only indirectly. As we can see, using only eleven propagation steps our method successfully reproduces the low-frequency indirect illumination which is dominant on the left part of the scene with very few visible artifacts.

Figure 56 shows an enclosed arena scene, a typical outdoor scene in video games. Twelve propagation steps are used in this case and we can see that the resulting indirect illumination greatly improves the visual quality of the final image.

Figure 56: Arena scene solution. From left to right: direct only lighting, indirect illumination using our method and final image with direct and indirect lighting.

Athanasios Gaitatzes

### 5.7 Discussion

A nice feature of our method is that for scenes with static or smoothly changing geometry and lighting conditions, the cost of the indirect illumination can be amortized among many frames without introducing any visible artifacts. In other words, the rate of indirect lighting updates can be reduced to meet the final performance goals. For scenes with good temporal coherence — hence, with slow illumination changes — it is possible to perform the 3-way voxelization in an interleaved manner (one direction per frame). In this case the volume is completely updated after three frames but the voxelization cost is reduced by a factor of three.

Since voxelization is a rough discretization of the scene geometry, secondary shadows from small scale geometric details cannot be reproduced accurately by our method. Higher voxel resolutions can always be used, but with a performance hit. Also, due to graphics hardware limitations, we only used second order spherical harmonics, which they do not have sufficient accuracy to represent high frequency indirect light. This is not crucial if the direct illumination covers large parts of a scene yielding only very low-frequency indirect shadows in the first place. Interestingly, imperfect shadow maps have exactly the same issue (but for different reasons) but we think that our

Figure 57: Knossos scene solution. From left to right: direct lighting, radiosity using our method and final image with direct and indirect lighting.

Athanasios Gaitatzes

method is preferable since it does not require the maintenance of a secondary point based scene representation and the performance is mostly independent from final image resolution.

The performance and quality of our method depends on two parameters: the volume resolution and the number of iterations. Empirically, we have found that a grid size of 32 is sufficient in most cases. For outdoor scenes we have found that a low iteration count (around 12) is sufficient but for indoor ones a much higher iteration count is required (around 64) to accurately simulate the bouncing of the light inside the building.

### 5.8 Summary

We have presented a new method for the computation of indirect diffuse light transport in dynamic environments in real-time using a discretization of the illumination in the scene. Unlike previous work, our method takes in to account indirect occlusion and secondary light bounces. We have demonstrated that our method gives good results in a variety of test cases and always maintains a high frame rate.

Since the test results showed that the voxelization step is relatively costly, in the future we intent to introduce a much faster voxelization scheme. Furthermore, the possibility of a more accurate but still manageable radiance deflection mechanism will be investigated. Finally, another interesting direction of research is to extend this method to take specular light transport in to account.

In the next chapter we will discuss the discretization of the scene geometry that will assist us in accelerating further more the method just presented.

Athanasios Gaitatzes

Athanasios Gaitatzes

# Part IV

## – Discretization of Geometry –

## Voxelization Methods

Athanasios Gaitatzes

Athanasios Gaitatzes

# Chapter 6

# Two Simple Single-pass GPU methods for Multi-channel Surface Voxelization of Dynamic Scenes

## 6.1 Motivation

An increasing number of rendering and geometry processing algorithms relies on volume data to calculate anything from effects like global illumination or visibility information. So a fast and efficient computation of this volume in real-time is imperative. Volume-based *Global illumination* uses an intermediate regular approximation of the geometry in order to store lighting and geometry data.

## 6.2 Overview

We present two real-time and simple-to-implement surface voxelization algorithms and a volume data caching structure, the Volume Buffer, which encapsulates functionality, storage and access similar to a *frame buffer object*, but for three-dimensional scalar data. The Volume Buffer can rasterize primitives in 3D space and accumulate up to 1024 bits of arbitrary data per voxel, as



Athanasios Gaitatzes

required by the specific application. The strengths of our methods is the simplicity of the implementation resulting in fast computation times and very easy integration with existing frameworks and rendering engines.

## 6.3 Introduction

In the previous chapter we presented a discretization of the illumination of the scene using a *Virtual Point Light* method. In this chapter we will discretize the scene geometry so that the *Global illumination* calculations become independent of the scene complexity. In addition, a volume-based technique provides access to the full-scene data in the local-only context of a shaded pixel which is important in computing *Global illumination* effects.

Volume representation of polygonal models is an important basic operation for many applications in computer graphics and related areas. Polygonal models, for example, have often been substituted by volume representations to remove unnecessary complexity for certain calculations, to provide a uniform sampling of the underlying data, to structure multi-resolution information in an easily and rapidly accessible manner or to enhance the models with additional data. Voxelization methods have been used in domains as diverse as global illumination computation [108], [55], fluids simulation [15] and ambient occlusion computation [81] [84], [73] [106] collision detection [66], procedural terrain generation [34] and rigid body simulation [42].

Surface voxelization describes the process of turning a scene representation consisting of discrete geometric entities (e.g. triangles) into a three-dimensional regular spaced grid that captures the surface of the scene. Each cell of the grid encodes specific information about the scene. In the case of binary voxelization, a cell represented by single bits in a bit-mask stores whether geometry is present in it or not. In a multi-valued voxelization, occupancy is extended to represent the (scalar) coverage of a voxel by the geometry and can also store additional spatial information.

Athanasios Gaitatzes

An increasing number of real-time rendering and geometry processing algorithms relies on volume data to calculate anything from global illumination approximations or visibility queries. We present a voxelization algorithm and volume data-caching structure, the Volume Buffer, which encapsulates functionality, storage and access similar to a *frame buffer object*, but for three-dimensional data. The Volume Buffer can rasterize primitives in 3D space and accumulate up to 1024 bits of arbitrary data per voxel, as required by the specific application, by using up to 8 floating point render targets, as necessary (where 8 is currently the maximum available number of MRTs). The strength of our method is the simplicity of the implementation (about 15 lines of geometry shader code and 1 line of pixel shader code - see Section 6.4.2) resulting in fast computation times and a very easy integration with existing engines and rendering frameworks.

Our multi-channel voxelization algorithm runs entirely on the GPU and can generate volume data from arbitrary complex and dynamic models in real-time. The proposed volume sampling technique is not limited to providing an occupancy volume representation of the scene, but also a complete attribute set for complex calculations (i.e. in global illumination calculations an albedo buffer, a normal buffer, a direct lighting buffer can be generated). This way heavy computations are disassociated from the surface representation data, thus making the method suitable for both primitive-order and screen-order rendering, such as deferred rendering. We do not require water-tight models nor is our method dependent on the depth complexity of the scene.

As real-time applications that utilize voxelization techniques increase lately, they can directly benefit from the use of our methods that offer fast discretization of complex polygonal representations.

Athanasios Gaitatzes

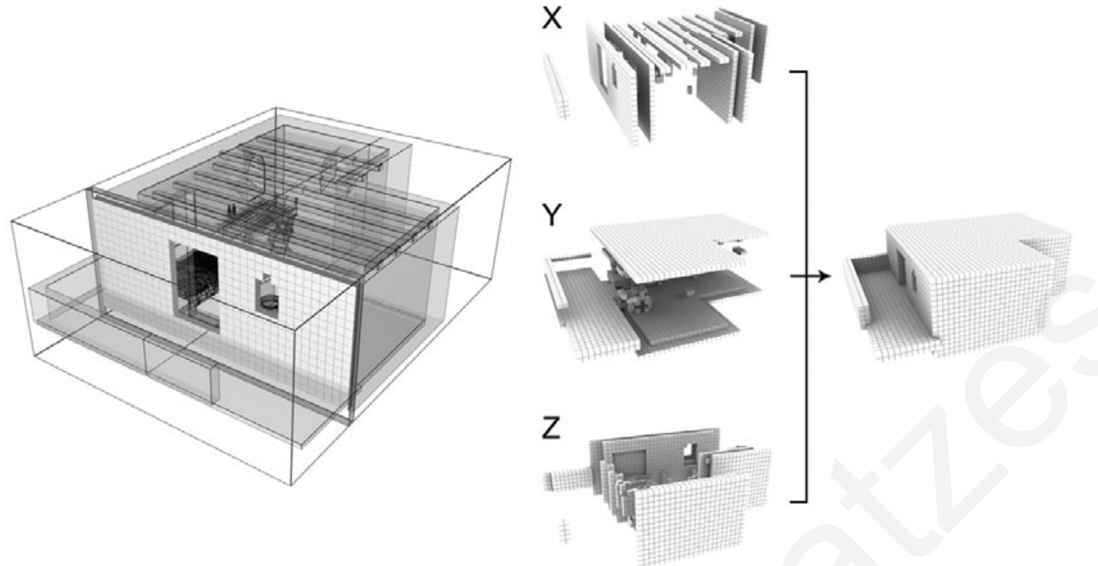



Figure 58: Axial voxelization pass (left) and composition of the three sub-volume passes into one voxelized volume (right).

### 6.4 Overview of Voxelization methods

The goal of our voxelization method is to reduce the rasterization and unnecessary clipping operations over the entire volume grid, while sending the geometry from host to device only once per slicing direction. To this end, we regard a slice-order voxel fragment generation along a principal axis.

A volume covering the extents of the scene is created and updated at every frame or whenever the environment changes. In order to sample the triangles coherently, we take three perpendicular volume sweep planes and each triangle is selectively rasterized only to the plane of maximum projection (i.e. to the direction where its normal is mostly aligned with). Therefore the primitives are rasterized *only once*. This ensures that the triangles' surface is densely sampled.

The main operation in both proposed voxelization methods is the clipping or "slicing" of the incoming triangles against the boundaries of each volume slice. The difference between the two

Athanasios Gaitatzes

methods is where the triangle clipping operation takes place. In the first method (see Section 6.4.1) each triangle is clipped against the current volume slice, in a geometry shader, allowing only the valid parts of the triangle to go through for rasterization. In the second method (see Section 6.4.2) each triangle is rasterized in each volume slice it intersects and the fragments are further clipped in the pixel shader.

The final volume is generated from the fusion of the three intermediate passes into a single multi-buffer (see Figure 58) by using the MAX blending operation. We substituted the OR operation commonly used in binary voxelization, as in our case, we deal with scalar data. Since all fragments have to be treated, face culling and z-test are disabled and hence no z-buffer is attached to the *frame buffer object*. All volume multi-channel attributes are computed and rendered simultaneously (e.g. occupancy, albedo, normals etc.) using *multiple render targets* into corresponding volume buffers (see Figures 59, 60 and 61).

We take advantage of the OpenGL® extension for layered rendering. It allows a geometry shader to write to the build-in special variable *gl_Layer* thus enabling the rendering of primitives to arbitrary volume texture layers computed at run time and eliminating the multiple passes over the incoming data or the restriction to record only a binary volume representation in one pass.

### 6.4.1 Geometry Shader Triangle Slicing

To assign the geometry (or parts thereof) to the appropriate buffer layer (see Figure 62) we intersect each triangle with the Eye Space Coordinates (ECS) of the volume slices of each axial sweep. If the triangle is aligned with the major axis of the specific pass, its vertices are sorted in ascending order. If the triangle is contained within one slice (Figure 63, Case A) the triangle is exported as is and the exit condition is met. Otherwise, we clip the triangle's edges against the planes perpendicular to the major axis (slice boundaries), producing a surface strip for each

Athanasios Gaitatzes

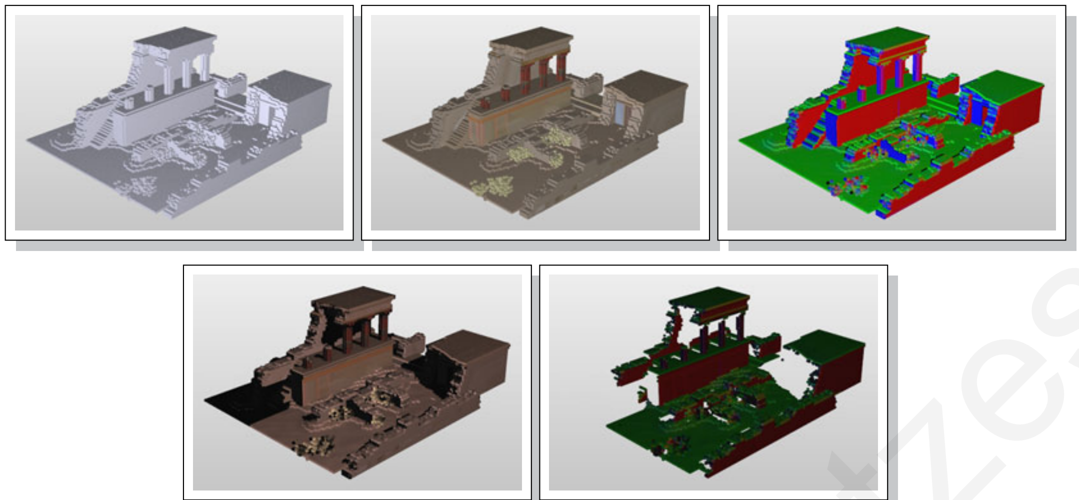


Figure 59: Voxelization of the Knossos model (109170 triangles) into a $128^3$ grid. The volumes in the order that they appear are the occupancy volume, the albedo volume, the normals volume and the lighting volume and the $2^{nd}$ order spherical harmonics volume of the direct illumination (R component).

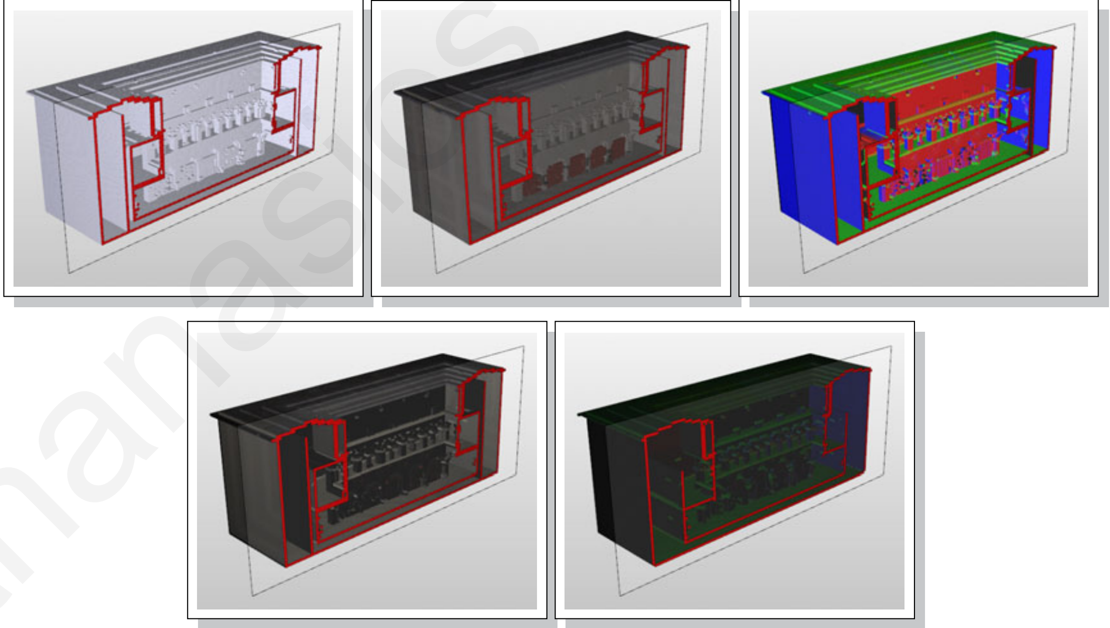


Figure 60: Voxelization of the Sponza II model (219305 triangles) into a $128^3$ grid; cross section depicted here.

Athanasios Gaitatzes

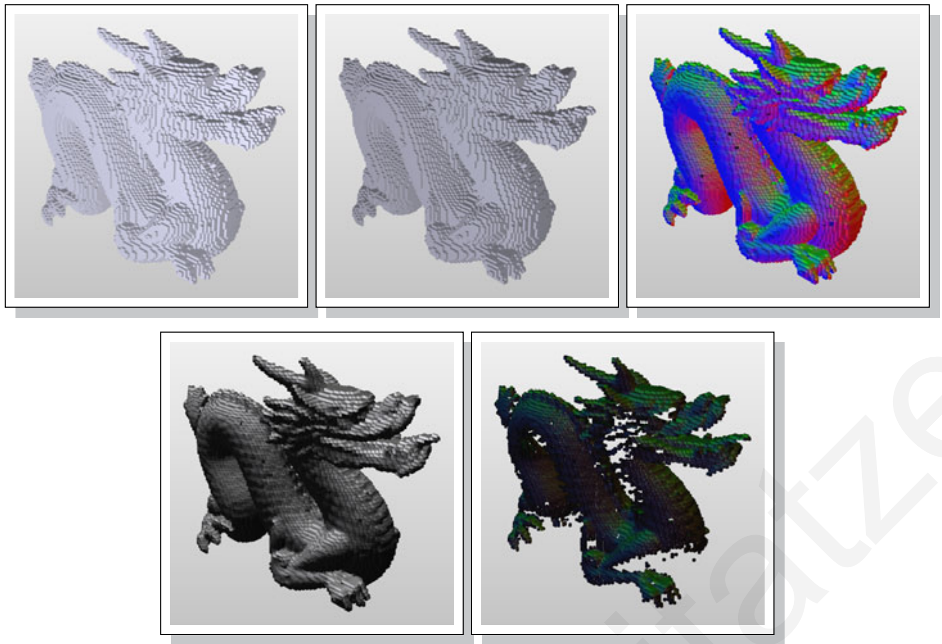


Figure 61: Voxelization of the Dragon model (871414 triangles) into a $128^3$ grid.

slice that the polygon intersects. At each step, we decide on the configuration of the triangle (see Figure 63) and whether a triangle split needs to occur or not. At each split, a quad-shaped triangle strip is being generated and rasterized to the appropriate volume layer (see Algorithm 5). The pixel shader is virtually empty, computing just the multi-channel data that an application requires.

We have expanded the simple for-loop construct, which could have been used in order to slice the model triangles, in order to achieve higher performance in the geometry shader.

### 6.4.2 Pixel Shader Fragment Clipping

The second algorithm is very simple as the geometry shader does not do any triangle clipping. Rather the method relies on fragment rejection in the pixel shader.

For each directional voxelization, each triangle in the scene passes from a geometry shader (see Algorithm 6) where, if it is aligned to the current sweeping direction, it is rasterized into *all* the volume slices that it intersects. The slice boundaries are computed in Normalized Device

Athanasios Gaitatzes

**Algorithm 5:** Geometry Shader used for triangle slicing (Z-Pass). (ECS: Eye Coordinate Space)

**Input**: $v1$, $v2$, $v3$ - the △ vertices
**Data**: z slice thickness (in volume sweep ECS)
**Result**: △ sliced into stripes and rasterized into the appropriate volume layer. New □ is emitted with generated vertices $v1L$, $v1R$, $v2L$ and $v2R$ per slice.

**if** △ *not aligned with Z-axis* **then** return

sort vertices according to Z-axis.
**if** *winding of* △ *is altered* **then**
    change order of emitted attributes

layer ← minimum slice index for the first vertex
slice ← current slice depth in ECS

**if** $v3$ *depth is* $\geq$ *slice* **then** CASE A
    Emit △ $v1$, $v2$, $v3 \rightarrow$ layer
    return

**if** $v2$ *depth is* $\geq$ *slice* **then** $v1L \leftarrow v1R \leftarrow v1$
**else** $v1L \leftarrow v2$ ; $v1R \leftarrow v1$

$v2L$, $v2R \leftarrow$ Intersect Edges (slice)
Emit □ $v1R$, $v1L$, $v2L$, $v2R \rightarrow$ layer

**repeat**
    slice += z thickness ; layer ++
    $v1L \leftarrow v2L$ ; $v1R \leftarrow v2R$
    **if** $v2$ *depth is* $\geq$ *slice* **then** CASE B
        $v2L$, $v2R \leftarrow$ Intersect Edges (slice)
    **else**
        **if** $v3$ *depth is* $<$ *slice* **then**
            **if** $v2$ *depth was* $\geq$ *slice* **then** CASE C
                $v2L \leftarrow v2$ ; $v2R \leftarrow v3$
            **else** CASE D
                $v2L \leftarrow v2R \leftarrow v3$
        **else**
            **if** $v2$ *depth was* $<$ *slice* **then** CASE E
                $v2L$, $v2R \leftarrow$ Intersect Edges (slice)
            **else** CASE F
                $v2L$, $v2R \leftarrow$ Intersect Edges ($v2$ depth)
                Emit □ $v1R$, $v1L$, $v2L$, $v2R \rightarrow$ layer
                $v1L \leftarrow v2L$; $v1R \leftarrow v2R$
                $v2L$, $v2R \leftarrow$ Intersect Edges (slice)
    Emit □ $v1R$, $v1L$, $v2L$, $v2R \rightarrow$ layer
**until** $v3$ *depth* $<$ *slice*

Athanasios Gaitatzes

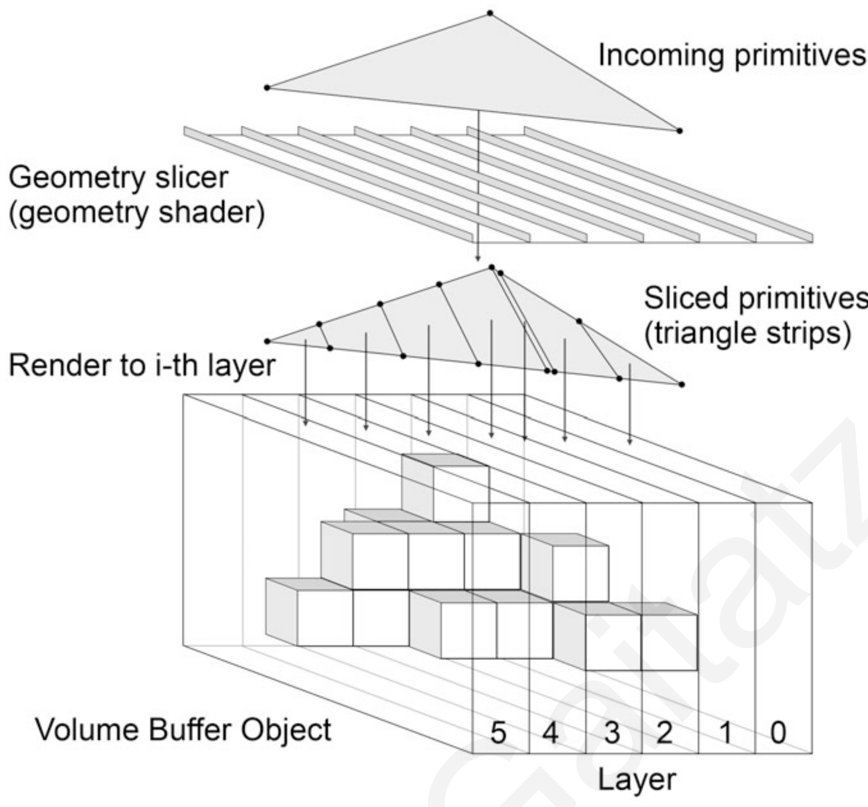



Figure 62: Geometry shader triangle slicing. The incoming triangles are sliced into stripes and each stripe is rasterized into the associated layer. (See Algorithm 5)

Coordinates (NDC) and passed to the pixel shader where fragments are *discarded* if their depth is outside these boundaries. The process is demonstrated in Figure 64.

### 6.5 Implementation

In order to create the data storage structure, we generate on the GPU a uniform spatial partitioning structure in real-time. For the voxelization, the user has the option to request several attributes to be computed and stored into floating point buffers for later use. Among them are surface attributes like albedo and normals, but also dynamic lighting information and radiance values

Athanasios Gaitatzes

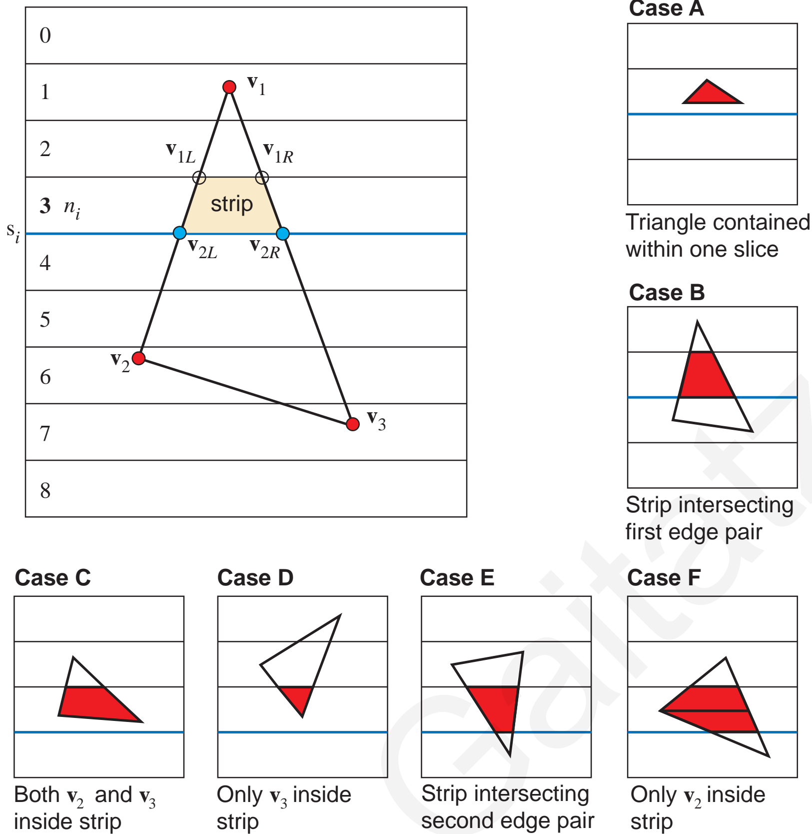



Figure 63: The six possible triangle strip configurations with respect to the volume grid. (See Algorithm 5)

in the form of low-order spherical harmonics (SH) coefficients representation (either monochrome radiance or separate radiance values per color band).

Each Volume Buffer is attached to a *frame buffer object* and through the *multiple render targets* mechanism, we store the user-requested attributes on all buffers simultaneously. For each voxel, the albedo is computed from the surface material information. The lighting information is determined using direct illumination, complete with shadows and emissive illumination. The radiance of the corresponding scene location is calculated and stored as a $2^{nd}$ order spherical harmonic representation for each voxel. For each color band, four SH coefficients are computed and

Athanasios Gaitatzes

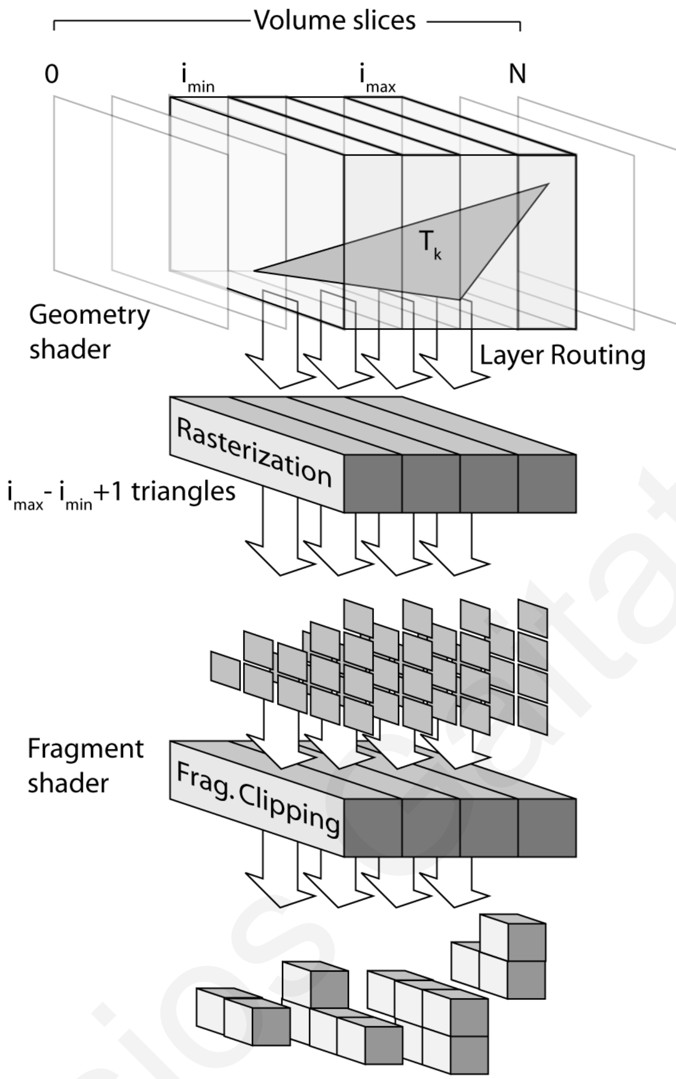



Figure 64: Pixel shader clipping method. The Geometry shader rasterizes each triangle into all the volume slices it intersects and the Pixel shader discards the fragments based on their depth (See Algorithm 6).

Athanasios Gaitatzes

encoded as RGBA float values, since the four SH coefficients map very well to the four component buffers supported by the graphics hardware. Since many works in the literature as well as practical implementations of shading algorithms rely on the spatial storage of radiance fields in the form of Spherical Harmonics, the later have been included in our test implementation in order to show the

**Algorithm 6:** Geometry and Pixel Shaders used for triangle rasterization (Z-Pass). (ECS: Eye Coordinate Space, NDC: Normalized device Coordinates).

**Input**: $v1$, $v2$, $v3$ - the △ vertices
**Data**: z slice thickness and z volume min (in volume sweep ECS)
**Result**: △ rasterized into the appropriate volume layer.

/* Geometry Shader */

flat out zMin, zMax // directed to pixel shader

**if** △ *not aligned with Z-axis* **then** return

$sliceMin \leftarrow min\ triangle\ z \quad z\ volume\ min/z\ slice\ thickness$
$sliceMax \leftarrow max\ triangle\ z \quad z\ volume\ min/z\ slice\ thickness$

**for** *slice between $sliceMin$ and $sliceMax$* **do**
  zMin ← min depth of $slice$ in NDC
  zMax ← max depth of $slice$ in NDC
  layer ← $slice$
  Emit △ $v1$, $v2$, $v3$ → layer

/* Pixel Shader */

flat in zMin, zMax

**if** *frag depth not between (zMin, zMax)* **then** discard
**else** write data to volume

applicability of our methods. The interested reader should refer to [90], [100] and [39] for further information.

### 6.6 Performance & Evaluation

We have integrated the multi-channel voxelization algorithm in a real-time deferred renderer using OpenGL® and the OpenGL® Shading Language (GLSL) [61]. We tested our methods for multiple models and various voxel grid resolutions. The results, reported in the following tables, were obtained on an Intel Core i7 860 at 2.8GHz with 8GB of RAM and equipped with an NVIDIA® GeForce GTX285 GPU with 1GB of video memory.

We compare our two methods based on the number of vertices that a geometry shader can output as the running times can vary greatly even for small changes to the number of requested output vertices. The number of vertices emitted from the geometry shader triangle slicing method

Athanasios Gaitatzes

| Model | Grid size | Grid actual | Memory (MB) | Geometry slicing vertices out 7 | 11 | 15 | 19 | Pixel clipping vertices out 6 | 9 | 12 | 15 | Voxels (K) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bunny | $64^3$ | 53 × 64 × 41 | 1.06 | 1.74 | 1.79 | 2.03 | 2.51 | 1.13 | 1.15 | 1.28 | 1.58 | 5.3 |
| | $128^3$ | 106×128× 82 | 8.49 | 2.37 | 2.38 | 2.66 | 3.19 | 1.71 | 1.72 | 1.86 | 2.16 | 22 |
| 69451 | $256^3$ | 213×256×165 | 68.64 | 5.92 | 5.97 | 6.43 | 6.98 | 4.92 | 4.98 | 5.12 | 5.48 | 89.6 |
| triangles | $512^3$ | 425×512×330 | 547.85 | 28.2 | 28.6 | 29.3 | 30.1 | 26.9 | 27.3 | 27.5 | 27.8 | – |
| Knossos | $64^3$ | 52 × 23 × 64 | 0.58 | 3.04 | 3.09 | 3.39 | 3.98 | 1.82 | 1.84 | 2.03 | 2.45 | 9.7 |
| | $128^3$ | 104× 46 ×128 | 4.67 | 3.85 | 3.86 | 4.26 | 4.92 | 2.45 | 2.46 | 2.68 | 3.14 | 45 |
| 109168 | $256^3$ | 208× 93 ×256 | 37.78 | 6.46 | 6.55 | 6.95 | 7.71 | 4.86 | 4.91 | 5.11 | 5.60 | 196 |
| triangles | $512^3$ | 416×185×512 | 300.63 | 20.88 | 20.94 | 21.59 | 22.58 | 18.33 | 18.43 | 18.75 | 19.28 | 800 |
| Sponza II | $64^3$ | 39 × 27 × 64 | 0.51 | 3.99 | 4.03 | 4.52 | 5.38 | 2.88 | 2.91 | 3.24 | 4.01 | 20 |
| | $128^3$ | 79 × 54 ×128 | 4.17 | 5.10 | 5.13 | 5.78 | 6.74 | 3.53 | 3.57 | 3.94 | 4.79 | 100 |
| 219305 | $256^3$ | 157×107×256 | 32.81 | 8.38 | 8.40 | 9.26 | 10.66 | 5.93 | 6.02 | 6.48 | 7.51 | 445 |
| triangles | $512^3$ | 315×214×512 | 263.32 | 21.92 | 22.01 | 23.03 | 24.86 | 18.44 | 18.64 | 19.23 | 20.51 | 1980 |
| Dragon | $64^3$ | 64 × 62 × 29 | 0.88 | 69.1 | 70.6 | 71.3 | 72.0 | 70.0 | 71.2 | 74.1 | 74.9 | 5.4 |
| | $128^3$ | 128×123× 57 | 6.85 | 75.4 | 75.8 | 76.0 | 76.3 | 75.1 | 75.4 | 75.8 | 76.2 | 22.5 |
| 871414 | $256^3$ | 256×247×114 | 55.00 | 77.1 | 77.5 | 77.8 | 78.3 | 76.9 | 77.3 | 77.6 | 78.0 | 93 |
| triangles | $512^3$ | 512×493×229 | 441.00 | 88.1 | 88.7 | 89.4 | 90.3 | 88.3 | 89.1 | 89.6 | 90.4 | – |

Table 3: Running time (in ms) for the construction of a half-float (16bit) single channel Occupancy Volume buffer for the two surface voxelization methods, based on the number of vertices that the geometry shader outputs. The third column gives the actual grid sizes as tight volume grids are generated dynamically. The last column reports the number of the resulting voxels.

Athanasios Gaitatzes

is $3 + 4n$ (we detect the emittance of a triangle and do not produce a degenerate quad) and from the pixel shader clipping method is $3n$, where $n$ is the number of slices that a triangle spans.

We observe (see Table 3) that both methods have approximately the same running speed and produce the same number of voxels. The pixel shader clipping method achieves slightly better results but when the number of triangles that need to be processed increases (i.e. Dragon model) then the two methods are equivalent.

The quality of the voxelization depends on the number of volume slices each triangle spans. The smaller the limit of output vertices of the geometry shader, the higher the probability that the triangle will be partially sliced, resulting in empty voxels. However, due to the fact that triangles are selectively processed in the volume sweep plane of maximum projection, this is seldom the case.

Figures 59 60 61 depict the multi-channel voxelization of several models, (closed surfaces but also open environments with no watertight surfaces).

In Figure 65 we visually compare the voxelization of the two methods. The difference (red / green voxels) is attributed to the rasterization process even though a conservative approach did not yield better results probably because our tested models did not have any sub-pixel triangles.

In Figure 66 we compare the voxelization of the pixel shader clipping method for various geometry shader output vertices. If we request too few output vertices from the geometry shader (eg. 3 vertices) then holes start to appear in the voxelization. A higher output vertex count gradually remedies this issue. In many effects, such as in the case of diffuse global illumination (e.g. virtual point light injection), a high vertex output limit would not be necessary, since even a sparse or incomplete volume representation can still work satisfactorily due to the low-frequency nature of secondary diffuse light transport. On the contrary, fluid effects (e.g. water) require a higher

Athanasios Gaitatzes

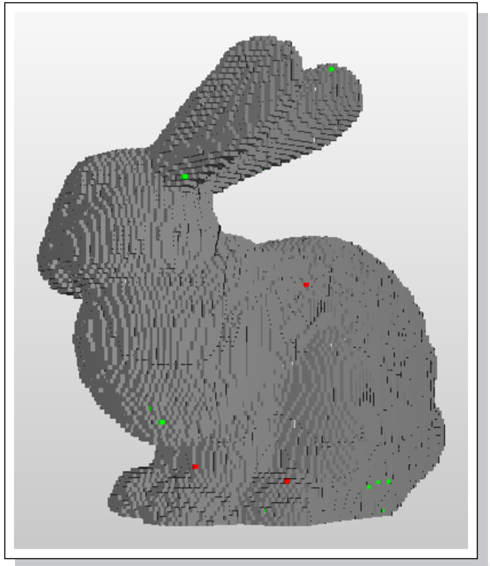


Figure 65: Visual comparison of the geometry shader triangle slicing method and the pixel shader clipping methods for the bunny model at $128^3$ volume resolution. Result with 11 output vertices. Gray voxels are common to both method variations. Green voxels are only present in the geometry shader triangle slicing, while red voxels are only generated by the pixel shader clipping. The total number of different voxels amounts to 33 which is about 0.15% of variation.

vertex output limit in order to produce dense volumes as the granularity of the volume affects the simulation as a whole.

The results were acquired using the most naive draw method, namely display lists for caching of the OpenGL® commands. We consider arbitrary dynamic polygonal models and we assume that we are drawing a triangle soup.

Apart from the occupancy buffer, where virtually no computations are involved, the construction speed of the rest of the buffers depends on the computations that are involved in their creation. Table 4 lists the minimum required time to *write* to 1, 2 or 3 *multiple render targets* without performing any computations.

Athanasios Gaitatzes

| Grid size | Geometry slicing | | | Pixel clipping | | |
|---|---|---|---|---|---|---|
| | 15 vertices output | | | 9 vertices output | | |
| | MRTs used | | | MRTs used | | |
| | 1 | 2 | 3 | 1 | 2 | 3 |
| $64^3$ | 2.36 | 2.54 | 2.70 | 1.33 | 1.33 | 1.52 |
| $128^3$ | 3.22 | 4.12 | 5.18 | 2.90 | 2.90 | 3.96 |
| $256^3$ | 10.2 | 17.4 | 25.1 | 15.8 | 15.9 | 24.3 |

Table 4: Comparison of the running time (in ms) for the bunny model for a floating (32bit) four channel buffer and different sizes of *multiple render targets* (MRTs).

For the sake of comparative examination (see Table 5), we implemented a version of the method by Fang et al. [25]. Their algorithm, using a variation of the XOR slicing method, renders the geometry once for each slice of the volume grid, each time restricting the clipping volume to this slice. We implemented a modified version of the method, that supports multi-channel data and renders the geometry only once. The main difference is that we do not use the originally proposed XOR operation because in practice, very few models are watertight and many volume attributes

Figure 66: Comparison of the voxelization using the pixel shader clipping method at $256^3$ volume resolution with 3, 6, 9 and 12 geometry shader vertices output. The number of voxels produced are 52382, 87690 and 89696 (complete voxelization) for 3, 6, 9 and above geometry shader output vertices, respectively.

Athanasios Gaitatzes

| Model | Grid | Fang et al. | | Eisemann et al. | |
|---|---|---|---|---|---|
| | size | **time (ms)** | **# voxels** | **time (ms)** | **# voxels** |
| Bunny | $64^3$ | 20.3 | 5.5K | 0.171 | 2.1K |
| | $128^3$ | 40.8 | 22.2K | 0.174 | 18.6K |
| (69451 tris) | $256^3$ | 83.5 | 90.3K | 0.21 | 145.4K |
| | $512^3$ | 181 | – | 0.61 | 1124K |
| Knossos | $64^3$ | 49.9 | 9.8K | 0.42 | 2.8K |
| | $128^3$ | 99.8 | 45.7K | 0.44 | 26.9K |
| (109168 tris) | $256^3$ | 201 | 198.5K | 0.51 | 190.3K |
| | $512^3$ | 409 | 823.6K | 0.83 | 151K |
| Sponza II | $64^3$ | 77.4 | 20.2K | 0.73 | 6.6K |
| | $128^3$ | 155 | 102K | 1.09 | 60K |
| (219305 tris) | $256^3$ | 310 | 452.3K | 2.53 | 408.3K |
| | $512^3$ | 629 | 2090K | 7.11 | 3283K |
| Dragon | $64^3$ | 3206 | 5.7K | 35.1 | 1.5K |
| | $128^3$ | 7710 | 23.2K | 35.3 | 11.2K |
| (871414 tris) | $256^3$ | – | 94.5K | 36.5 | 91.9K |
| | $512^3$ | – | – | 38.8 | 739.6K |

Table 5: Comparison of the running time (in ms) and number of voxels produced by different approaches.

cannot be defined for interior voxels. The big difference in the running times is attributed to the number of passes that Fang et al. do over the geometry data which increases their timings especially for large models. As per the quality of the voxelization we produce approximately the same number of voxels.

In addition we show the timing results for our implementation of Eisemann et al. [22] binary occupancy-only voxelization method but from three viewpoints instead of one in order to reduce the number of holes produced. Still, even though the running time is the fastest of all, the quality of the results is not very good. We attribute this to the fact that multiple voxelizations from different directions assumes (in our implementation) a cube as a bounding volume of the scene, wasting lots of empty space and reducing the number of useful voxels. In addition the method cannot take into

Athanasios Gaitatzes

| GPU | Geometry slicing | Pixel clipping |
|---|---|---|
| | 15 vertices output | 9 vertices output |
| G 105M | 51.2 ms | 30.8 ms |
| GTS 9800M | 13.5 ms | 6.33 ms |
| GTX 285 | 2.66 ms | 1.72 ms |

Table 6: Comparison of the running time on various types of hardware of the bunny model at $128^3$ resolution.

account partial occupancy or transparency. Our method works with an arbitrary and thus tighter bounding box.

Table 6 shows the improvement in the running time of our methods on different GPUs.

### 6.7 Discussion

The decision for the choice of method depends mostly on the GPU architecture. Implementations for non-unified architectures may favor the geometry shader approach (see Section 6.4.1), if the pixel shader cores are intensively used and vice versa. For unified architectures, the expected load in terms of triangle count is indicative of the best approach. Furthermore, certain GPU implementations do not favor the execution of complex geometry shaders with large output primitive streams. Our pixel shader approach (see Section 6.4.2) is very simple to implement but produces a lot of fragments in the geometry shader. These are rejected in the pixel shader but on architectures with small bandwidth, this could be an issue. It is a reason to favor the first method which produces exactly the fragments that are going to be rasterized in the final volume slices.

For scenes with slow or gradual animations, the three directional voxelization steps could be interleaved, recalculating only one axis pass in each frame, further reducing the volume buffer creation time by a factor of 3.

Athanasios Gaitatzes

| Model | Grid size | half-float buffer (16bit) | float buffer (32bit) |
|---|---|---|---|
| Bunny | $64^3$ | 2.03 ms | 2.04 ms |
| | $128^3$ | 2.66 ms | 2.80 ms |
| | $256^3$ | 6.43 ms | 7.25 ms |

Table 7: Comparison of the running time for 16- and 32-bit floating point buffers and 15 geometry shader vertices output.

A general improvement in many volume generation techniques, applicable in our method as well is, in order to reduce the time to construct the data structure one could also sort the scene primitives into two sets of static and dynamic geometry. This way, two volume buffers would be created, one for static and one for dynamic geometry. The static volume buffer could be created once and would not be updated again. The dynamic volume buffer would get updated at each draw frame. In practice, most parts of a three-dimensional environment are static and therefore, the respective volume buffer would only be updated after a triggered event of a change in one of the light sources. This method can alleviate the constant updates of a more generic volume buffer at the expense of extra texture space to store a separate volume buffer for the static geometry.

For some applications using a 32-bit floating point buffer might be too large for storing external data. In that case, a 16-bit buffer could be used with negligible quality degradation but higher performance. (as can be seen in Table 7).

On current hardware with OpenGL® implementations with version less than 4.0 the user cannot set the *BlendEquation* of each sub-buffer (ColorAttachment) individually. As a result we had to choose one *BlendEquation* for all four sub-buffers. We chose the GL_MAX operator which would compute the correct results for the albedo and the normals buffers. For the lighting and

Athanasios Gaitatzes

SH buffers it would be ideal to use the GL_FUNC_ADD operator that would produce additive blending results which would of course be correct when multiple lights were present in the scene.

The geometry shader architecture requires that triangle $n$ is processed after triangle $n - 1$ has completed its processing. New geometry shader features include instancing, which provides a performance increase when the order of primitives in the stream doesn't matter. As OpenGL® 4.0 hardware becomes pervasive, these limitation will be overcome.

Finally for rasterization based voxelizations the use of intermediate buffers is unavoidable when using the GPU. For voxel grids of arbitrary size (per dimension), current hardware architectures do not allow the viewport transformation to be part of the programmable pipeline. The geometry shader output must be in clip space coordinates (CSC) against which the driver will perform polygon clipping. As a result we need three viewport transformations that direct the triangle fragments to the appropriate volume grid slice. In addition, in voxel grids of equal dimension where the above problem is mitigated, the layered rendering mechanism requires layers to be parallel to each other enforcing again three axial passes of the voxel space.

### 6.8 Summary

We presented two methods for the surface discretization of dynamic scenes which along with the discretization of the illumination in an environment (previous chapter) achieve real-time performance for the simulation of *Global illumination* effects in dynamic environments where both the scene geometry and the lighting conditions can change.

The two strong points of the methods are the ability to generate multi-channel data at high performance and their simplicity in implementation and integration into existing frameworks in order to create anything from effects like global illumination or visibility computations.

Athanasios Gaitatzes

In the next chapter we will discuss the discretization of the scene geometry in image space that will improve the overall image quality of image-based *Global Illumination* methods.

Athanasios Gaitatzes

Athanasios Gaitatzes

# Chapter 7

## Incremental Image-based Multi-valued Voxelization for Global Illumination

### 7.1 Motivation

An increasing number of rendering and geometry processing algorithms relies on volume data to provide fast access to a uniform sampling of the geometry from any stage in the graphics pipeline. Recently, volume representations have been extensively used for the simulation of global illumination effects and fast view-dependent techniques have been also implemented in commercial graphics engines.

### 7.2 Overview

We introduce the concept of *Incremental Voxelization* for the multi-valued, scalar volume rasterization of fully dynamic scenes (geometry, materials and lighting) and demonstrate its application in the context of volume-based global illumination. Where current image-based voxelization algorithms repeatedly regenerate the volume using the deferred geometry image buffers of a single frame, the proposed method incrementally updates the existing voxels using a depth-buffer



Athanasios Gaitatzes

re-projection scheme and therefore, produces a more complete voxelization of the scene. *Incremental Voxelization* can be used for multi-attribute volumes and complex dynamic scenes. It offers improved quality and stability over non-incremental image-based methods at a very small overhead. Furthermore, image-based volume updates can be distributed across time as well as space, achieving a controllable cost amortization of existing voxelization techniques.

### 7.3 Introduction

In the previous chapter we presented a discretization of the full scene geometry so that the *Global illumination* calculations become independent of the scene complexity. In this chapter we will discuss an image-based approach that improves previous non-incremental approaches by producing a more complete volume representation of the scene geometry thus producing more accurate *Global illumination* effects.

Volume representation of polygonal models is an important basic operation for many applications in computer graphics and related areas. Polygonal models, for example, have often been substituted by volume representations to remove unnecessary complexity for certain calculations, to provide a uniform sampling of the underlying data, to structure multi-resolution information in an easily and rapidly accessible manner or to enhance the models with additional data. Voxelization methods have been used in many domains of computational science, engineering and computer graphics with applications ranging from global illumination simulation [108], [55], fluids simulation [15] and ambient occlusion computation [81], collision detection [66], procedural terrain generation [34] and rigid body simulation [42].

Real-time voxelization has reached a point where the binary (occupancy) volume representation of a 100K-triangle model can be generated in less than 5 ms at a resolution of $256^3$. However, when the per frame time budget is limited due to other, more important operations that must take

Athanasios Gaitatzes

place while maintaining a high frame rate, the fidelity of full-scene voxelization has to be traded off for less accurate but faster techniques. This is especially true for video game applications, where many hundreds of thousands of triangles must be processed in less than 2-3ms.

Among other applications, an increasing number of techniques for real-time global illumination effects rely on volume data, as they allow the fast, out-of-order access to spatial data from any deferred shading graphics pipeline stage as in Thiedemann et al. [108], Mavridis et al. [72] and Kaplanyan et al. [55]. Typical volumes produced for global illumination and other effects include multiple scalar attributes, such as albedo, spherical harmonics coefficients of incident light, normal vectors etc. Common to these techniques is the use of a deferred shading strategy to render and capture image-based spatial data, such as geometric occlusion (i.e. occupancy). Typically, data from the virtual camera view are stored in multiple deferred rendering targets (MRTs, e.g. depth, albedo, normals, lighting etc.) and similar data are obtained from the view point of one or more light sources (RSMs by Dachsbacher et al. [18]). These data are subsequently combined and embedded (or injected) in the volume representation of the scene, as for instance in Kaplanyan et al. [55] and [56].

Image-based volume generation methods provide very fast and guaranteed response times compared to geometry-based techniques but suffer from view dependency. More specifically, any technique that is performed entirely in image-space (as in deferred shading) considers only geometry that has been rendered into the depth buffer and thus has the following strong limitations. First, it ignores geometry located outside the field of view and second it ignores geometry that is inside the view frustum but is occluded by other objects. Yet these geometry parts may have a significant influence to the desired final result (see for example our indirect illumination case study). Essentially, in single-frame image-based voxelization, the only volume samples that can be produced in each frame are the ones that are visible in at least one of the images available

Athanasios Gaitatzes

in the rendering pipeline (view camera MRTs and light RSMs). Each time the (camera or light) view changes, a new set of sample points become available and the corresponding voxels are generated from scratch to reflect the new available image samples. Thus, the generated volume will never contain a complete voxelization of the scene. This leads to significant frame-to-frame inconsistencies and potentially inadequate volume representation for the desired volume-based effect, especially when the coverage of the scene in the available image buffers is limited.

On the other hand, adopting a more robust, full-scene geometry- or slicing-based voxelization cannot result in a predictable and - most importantly - controllable upper limit in the voxelization time for a particular volume granularity, across different 3D data. However, this is a critical aspect in the design of modern real-time rendering engines, where the time that can be dedicated to voxelization is usually less than the required time of current full-scene multi-channel voxelization methods.

To alleviate the problems of image-based voxelization techniques, but maintain their benefit of controllable, fixed execution time relative to full-scene volume generation methods, we introduce the concept of *Incremental Voxelization* and a corresponding incremental image-based volume generation algorithm. The volume representation is progressively updated and improved to include the newly discovered voxels and discard the set of invalid voxels, which are not present in any of the current image buffers (see Figures 67 and 68). Using the already available camera and light source buffers, a combination of volume injection and voxel-to-depth-buffer re-projection scheme continuously updates the volume buffer and discards invalid voxels, incrementally constructing the final voxelization. The buffers used are common to most deferred real-time rendering engines, i.e. view camera G-buffers (depth, albedo, normals) plus direct lighting and the corresponding light RSM buffers. Thus, the voxelization does not incur any additional rasterization cost, as it simply re-uses existing information.

Athanasios Gaitatzes

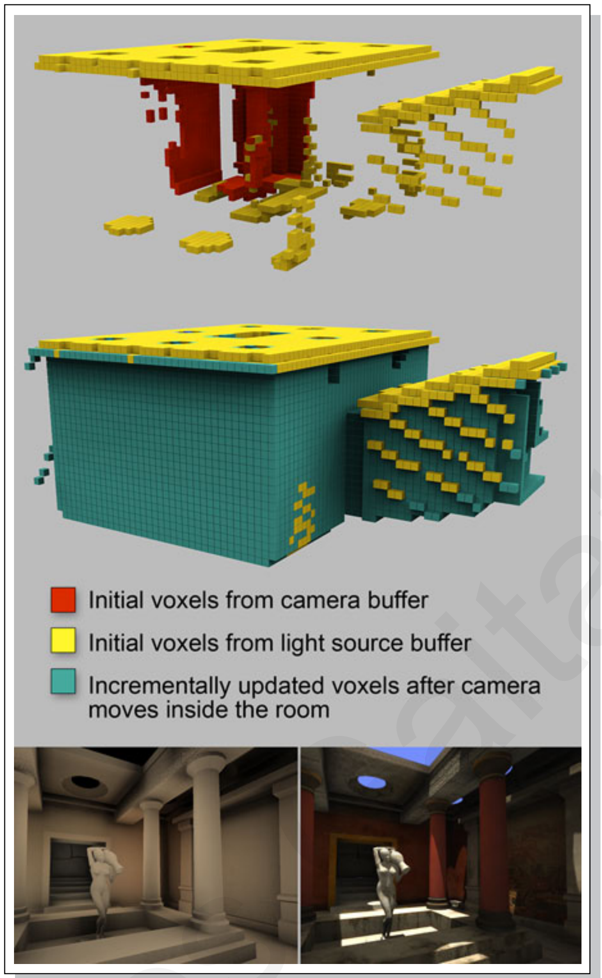



Figure 67: Top: Image-based voxelization after one step of the process having injected the camera and light buffers. Middle: Voxelization of the scene after the camera has moved for several frames. Bottom: Example of resulting global illumination.

The algorithm is lightweight and operates on complex dynamic environments where geometry, materials and lighting can change arbitrarily. As is the case with most image-based techniques, the algorithms' performance is largely independent of geometric complexity and is dominated by the fill rate of the target volume buffers. The accuracy and spatial coverage of the voxelization depends on the specific trajectories of the camera and lights but improves as the user interacts with the environment.

Athanasios Gaitatzes

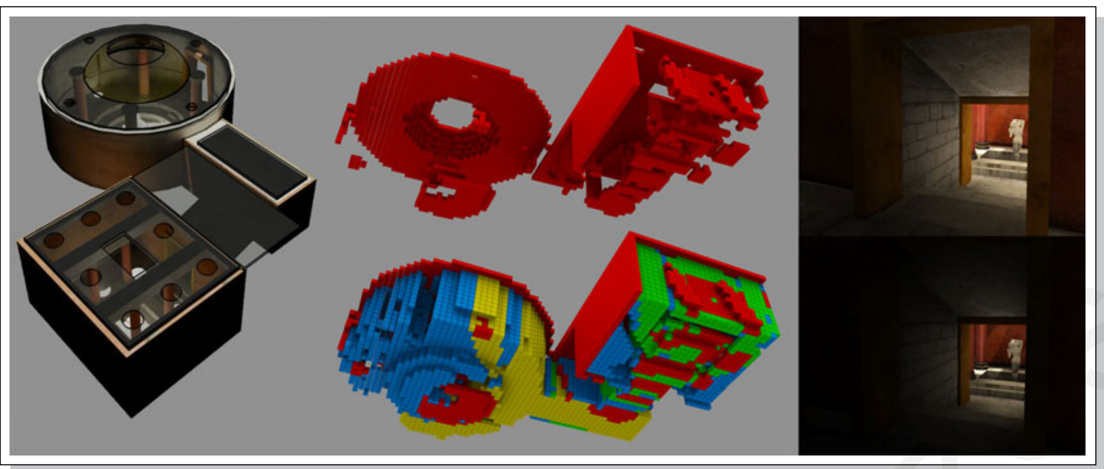


Figure 68: *Incremental Voxelization* (IV) of a scene. Red voxels correspond to image-based voxelization using image buffers from the current frame only, while other colors refer to voxels generated during previous frames using IV. Right: Volume-based global illumination results using the corresponding volumes. IV achieves more correct occlusion and stable lighting.

Compared to single-frame image-based voxelization, our method provides:

- Improved volume coverage (completeness) over non-incremental methods, as demonstrated by our global illumination case study, while maintaining its high performance merits.
- The ability to perform lazy or spatially and temporally scattered volume updates, thus further amortizing the voxelization cost among the frames.

With respect to full-scene multi-valued voxelization solutions, the benefits of incremental voxelization include:

- Predictable, controllable and bound execution time.
- Reuse of data that are already available to a real-time deferred shading application; thus the voxelization does not incur any additional rasterization cost.

It is important to note that full-scene voxelization methods, especially geometry-based ones, are in general more robust and accurate. However, our algorithm offers a flexible solution,

Athanasios Gaitatzes

performance-wise, that helps bridge the gap between the image-based and full-scene volume generation classes of techniques. Incremental voxelization targets mostly rendering engines and deferred shading frameworks rather than stand-alone volume generation applications and frameworks, whose demands in terms of accuracy are completely different. Still, as demonstrated by the Hausdorff distance evaluation of our results relative to the initial geometry (Table 9), the representation accuracy is more than satisfactory for most use scenarios.

We demonstrate our technique by applying it as an alternative voxelization scheme for the light propagation volumes diffuse global illumination method of Kaplanyan et al. [55]. However, being a generic multi-attribute scalar voxelization method, it can be used in any other real-time volume generation problem.

### 7.4 Overview of Voxelization method

When only occupancy of volume cells is required, geometric binary voxelization algorithms such as Eisemann et al. [22] provide a sufficiently accurate and fast solution. However, many techniques or measurements, such as global illumination estimation methods, rely on scalar or multidimensional data.

In order to be able to take advantage of the merits of an image-based volume data generation scheme (simplicity, geometry-independence, speed) and at the same time be able to produce more stable and valid volume data, we propose an *Incremental Voxelization* scheme based on the camera multiple render targets (MRTs), the light source RSM buffers [18] or any other available buffer that includes depth information and the volume representation attributes. As the user interacts with the environment, dynamic objects move or light information changes, new voxel data are accumulated into the initial voxelization data structure and old voxels are invalidated or updated if their projection in any of the image buffers proves inconsistent with the available recorded depth.

Athanasios Gaitatzes

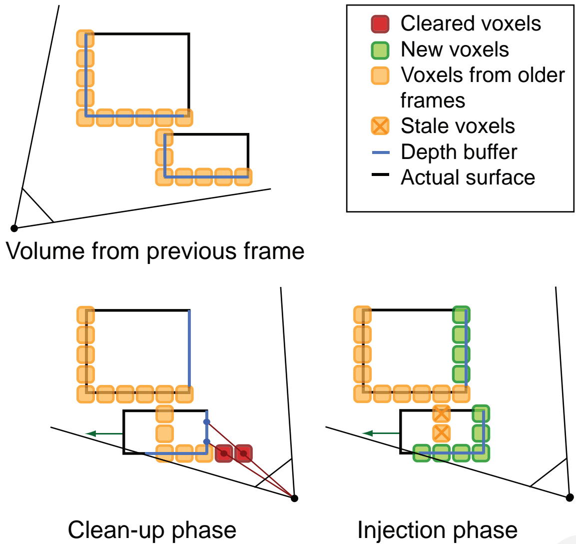



Figure 69: Schematic overview of the algorithm. During the cleanup phase each voxel is tested against the available depth images. If the projected voxel center lies in front of the recorded depth, it is cleared; otherwise it is retained. During the injection phase, voxels are "turned-on" based on the RSM-buffers and the Camera-based depth buffer.

In each frame, two things occur: First, in a *cleanup* stage, the volume is swept voxel-by-voxel and the center of each voxel is transformed to the eye-space coordinate system of the buffer and tested against the available depth image value, which is also projected to eye-space coordinates. If the voxel lies closer to the image buffer viewpoint than the recorded depth, the voxel is invalidated and removed. Otherwise, the current voxel attributes are maintained. The update of the volume is performed by writing the cleared or retained values into a separate volume in order to avoid any atomic write operations and thus make the method fast and a very broadly applicable one. At the end of each cleanup cycle, the two volume buffers are swapped. After the cleanup phase, samples from all the available image buffers are injected into the volume (similar to the LPV method [54]).

When multiple image buffers are available, the cleanup stage is repeated for each image buffer, using the corresponding depth buffer as input for voxel invalidation. Each time, the currently

Athanasios Gaitatzes

**Algorithm 7:** *Incremental Voxelization* pseudo-code using Camera G-buffers.

$V_{prev} \leftarrow$ current attribute volume $V_{curr}$
**foreach** *depth buffer* $Z_i$ **do**
  **foreach** *voxel* $\mathbf{p}_v \in V_{curr}$ **do**
    $V_{curr}(\mathbf{p}_v) \leftarrow$ Cleanup $(Z_i, V_{prev}, \mathbf{p}_v)$
**foreach** *attribute & depth buffer pair* ($\mathbf{a}_i$, $Z_i$) **do**
  Inject i-*th* buffer in $V_{curr}$

/* returns the updated attribute $\mathbf{a}$ at $\mathbf{p}$ */ **function** Cleanup (depth buffer $Z$, previous attribute volume $V$, voxel position $\mathbf{p}$)
  $\mathbf{a}_v \leftarrow V(\mathbf{p})$
  $\mathbf{p}' \leftarrow$ projection of $\mathbf{p}$ to eye-space
  $z_e \leftarrow Z(\mathbf{p}')$ depth value projected to eye-space
  **if** $\mathbf{p}'_z > z_e + b$ **then** return $\mathbf{0}$
  **else** return $\mathbf{a}_v$

updated (read) and output (written) buffers are swapped. The current image buffer attributes are then successively injected in the currently updated volume. The whole process is summarized in Figure 69 and in Algorithm 7.

### 7.4.1 Clean-up phase

Throughout the entire voxelization process, each voxel goes through three state transitions: "turn-on", "turn-off" and "keep". The "turn-on" state change is determined during the injection phase. During the clean-up stage we need to be able to determine if the state of the voxel will be retained or turned off (cleared). For each one of the available depth buffers, each voxel center $\mathbf{p}_v$ is transformed to eye-space coordinates $\mathbf{p}'_v$; accordingly the corresponding image buffer depth $Z(\mathbf{p}')$ is transformed to eye-space coordinates ($z_e$) using the equation:

$$z_e = \frac{2 \cdot n \cdot f}{(f \quad n) \cdot z_{NDC} \quad (f + n)} \tag{16}$$

where $z_{NDC}$ is the value of the depth buffer $Z(\mathbf{p}')$ transformed into $NDC$ space and $n$, $f$ correspond to the near and far values of the image buffer currently used (see Listing 7.1).

Athanasios Gaitatzes

```
in vec3 voxel_position, voxel_tex_coord;
uniform float voxel_r; // voxel radius
uniform sampler3D vol_shR, vol_shG, vol_shB, vol_normals;

void main (void)
{
  vec4 voxel_pos_wcs = vec4 (voxel_position, 1.0);
  vec3 voxel_pos_css = PointWCS2CSS (voxel_pos_wcs.xyz);
  vec3 voxel_pos_ecs = PointWCS2ECS (voxel_pos_wcs.xyz);
  vec3 zbuffer_ss = MAP_ 1To1_0To1 (voxel_pos_css);
  float depth = SampleBuf (zbuffer, zbuffer_ss.xy).x;
  vec3 zbuffer_css = vec3 (voxel_pos_css.xy, 2.0*depth  1.0);
  vec3 zbuffer_ecs = PointCSS2ECS (zbuffer_css);

  vec3 voxel_mf_wcs = voxel_pos_wcs.xyz + voxel_r * vec3(1.0);
  voxel_mf_wcs = max (voxel_mf_wcs,
                      voxel_pos_wcs.xyz + voxel_half_size);
  vec3 voxel_mb_wcs = voxel_pos_wcs.xyz + voxel_r * vec3( 1.0);
  voxel_mb_wcs = min (voxel_mb_wcs,
                      voxel_pos_wcs.xyz   voxel_half_size);
  vec3 voxel_mf_ecs = PointWCS2ECS (voxel_mf_wcs);
  vec3 voxel_mb_ecs = PointWCS2ECS (voxel_mb_wcs);
  float bias = distance (voxel_mf_ecs, voxel_mb_ecs);

  vec4 shR_value = SampleBuf (vol_shR, voxel_tex_coord);
  vec4 shG_value = SampleBuf (vol_shG, voxel_tex_coord);
  vec4 shB_value = SampleBuf (vol_shB, voxel_tex_coord);
  vec4 normal_value = SampleBuf (vol_normals, voxel_tex_coord);

  if (voxel_pos_ecs.z > zbuffer_ecs.z + bias) { // discard
    normal_value = vec4 (0,0,0,0);
    shR_value = shG_value = shB_value = vec4 (0,0,0,0);
  }

  // keep
  gl_FragData[0] = normal_value;
  gl_FragData[1] = shR_value;
  gl_FragData[2] = shG_value;
  gl_FragData[3] = shB_value;
}
```

Listing 7.1: Cleanup phase fragment shader

Athanasios Gaitatzes

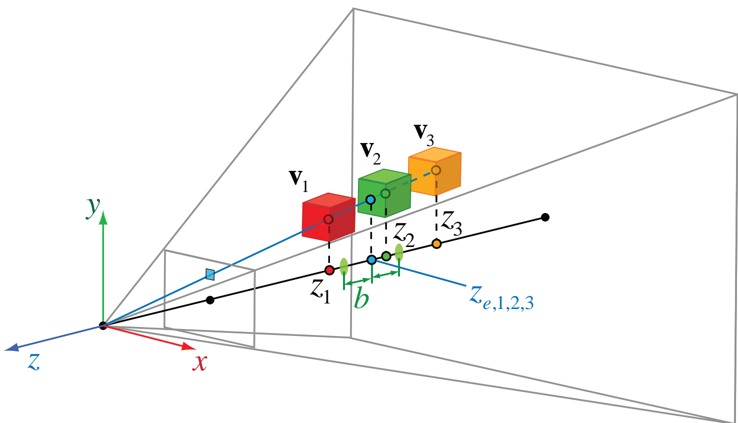



Figure 70: Cleanup stage: Voxels beyond the boundary depth zone are retained (orange), while voxels closer to the buffer center of projection are rejected (red). Voxels that correspond to the depth value registered in the buffer must be updated (green).

Expressing the coordinates in the eye reference frame (Figure 70), if $\mathbf{p}'_{v,z} > z_e$ the voxel must be cleared, as it crosses the recorded depth boundary in the image buffer. However, the spatial data are quantized according to the volume resolution and therefore a bias $b$ has to be introduced in order to avoid rejecting boundary samples. Since the depth comparison is performed in eye-space, $b$ is equal to the voxel's $\mathbf{p}_v$ radius (half diagonal) clamped by the voxel boundaries in each direction. Therefore the rejection condition becomes:

$$\mathbf{p}'_{v,z} > z_e + b \tag{17}$$

The example in Figure 70 explains the cleanup and update state changes of a voxel with respect to the available depth information in an image buffer. All voxels in the figure correspond to the same image buffer sample with eye-space value $z_{e,1,2,3}$. Voxel $\mathbf{v}_1$ is rejected (cleared) because $z_1$ is greater than $z_{e,1,2,3} + b$. Voxel $\mathbf{v}_2$ must be updated since it lies within the boundary depth zone $[z_{e,1,2,3} - b, z_{e,1,2,3} + b]$. Finally, voxel $\mathbf{v}_3$ is retained, since it lies beyond the registered depth value.

Athanasios Gaitatzes

### 7.4.2 Injection phase

In the injection phase, a rectangular grid of point primitives corresponding to each depth image buffer is sent to a vertex shader which offsets the points according to the stored depth. The points are subsequently transformed to world space and finally to volume-clip space. If world space or volume clip-space coordinates are already available in the buffers, they are directly assigned to the corresponding injected points. The volume clip-space depth is finally used to determine the slice in the volume where the point sample attributes are accumulated (see Listing 7.2). At the end of this stage, the previous version of the scene's voxel representation has been updated to include a partial voxelization of the scene based on the newly injected point samples. The resolution of the grid of 2D points determines how much detail of the surfaces represented by the depth buffer is transferred into the volume and whether or not the geometry is sparsely sampled. If too few points are injected, the resulting volume will have gaps. This may be undesirable for certain application cases, such as the LPV method [54] or ray-marching algorithms.

### 7.4.3 Single-pass Incremental algorithm

In order to transfer the geometric detail present in the G-buffers to the volume representation and ensure a dense population of the resulting volume, a large resolution for the grid of injected points must be used. However, the injection stage involves rendering the point grid using an equal number of texture lookups and, in some implementations, a *geometry shader*. This has a potentially serious impact on performance (see Figure 73), especially for multiple injection viewpoints.

We can totally forgo the injection phase of the algorithm and do both operations in one stage. Using the same notation as before, the logic of the algorithm remains practically the same. If the projected voxel center lies in front of the recorded depth (i.e. $\mathbf{p}'_{v,z} > z_e + b$), it is still cleared. If

Athanasios Gaitatzes

```
// Vertex Shader Stage

flat out vec2 tex_coord;
uniform sampler2D zbuffer;

void main (void)
{
  tex_coord = gl_Vertex.xy;
  float depth = SampleBuf (zbuffer, tex_coord).x;

  // screen space  > canonical screen space
  vec3 pos_css = MAP_0To1_ 1To1 (vec3 (gl_Vertex.xy, depth));

  // canonical screen space  > object space
  vec3 pos_wcs = PointCSS2WCS (pos_css);

  // world space  > clip space
  gl_Position = gl_ModelViewProjectionMatrix *
                vec4 (pos_wcs, 1.0);
}

// Geometry Shader Stage

layout(points) in;
layout(points, max_vertices = 1) out;

uniform int vol_depth;
flat in vec2 tex_coord[];
flat out vec2 gtex_coord;

void main (void)
{
  gtex_coord = tex_coord[0];

  gl_Position = gl_PositionIn[0];
  gl_Layer = int (vol_depth * MAP_ 1To1_0To1 (gl_Position.z));

  EmitVertex();
}
```

Listing 7.2: Injection phase using a geometry shader to select the destination slice of the volume for the point samples.

Athanasios Gaitatzes

the projected voxel center lies behind the recorded depth (i.e. $\mathbf{p}'_{v,z} < z_e \quad b$), the voxel is retained; otherwise it is turned-on (or updated) using the attribute buffers information. The last operation practically replaces the injection stage.

As we are effectively sampling the geometry at the volume resolution instead of doing so at higher, image-size-dependent rate and then down-sampling to volume resolution, the resulting voxelization is expected to degrade. However, since usually depth buffers are recorded from multiple views, missing details are gradually added. Comparison of the method variations and analysis of their respective running times is given in Section 7.7.

### 7.5 Incremental Voxelization for Lighting

As a case study, we applied incremental voxelization to the problem of computing indirect illumination for real-time rendering. When using the *Incremental Voxelization* technique for lighting effects, as in the case of the *Light Propagation Volumes* algorithm of Kaplanyan [54] or ray marching techniques (Thiedemann et al. [108], Mavridis et al. [72]), the volume attributes must include occlusion information (referred to as *geometry volume* in [54]), sampled normal vectors, direct lighting (VPLs) and optionally surface albedo in the case of secondary indirect light bounces. Direct illumination and other accumulated directional data are usually encoded and stored as low-frequency spherical harmonic coefficients (see Sloan et al. [100]).

*Virtual Point Lights* (VPLs) are points in space that act as light sources and encapsulate light reflected off a surface at a given location. In order to correctly accumulate VPLs in the volume, during the injection phase, a separate volume buffer is used which is cleared in every frame in order to avoid erroneous accumulation of lighting. For each RSM, all VPLs are injected and additively blended. Finally, the camera attribute buffers are injected to provide view-dependent dense samples of the volume. If lighting from the camera is also exploited (as in our implementation),

Athanasios Gaitatzes

the injected VPLs must replace the corresponding values in the volume, since the camera direct lighting buffer provides cumulative illumination. After the cleanup has been performed on the previous version of the attribute volume $V_{prev}$, non-empty voxels from the separate injection buffer replace corresponding values in $V_{curr}$. This ensures that potentially stale illumination on valid volume cells from previous frames is not retained in the final volume buffer.

## 7.6 Implementation

The *Incremental Voxelization* method runs entirely on the GPU and has been implemented on a deferred shading platform using basic OpenGL® 3.0 operations on an Intel Core i7 860 at 2.8GHz with 8GB of RAM and equipped with an NVIDIA® GeForce GTX285 GPU with 1GB of video memory. We have implemented two versions of the buffer storage mechanism in order to test their respective speed. The first uses 3D volume textures along with a geometry shader that sorts injected fragments to the correct volume slice. The second unwraps the volume buffers into 2D textures and dispenses with the expensive geometry processing (respective performance can be seen in Figure 73).

The texture requirements are two volume buffers for ping-pong rendering ($V_{prev}$, $V_{curr}$). Each volume buffer stores $N$-dimensional attribute vectors $\mathbf{a}$ and corresponds to a number of textures (2D or 3D) equal to $\lceil N/4 \rceil$, for 4-channel textures. For lighting applications an additional $N$-dimensional volume buffer is required, for the reasons explained in Section 7.5. In our implementation we need to store surface normals and full color spherical harmonics coefficients for incident flux in each volume buffer, which translates to $3 \times 4$ textures in total.

In order to create the data storage structure, we generate on the GPU a uniform spatial partitioning structure in real-time. For the voxelization, the user has the option to request several

Athanasios Gaitatzes

attributes to be computed and stored into floating point buffers for later use. Among them are surface attributes like albedo and normals, but also dynamic lighting information and radiance values in the form of low-order spherical harmonics (SH) coefficients representation (either monochrome radiance or full color encoding i.e. separate radiance values per color band). In our implementation the radiance of the corresponding scene location is calculated and stored as a $2^{nd}$ order spherical harmonic representation for each voxel. For each color band, four SH coefficients are computed and encoded as RGBA float values, since the four SH coefficients map very well to the four component buffers supported by the graphics hardware.

The following sections provide an evaluation of the *Incremental Voxelization* method in terms of robustness, performance and usability and compare the proposed method against other modern techniques, as necessary.

### 7.7 Performance & Evaluation

In terms of voxelization robustness, our algorithm complements single-frame image-based voxelization and supports both moving image viewpoints and fully dynamic geometry and lighting. In Figure 71, a partial volume representation of the Crytek Sponza II Atrium model is generated at a $64^3$ resolution and a $128^2$-point injection grid using single-frame and *Incremental Voxelization*. (a) and (b) are the single-frame volumes from two distinct viewpoints. (c) is the *Incremental Voxelization* after the viewpoint moves across several frames. Using the partial single-frame volumes for global illumination calculation, we observe abrupt changes in lighting as the camera reveals more occluding geometry (e.g. left arcade wall and floor - insets (e) and (f)). However, the situation is gradually remedied in the case of *Incremental Voxelization*, since newly discovered volume data are retained for use in following frames (insets (g) and (h)).

Athanasios Gaitatzes

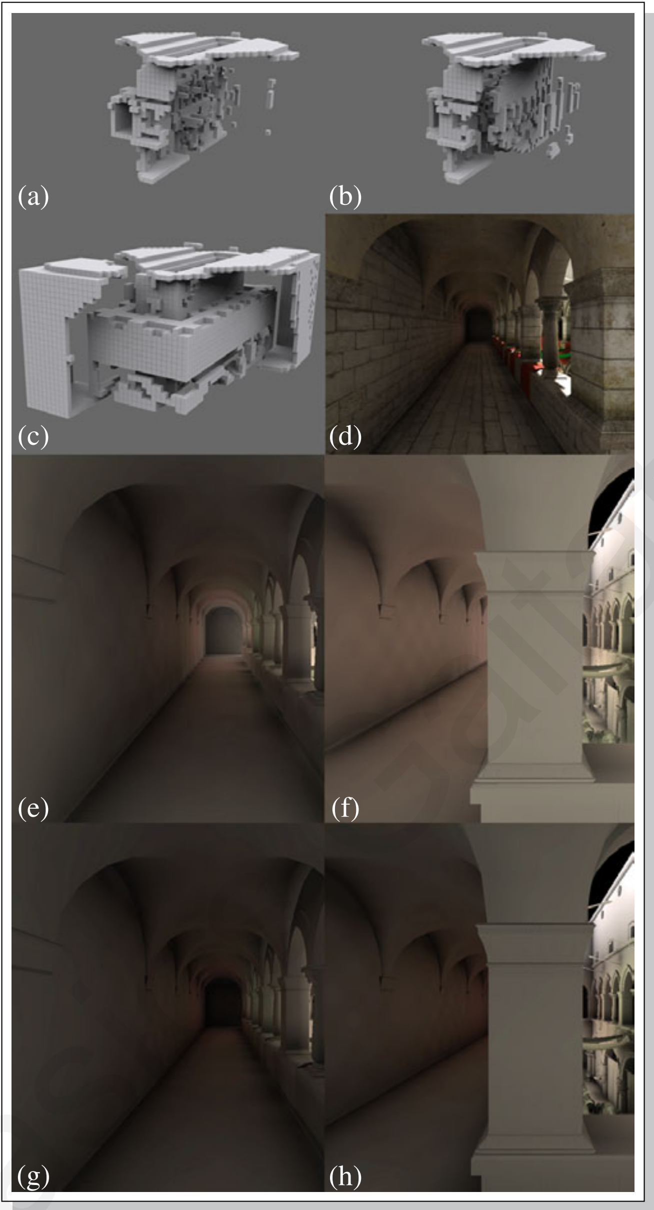



Figure 71: Comparison of the voxelization of the Crytek Sponza II Atrium. (a), (b) Single frame image-based voxelization from two distinct viewpoints where it is not possible to capture all environment details as no information exists in the buffers. (c) *Incremental Voxelization* (*IV*) produced over several frames. (d) Complex illumination using *IV*. (e), (f) Indirect lighting buffers corresponding to the single frame voxelization of (a) and (b). (g), (h) *IV* indirect lighting buffers (of the voxelization in c).

Athanasios Gaitatzes

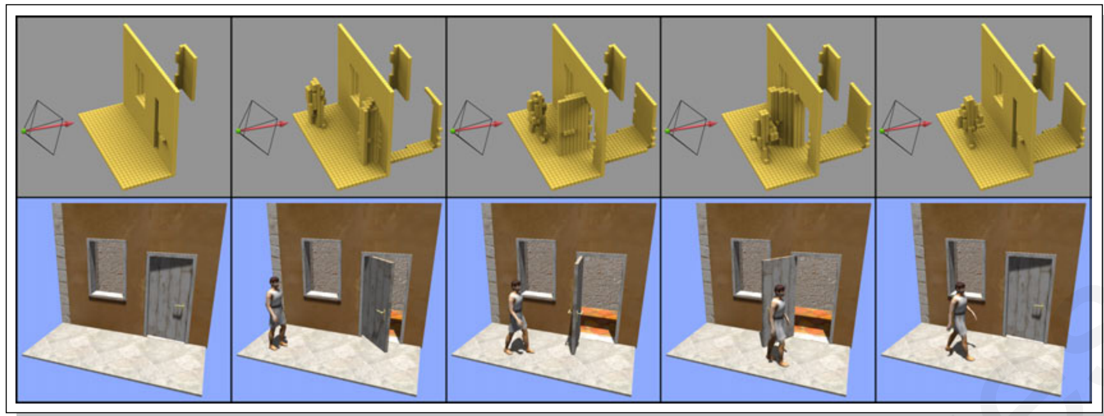


Figure 72: Image-based voxelization of a dynamic scene containing an articulated object using only camera-based injection.

Figure 72 demonstrates *Incremental Voxelization* in a dynamic environment. In particular, it shows an animated sequence of a scene with moving and deformable objects, as well as the corresponding voxelization from the camera viewpoint. Notice how the wall behind the closed door is not initially present in the volume, but after the door opens, it is gradually added to the volume and remains there even after the door swings back. The same holds for the geometry behind the animated character.

The top of Figure 73 shows a decomposition of the total algorithm running time into the cleanup and injection stage times respectively versus different volume buffer resolutions for three different injection grid sizes. For fixed injection grid resolutions, we have observed that injection times are not monotonically increasing with respect to volume size, as one would expect. The performance also decreases when the buffer viewpoint moves close to geometry. We attribute this to the common denominator of both cases, namely the fact that pixel overdraw is induced, as points are rasterized in the same voxel locations. This is particularly evident in the blue curve of the $64^2$ injection stage graph of Figure 73 (left inset). Note that this behavior is an inherent attribute of injection techniques in general; image-based voxelization methods depend heavily

Athanasios Gaitatzes

on the sampling rate used. When this rate is incompatible with the voxel space resolution, holes might appear (under-sampling). To ensure adequate coverage of the voxel grid, dense image-space point samples are drawn, which in turn leads to overdraw problems in many cases. One can use an injection grid proportional to the volume resolution, which partially alleviates the overdraw issue but in turn decreases performance as can be seen in the red curve of the injection graph of Figure 73.

The time required for a single-frame image-based voxelization (one G-buffer) equals the time of our injection stage plus a very small overhead to clear the volume buffer, since the two operations are equivalent. Thus, the only difference in the execution time of *Incremental Voxelization* is the cleanup stage time. With regard to the quality of the two methods, *IV* offers more stable and accurate results as new viewpoints gradually improve the volume.

The total voxelization time (bottom inset of Figure 73) is the sum of the cleanup and injection stages. As the cleanup stage performance depends only on the volume resolution and not on the injection grid size, it vastly improves the voxelization quality compared to using only screen-space injection from isolated frames, at a constant overhead per frame. Especially, when applied to global illumination calculations, where small volumes are typically used, the version of the algorithm that uses 2D textures (top-right inset) has a significantly lower execution footprint, as it is not influenced by the geometry shader execution of the 3D textures version (top-left inset), though both methods are affected by pixel overdraw during injection. Since the injection of large point grids resulted in pixel overdraw in the target buffers, we experimented with different methods for grid creation (linear, block or shifting stippled pattern) and found that it has no effect in pixel overdraw.

Athanasios Gaitatzes

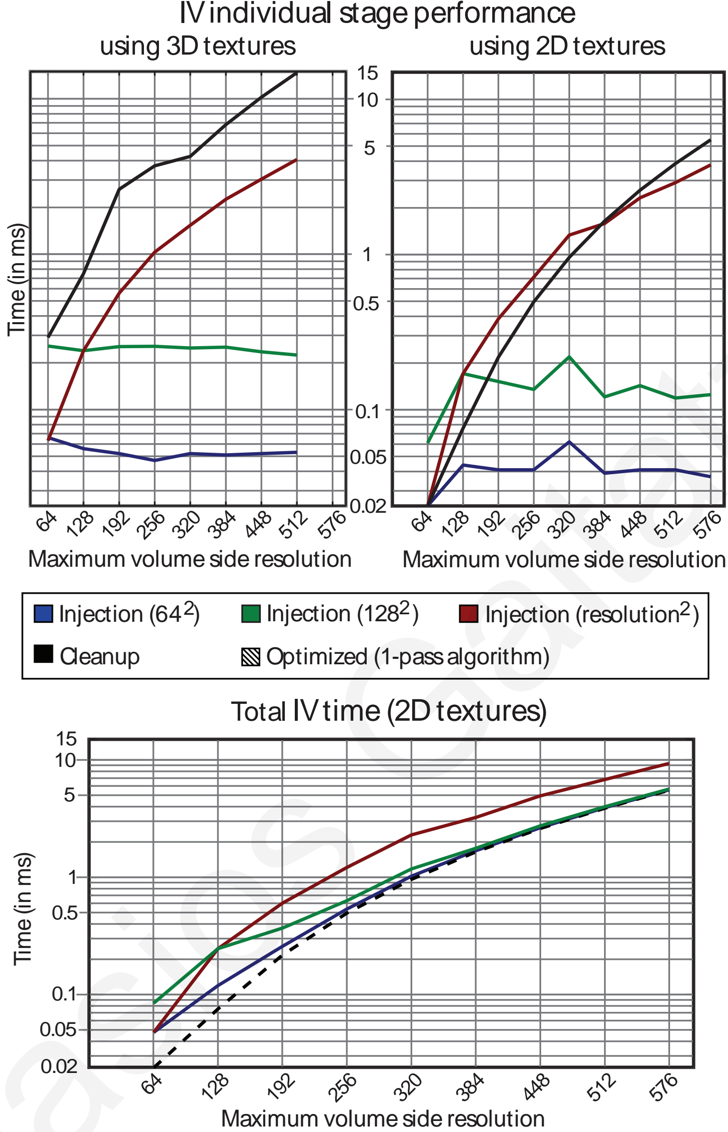



Figure 73: Top: Running time (in ms) for the cleanup and injection stages against different volume resolutions for the Crytek Sponza II Atrium model. We used a single G-buffer (camera) as input and 1 MRT (4 floats) as output. Injection is measured for three different grid sizes, one being proportional to the volume side. Bottom: Total incremental voxelization times. Note that the performance of the optimized *Incremental Voxelization* is identical to that of the cleanup stage.

Athanasios Gaitatzes

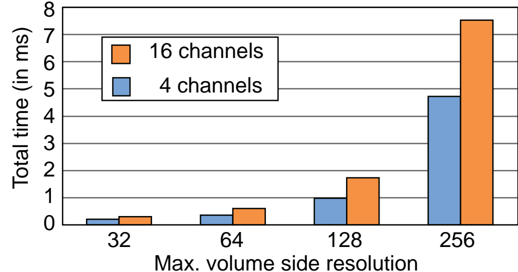



Figure 74: Multi-channel voxelization performance for the Crytek Sponza II Atrium model, using 1 MRT (emitting 4 floating point values) and 4 MRTs (emitting 16 floating point values) in the GPU fragment shader stage.

The performance of the optimized *Incremental Voxelization* is identical to that of the cleanup stage as expected, since it is essentially a modified cleanup stage. It follows that the dual stage version performance will always be slower than the optimized one.

The maximum volume resolution reported is due to hardware resource limitations on the number and size of the allocated buffers and not algorithm bounds.

In Table 8 we report the voxelization performance results for several scenes using our method and the geometry-based scalar volume generation method of Pantaleoni [80] *VoxelPipe*, which uses the general purpose compute pipeline. As the latter system exploits capabilities available only on a Shader Model 5 GPU, we multiplied Pantaleoni's reported timings by 1.5x (see Benchmarks [102], [121]) which is roughly the improvement in the overall compute performance between the GeForce GTX285 (240 CUDA cores [119]) and the GeForce GTX480 (480 CUDA cores [120]) GPUs. In addition, GPU implementations favor emitting 4 floats into an MRT and there is no significant advantage of emitting just 1 float. GPGPU implementations on the other

Athanasios Gaitatzes

hand have no such limitation so the comparison results of Table 8 in conjunction with the comparative multi-channel performance of Figure 74 show from a 3x to a more than an order of magnitude speed increase of our method vs. that of Pantaleoni's.

Comparing our method with the geometry-based multi-channel full scene voxelization method of Gaitatzes et al. [32], which, as ours, is based on the rendering pipeline (GPU), we show a big speed improvement even when adding in the whole process the G-buffers creation time.

In Table 9 we report on the quality of our voxelization method. The camera was moved around the mesh for several frames, in order for the algorithm to *incrementally* compute the best possible voxelization. For several models and resolutions we show the Hausdorff distance (defined as $d_H(X,Y) = \max(d(X,Y), d(Y,X))$ where $d$ is the metric $d(X,Y) = \sup_{x \in X} \inf_{y \in Y} d(x,y)$) between the original mesh and the resulting voxelization using the *IV* method (see column 3).

| **Scene** | **Grid** size | **VP** 1-float | **GS** 4-floats | **G-buffers** creation | **IV** 4-floats |
|---|---|---|---|---|---|
| Conference (282K tris) | $128^3$ | 5.1 | 31.73 | 3.2 | 0.28 |
| | $512^3$ | 12.5 | 64.67 | | 4.93 |
| Dragon (871K tris) | $128^3$ | 7.5 | 198.33 | 59 | 0.18 |
| | $512^3$ | 11.2 | – | | 6.98 |
| Turbine Blade (1.76M tris) | $128^3$ | 11.9 | 265.7 | 121 | 0.14 |
| | $512^3$ | 15.2 | – | | 5.37 |
| Hairball (2.88M tris) | $128^3$ | 23.0 | 436.2 | 198 | 0.33 |
| | $320^3$ | – | – | | 4.04 |
| | $512^3$ | 58.4 | – | | – |

Table 8: Voxelization timings (in ms) of various scenes and methods. VP stands for *VoxelPipe* and GS is the Geometry Slicing method of Gaitatzes et al. with 11 output vertices. IV stands for *Incremental Voxelization*. We present the total (injection + cleanup) performance values of our 2D textures implementation using an injection grid proportional to the volume size, which is our algorithm's worst case as can be seen from the red plot of Figure 73.

Athanasios Gaitatzes

| Scene | Grid | Hausdorff | |
|---|---|---|---|
| | size | $\% \, d_H(X,Y)$ | |
| Bunny | $64^3$ | 0.3289 | 0.2168 |
| | $128^3$ | 0.1694 | 0.1091 |
| (69.5K tris) | $256^3$ | 0.1064 | – |
| Dragon | $64^3$ | 0.3621 | 0.2565 |
| | $128^3$ | 0.1878 | 0.1289 |
| (871K tris) | $256^3$ | 0.1256 | 0.0645 |
| Turbine Blade | $64^3$ | 0.3457 | 0.2763 |
| | $128^3$ | 0.1821 | 0.1424 |
| (1.76M tris) | $256^3$ | 0.1232 | 0.0697 |

Table 9: Comparison of a full voxelization. We record the normalized (with respect to the mesh bounding box diagonal) average Hausdorff distance (percent). Mesh $X$ is the original mesh to be voxelized and $Y$ is the point cloud consisting of the voxel centers of the voxelization using *IV* (column 3) and a geometry-based full scene voxelization (column 4).

Athanasios Gaitatzes

We notice that our voxelized object (voxel centers) is on average 0.1% different from the original mesh. In addition, we report the Hausdorff distance between the original mesh and the geometry-based full scene voxelization of Gaitatzes et al. [32] (see column 4). We observe that the difference between the corresponding volumes is in the 0.01% range.

In Figure 75 we show a series of voxelizations of the dragon model using only the camera G-buffers. In addition, we show the respective Hausdorff distance between the original dragon model and the computed voxel centers (see plot in Figure 76). The voxelization is *incrementally* updated and improved over several frames as the camera does a complete rotation around each of the principal axis for an equal amount of frames. As the animation progresses, we observe that the Hausdorff distance decreases as the process converges to a full voxelization.

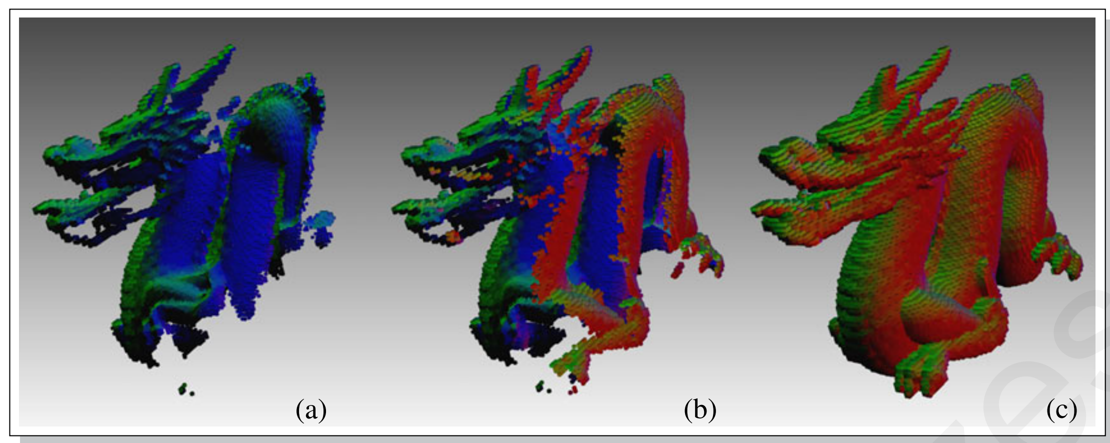



Figure 75: A series of voxelizations of the dragon model at $128^3$ resolution showing the normal vectors. The voxelization is *incrementally* updated and improved over several frames as the camera moves around the model.

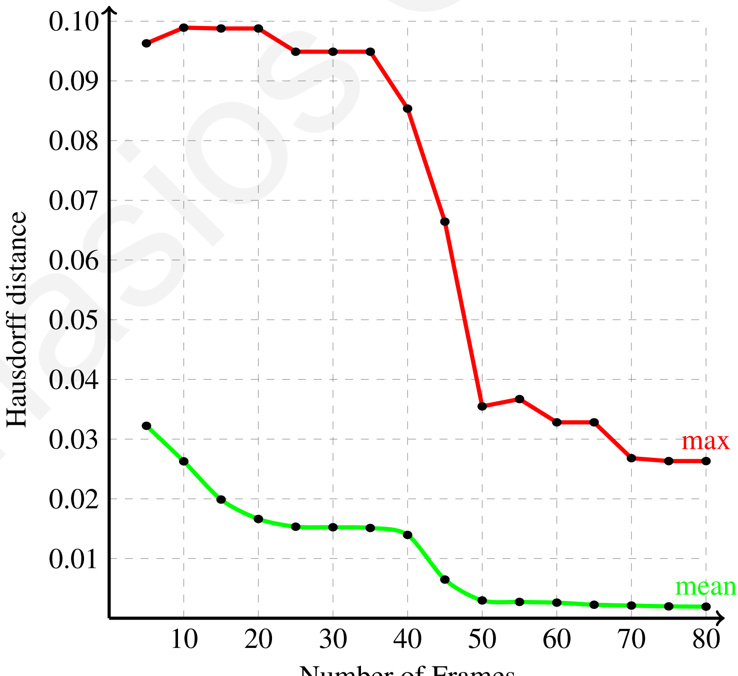



Figure 76: The decreasing Hausdorff distance between the original dragon model and the computed *Incremental Voxelizations* of Figure 75.

Athanasios Gaitatzes

## 7.8 Optimizations

The static parts of the scene (with respect to geometry, materials and lighting) can be voxelized once and the voxelization data reused thus escaping the continuous cost of the voxelization process and possibly utilizing a more robust, geometry-based voxelization method. This improves performance for dynamic scenes with large amounts of static geometry.

In addition, in scenes with dynamic geometry occupying only a small fraction of the volume (a quite typical situation), currently the clean-up stage traverses all voxels, whereas only a small amount of voxels are prone to become invalid. Restricting the clean-up stage to the volume region containing moving geometry should result in significant speed-ups.

## 7.9 Limitations

One limitation of our method is that the clean-up phase will only remove invalid voxels that are visible in any of the current image-buffers (camera MRTs and light RSMs). The visible invalid voxels will be removed from the voxelization the next time they appear in the image buffers. However, the correctness of the voxelization cannot be guaranteed for existing voxels that are not visible in any buffer. For moving geometry, some incrementally generated voxels may become stale, as shown in the case of the bottom right inset of Figure 69. Nevertheless, in typical dynamic scenes, the stale voxels are often eliminated either in subsequent frames due to their invalidation in the moving camera buffer or due to their invalidation in other views in the same frame (see Figure 77).

Another limitation is that the extents of the voxelization region must remain constant throughout volume updates; otherwise computations are performed with stale buffer boundaries. When the bounding box of the scene is modified or the scene changes abruptly or it is reloaded, the

Athanasios Gaitatzes

Figure 77: Correct indirect shadowing effects and color bleeding: Stale voxels from one view (behind the tank) are effectively invalidated in other views (reflective shadow map).

Figure 78: Scene with dynamic geometry, highlighting the shadowing effects of the tank model, as it moves towards the user, on the right wall of the tunnel.

Athanasios Gaitatzes

attribute volumes must be deleted and incrementally populated again. This is also the reason why the *Cascaded Light Propagation Volumes* (CLPV) method of Kaplanyan et al. [55] could not take advantage of *Incremental Voxelization* for the cascades near the user, as the method assumes that they follow the user around, constantly modifying the current volume extents.

### 7.10 Discussion & Summary

We presented an image-based method to incrementally build a discretization of dynamic scenes as demonstrated in Figure 78 and Figure 79. Our method achieves improved quality over non-incremental methods, while it maintains the high performance merits of image-based techniques.

In the same manner that *Incremental Voxelization* combines partial volumes distributed across different locations, it can be adapted to handle volume samples distributed across time, achieving a controllable cost amortization of existing voxelization techniques. This is easily implemented by splitting the injection point grid into multiple point sets, which are injected in the volume in an interleaved manner or by lazily updating regions of the volume when and where changes occur.

The *Incremental Voxelization* framework will be publicly available from the website of the author `http://www.virtuality.gr/gaitat/en/publications.html` under the appropriate paper.

Athanasios Gaitatzes

Figure 79: Scene with dynamic lighting. Sequence of a side-by-side comparison of a single-frame image-based voxelization (left images) vs. *incremental* image-based voxelization (right-images). The curtains that are hidden behind the colonnade do not obstruct the light in the case of the single-frame voxelization.

# Chapter 8

# Conclusion

The goal of this research was to develop new real-time algorithms that improve the quality of the illumination in dynamic complex environments or accelerate the expensive computation of existing algorithms using graphics hardware. To achieve this goal, we investigated some reasonable approximations in order to find a visually plausible compromise between quality and performance. We considered the creation of a discretized representation of the visibility function around an object, as the exact visibility computation is expensive to compute in real-time. We examined the creation of a discretized representation of the incoming light in order to estimate diffuse interactions from multiple light bounces. Finally, we investigated the creation of a discretized representation of the scene geometry and used it for accelerating the above process. In Figure 80 we show a diagram indicating the correlation between the *Discretization* methods that were developed and analyzed in this dissertation.

## 8.1 Summary of Contributions

In Chapter 4, we presented an *Ambient occlusion* method that is a direct application of *Visibility discretization* as the hemisphere of infinite rays around a point is discretized to several samples.



Athanasios Gaitatzes

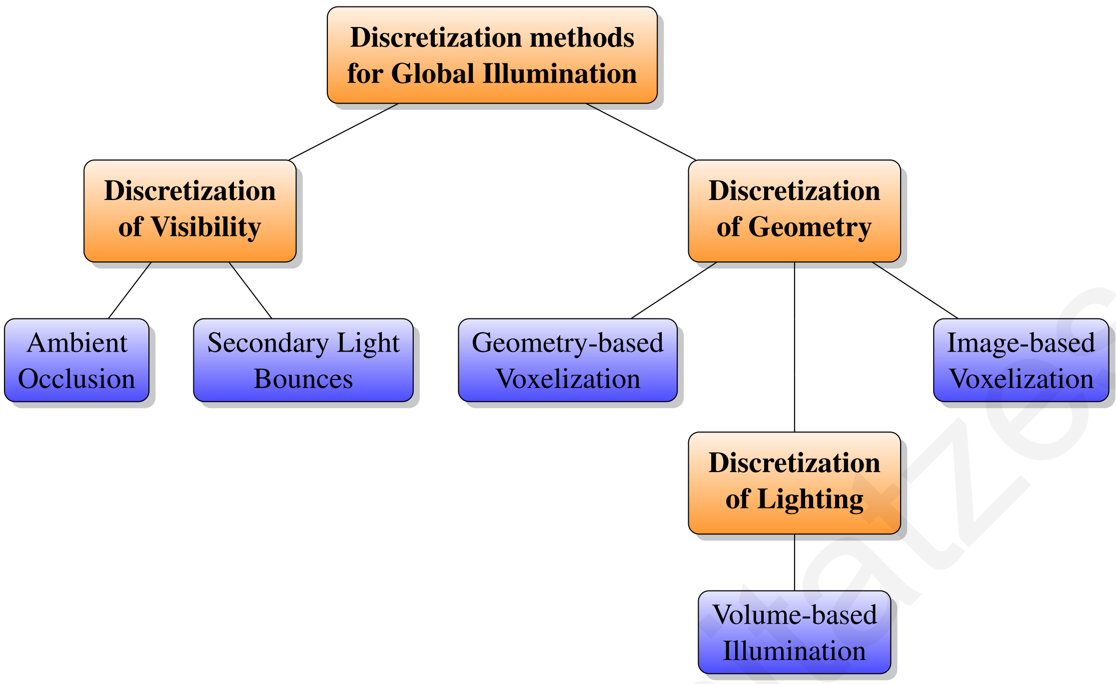



Figure 80: Diagram indicating the correlation of the *Discretization* methods used in this dissertation. The blue rectangles indicate the application domains.

Athanasios Gaitatzes

In order to accelerate the visibility function computation in dynamic scenes composed of rigid non-penetrating objects, we proposed a pre-computation, for each object in the scene, of the visibility information, as seen from the environment, onto the bounding sphere surrounding the object. The visibility function was encoded by a four-dimensional *visibility field* that described the distance of the object in each direction for all positional samples on a sphere around the object. Thus, we were able to speed up the calculation of most algorithms that trace visibility rays to real-time frame rates. The method has several advantages over the previous work. First the displacement maps are pre-calculated faster and stored as grayscale textures. Then, during the real-time simulation the time to access the displacement values is constant and in addition, the displacement maps contain information that is transformation invariant. Finally the method can handle several different cases

like intra-object occlusion and inter-object occlusion but also shadow and reflection rays in the case of ray-tracing, cases which previous work [63], [70] could not handle.

In Chapter 5, we presented our efforts in simulating *Global illumination* by *Discretizing the illumination* of the scene by using *Virtual Point Light* methods. In order to capture the complex interactions of light with the environment, we proposed a real-time algorithm to compute the global illumination of dynamic scenes with complex dynamic illumination. We used a virtual point light (VPL) illumination model on the volume representation of the scene. The method handled occlusion (shadowing and masking) caused by the interference of geometry and was able to estimate diffuse inter-reflections from multiple light bounces. It has several advantages over previous work [54]; by taking into account indirect occlusion and secondary light bounces we were able to produce more accurate illumination while always maintaining a high frame rate.

In Chapter 6, we presented our efforts in *Discretizing the geometry* of the scene as it was one of the bottlenecks of the *Light Propagation Volumes* method. As an increasing number of rendering and geometry processing algorithms relies on volume data to calculate anything from effects global illumination or visibility information, we proposed two real-time and simple-to-implement surface voxelization algorithms, the *Volume Buffer*, which encapsulates functionality, storage and access similar to a *frame buffer object*, but for three-dimensional scalar data. The *Volume Buffer* can rasterize primitives in 3D space and accumulate up to 1024 bits of arbitrary data per voxel, as required by the specific application. The method is much faster to compute than previous methods [11], [25] that perform rasterization-based voxelization by using the rendering pipeline (GPU). It also has the ability to store arbitrary data on each voxel (up to 1024 bits when using 8 MRT).

In Chapter 7 we introduced the concept of *Incremental Voxelization* for the multi-valued, scalar volume rasterization of fully dynamic scenes (geometry, materials and lighting) and demonstrated

Athanasios Gaitatzes

its application in the context of volume-based global illumination. Where current image-based voxelization algorithms [55] repeatedly regenerate the volume using the deferred geometry image buffers of a single frame, we incrementally updated the existing voxels using a depth-buffer re-projection scheme and therefore, produced a more complete voxelization of the scene. We showed that *incremental voxelization* can be used for multi-attribute volumes and complex dynamic scenes. The *Incremental Voxelization* framework will be publicly available from the website of the author http://www.virtuality.gr/gaitat/en/publications.html under the appropriate paper.

## 8.2 Thoughts about Future Work

The first main area we would like to explore is the voxel representations of sparse scenes, as in the work of Crassin et al. [17] but for fully dynamic and animated environments. This way we will be able to avoid the computations required for empty voxels. A new scheme will be required for the radiance deflection mechanism in the environment as to jump over void space until valid voxels are found. Investigating non-rectangular grid structures might also be a promising research direction.

In regards to the illumination of the scene itself, as we only considered illumination arriving from point and spot lights, other types can be investigated like area lights and illumination arriving from an environment map. For these cases a new *Reflective shadow maps* strategy would have to be devised in order to correctly sample and create the required *Virtual points lights* of the scene.

Furthermore, the possibility of a more accurate but still manageable radiance deflection mechanism will be investigated to further enhance light propagation in large and dynamic environments. Another interesting direction of research is to extend the *Light Propagation Volumes* method in order to take into account the specular light transport which can not be currently addressed as

Athanasios Gaitatzes

the energy propagation between neighboring cells requires repeated interpolation, which does not allow light beams to maintain sharp profiles.

Since the lattice is relatively coarse in the *Light Propagation Volumes* method, light leaks despite fuzzy blockers, as described in [55]. A better method for modeling blocking could improve correctness and quality significantly. For instance, by adaptively increasing grid resolution, blocking and propagation quality could be improved. Additionally, one could investigate different packing structures where cells have fewer and/or more equidistant neighbors, which would improve performance and quality, respectively.

In addition, in the same manner that *Incremental Voxelization* combines partial volumes distributed across different locations, it can be adapted to handle volume samples distributed across time, achieving a controllable cost amortization of existing voxelization techniques. This can be implemented by splitting the injection point grid into multiple point sets, which are injected in the volume in an interleaved manner or by lazily updating regions of the volume when and where changes occur.

Finally, the use of multiple G-buffers for *Incremental Voxelization* can be extended to other domains like image-based *Ambient Occlusion* where a fast and accurate merging of the multiple view combinations would have to be devised.

Athanasios Gaitatzes

Athanasios Gaitatzes

# Bibliography

[1] Akenine-Möller, T., Haines, E., Hoffman, N.: Real-time Rendering, Third Edition. A. K. Peters, Ltd. (2008) [23]

[2] Amit, B.D.: GPU Ray Tracing. Master's thesis, Technion Israel Institute of Technology (2007) [66]

[3] Arvo, J.: Linear time voxel walking for octrees **1**(5) (1988). Available from http://tog.acm.org/resources/RTNews/html/rtnews2d.html [50]

[4] Bavoil, L., Sainz, M., Dimitrov, R.: Image-space horizon-based ambient occlusion. In: ACM SIGGRAPH Talks, SIGGRAPH '08. ACM, New York, NY, USA (2008). URL http://doi.acm.org/10.1145/1401032.1401061 [37]

[5] Blythe, D.: The Direct3D 10 system. ACM Transactions on Graphics **25**(3), 724–734 (2006). DOI 10.1145/1141911.1141947. URL http://doi.acm.org/10.1145/1141911.1141947 [23]

[6] Bunnell, M.: Dynamic Ambient Occlusion and Indirect Lighting. In: M. Pharr, R. Fernando (eds.) GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation, pp. 223–234. Addison-Wesley Professional (2005) [36]

[7] Carr, N.A., Hall, J.D., Hart, J.C.: The ray engine. In: ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (HWWS), pp. 37–46. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2002) [50]

[8] Carr, N.A., Hoberock, J., Crane, K., Hart, J.C.: Fast GPU ray tracing of dynamic meshes using geometry images. In: Proceedings of Graphics Interface (GI), pp. 203–209. Canadian Information Processing Society, Toronto, Ont., Canada, Canada (2006) [51]

[9] Cazals, F., Drettakis, G., Puech, C.: Filtering, Clustering and Hierarchy Construction: A New Solution for Ray Tracing Very Complex Environments. Computer Graphics Forum **14**, 371–382 (1995) [50]

[10] Chatelier, P.Y., Malgouyres, R.: A low-complexity discrete radiosity method. Computers & Graphics **30**(1), 37–45 (2006). URL http://dx.doi.org/10.1016/j.cag.2005.10.008 [39]

[11] Chen, H., Fang, S.: Fast voxelization of three-dimensional synthetic objects. Journal of Graphics Tools **3**(4), 33–45 (1998) [6, 41, 94, 161]

Athanasios Gaitatzes

[12] Christen, M., Engel, W.: Ray Tracing on GPU. Master's thesis, Univ. of Applied Sciences Basel, Switzerland (2005) [50]

[13] Clark, J.H.: Hierarchical geometric models for visible surface algorithms. Communications of the ACM **19**(10), 547–554 (1976). URL http://doi.acm.org/10.1145/360349.360354 [50]

[14] Cook, R.L.: Stochastic sampling in computer graphics. ACM Transactions on Graphics **5**(1), 51–72 (1986). URL http://doi.acm.org/10.1145/7529.8927 [49]

[15] Crane, K., Llamas, I., Tariq, S.: Real-time Simulation and Rendering of 3D Fluids. In: H. Nguyen (ed.) GPU Gems 3, chap. 30, pp. 723–739. Addison-Wesley Professional (2007) [41, 110, 132]

[16] Crassin, C., Neyret, F., Lefebvre, S., Eisemann, E.: GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 15–22. ACM, New York, NY, USA (2009). URL http://doi.acm.org/10.1145/1507149.1507152 [40]

[17] Crassin, C., Neyret, F., Sainz, M., Eisemann, E.: Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels. In: W. Engel (ed.) GPU Pro, pp. 643–677. A K Peters (2010). URL http://maverick.inria.fr/Publications/2010/CNSE10 [162]

[18] Dachsbacher, C., Stamminger, M.: Reflective shadow maps. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 203–231. ACM, New York, NY, USA (2005). URL http://doi.acm.org/10.1145/1053427.1053460 [45, 88, 133, 137]

[19] Dachsbacher, C., Stamminger, M.: Splatting indirect illumination. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 93–100. ACM, New York, NY, USA (2006). URL http://doi.acm.org/10.1145/1111411.1111428 [46, 88]

[20] Dong, Z., Chen, W., Bao, H., Zhang, H., Peng, Q.: Real-time Voxelization for Complex Polygonal Models. In: 12th Pacific Conference on Computer Graphics and Applications (PG), PG '04, pp. 43–50. IEEE Computer Society, Washington, DC, USA (2004). URL http://portal.acm.org/citation.cfm?id=1025128.1026026 [41, 42]

[21] Dutre, P., Bala, K., Bekaert, P.: Advanced Global Illumination. A. K. Peters, Ltd., Natick, MA, USA (2002) [20]

[22] Eisemann, E., Décoret, X.: Fast scene voxelization and applications. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 71–78. ACM, New York, NY, USA (2006). URL http://doi.acm.org/10.1145/1111411.1111424 [40, 41, 42, 125, 137]

[23] Eisemann, E., Décoret, X.: Single-pass GPU Solid Voxelization for Real-Time Applications. In: Proceedings of Graphics Interface (GI), vol. 322, pp. 73–80. Canadian Information Processing Society, Toronto, Ontario, Canada (2008) [40, 94]

[24] Ernst, M., Vogelgsang, C., Greiner, G.: Stack Implementation on Programmable Graphics Hardware. In: B. Girod, M.A. Magnor, H.P. Seidel (eds.) Vision, Modeling and Visualization Conference (VMV), pp. 255–262. Aka GmbH (2004). URL http://dblp.uni-trier.de/db/conf/vmv/vmv2004.html#ErnstVG04 [50]

Athanasios Gaitatzes

[25] Fang, S., Chen, H.: Hardware accelerated voxelization. Computers & Graphics **24**(3), 433–442 (2000). URL http://dx.doi.org/10.1016/S0097-8493(00)00038-8 [6, 41, 124, 161]

[26] Foley, T., Sugerman, J.: KD-tree acceleration structures for a GPU raytracer. In: ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (HWWS), pp. 15–22. ACM, New York, NY, USA (2005). URL http://doi.acm.org/10.1145/1071866.1071869 [51]

[27] Forest, V., Barthe, L., Paulin, M.: Real-time Hierarchical Binary-Scene Voxelization. Journal of Graphics, GPU and Game Tools **14**(3), 21–34 (2009) [40, 41]

[28] Fujimoto, A., Tanaka, T., Iwata, K.: Arts: Accelerated Ray-Tracing System. IEEE Computer Graphics and Applications pp. 16–26 (1986) [50]

[29] Gaitatzes, A., Andreadis, A., Papaioannou, G., Chrysanthou, Y.: Fast Approximate Visibility on the GPU using pre-computed 4D Visibility Fields. In: 18th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG) (2010). URL http://graphics.cs.aueb.gr/graphics/docs/papers/aowscg2010.pdf [9]

[30] Gaitatzes, A., Chrysanthou, Y., Papaioannou, G.: Presampled Visibility for Ambient Occlusion. In: 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG) (2008). URL http://wscg.zcu.cz/WSCG2008/Papers_2008/journal/B07-full.pdf [9]

[31] Gaitatzes, A., Mavridis, P., Papaioannou, G.: Interactive Volume-Based Indirect Illumination of Dynamic Scenes. In: D. Plemenos, G. Miaoulis (eds.) Intelligent Computer Graphics 2010, *Studies in Computational Intelligence*, vol. 321, pp. 229–245. Springer Berlin / Heidelberg (2010). URL http://www.springerlink.com/content/e763tt336vh10852/ [9]

[32] Gaitatzes, A., Mavridis, P., Papaioannou, G.: Two Simple Single-pass GPU methods for Multi-channel Surface Voxelization of Dynamic Scenes. 19th Pacific Conference on Computer Graphics and Applications - short papers (PG) pp. 31–36 (2011). URL http://diglib.eg.org/EG/DL/PE/PG/PG2011short/031-036.pdf [9, 152, 153]

[33] Gaitatzes, A., Papaioannou, G.: Progressive Screen-space Multi-channel Surface Voxelization. In: W. Engel (ed.) GPU Pro 4: Advanced Rendering Techniques. A. K. Peters, Ltd. / CRC Press (2013) [9]

[34] Geiss, R.: Generating Complex Procedural Terrains Using the GPU. In: H. Nguyen (ed.) GPU Gems 3, chap. 1, pp. 10–22. Addison-Wesley Professional (2007) [110, 132]

[35] Gigahertz-Optik, I.: Basic radiometric quantities (2011). URL http://www.light-measurement.com/basic-radiometric-quantities/ [20]

[36] Glassner, A.S.: Space subdivision for fast ray tracing. IEEE Computer Graphics and Applications **4**(10), 15–22 (1984) [50]

Athanasios Gaitatzes

[37] Goldsmith, J., Salmon, J.: Automatic Creation of Object Hierarchies for Ray Tracing. IEEE Computer Graphics & Applications **7**(5), 14–20 (1987). URL http://dx.doi.org/10.1109/MCG.1987.276983 [50]

[38] Goral, C.M., Torrance, K.E., Greenberg, D.P., Battaile, B.: Modeling the interaction of light between diffuse surfaces. In: 11th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 213–222. ACM, New York, NY, USA (1984). URL http://doi.acm.org/10.1145/800031.808601 [39]

[39] Green, R.: Spherical Harmonic Lighting: The Gritty Details. In: Archives of the Game Developers Conference (2003). URL http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf [120]

[40] Greger, G., Shirley, P., Hubbard, P.M., Greenberg, D.P.: The Irradiance Volume. IEEE Computer Graphics and Applications **18**(2), 32–43 (1998). URL http://dx.doi.org/10.1109/38.656788 [39, 44]

[41] Günther, J., Popov, S., Seidel, H.P., Slusallek, P.: Real-time Ray Tracing on GPU with BVH-based Packet Traversal. In: 2007 IEEE Symposium on Interactive Ray Tracing (RT), pp. 113–118. IEEE Computer Society, Washington, DC, USA (2007). URL http://dx.doi.org/10.1109/RT.2007.4342598 [51, 66]

[42] Harada, T.: Real-time Rigid Body Simulation on GPUs. In: H. Nguyen (ed.) GPU Gems 3, chap. 29, pp. 705–722. Addison-Wesley Professional (2007) [110, 132]

[43] Havran, V.: Heuristic Ray Shooting Algorithms. Ph.D. thesis, Czech Technical University in Prague (2000) [50]

[44] Havran, V., Bittner, J.: On Improving KD-Trees for Ray Shooting. In: 2002 International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG), pp. 209–217 (2002) [50]

[45] Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P.: Interactive k-d tree GPU raytracing. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 167–174. ACM, New York, NY, USA (2007). URL http://doi.acm.org/10.1145/1230100.1230129 [51, 74]

[46] Huang, P., Wang, W., Yang, G., Wu, E.: Traversal fields for ray tracing dynamic scenes. In: ACM Symposium on Virtual Reality Software and Technology (VRST), pp. 65–74. ACM, New York, NY, USA (2006). URL http://doi.acm.org/10.1145/1180495.1180510 [38, 59]

[47] Illingworth, V.: The Penguin Dictionary of Physics. Penguin (2001) [20]

[48] Iones, A., Krupkin, A., Sbert, M., Zhukov, S.: Fast, Realistic Lighting for Video Games. IEEE Computer Graphics and Applications **23**(3), 54–64 (2003). URL http://dx.doi.org/10.1109/MCG.2003.1198263 [35]

[49] Jansen, F.W.: Data structures for ray tracing. In: Proceedings of a workshop (Eurographics Seminars on Data structures for raster graphics, pp. 57–73. Springer-Verlag New York, Inc., New York, NY, USA (1986) [50]

Athanasios Gaitatzes

[50] Jensen, H.W.: Global illumination using Photon maps. In: 7th Eurographics Workshop on Rendering Techniques (EGSR), pp. 21–30. Springer-Verlag, London, UK (1996) [43]

[51] Jevans, D., Wyvill, B.: Adaptive Voxel Subdivision for Ray Tracing. In: Proceedings of Graphics Interface (GI), pp. 164–72. Canadian Information Processing Society, Toronto, Ontario (1989). Nested grid subdivision structures [50]

[52] Kajiya, J.T.: The rendering equation. In: 13th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), vol. 20, pp. 143–150. ACM, New York, NY, USA (1986). URL http://doi.acm.org/10.1145/15922.15902 [21, 90]

[53] Kaplan, M.R.: Space-Tracing: A Constant Time Ray-Tracer. In: SIGGRAPH '85 State of the Art in Image Synthesis seminar notes, pp. 149–158. ACM, New York, NY, USA (1985) [51]

[54] Kaplanyan, A.: Light Propagation Volumes in CryEngine 3. In: ACM SIGGRAPH Courses. ACM, New York, NY, USA (2009) [xxii, 6, 39, 47, 88, 91, 94, 96, 99, 102, 138, 142, 144, 161]

[55] Kaplanyan, A., Dachsbacher, C.: Cascaded light propagation volumes for real-time indirect illumination. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 99–107. ACM, New York, NY, USA (2010). URL http://doi.acm.org/10.1145/1730804.1730821 [7, 43, 47, 110, 132, 133, 137, 157, 162, 163]

[56] Kaplanyan, A., Engel, W., Dachsbacher, C.: Diffuse Global Illumination with Temporally Coherent Light Propagation Volumes. In: W. Engel (ed.) GPU Pro 2: Advanced Rendering Techniques. A K Peters Ltd (2011) [133]

[57] Karabassi, E.A., Papaioannou, G., Theoharis, T.: A fast depth-buffer-based voxelization algorithm. Journal of Graphics Tools **4**(4), 5–10 (1999) [42]

[58] Karlsson, F.: Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Chalmers Univ. of Technology (2004) [50]

[59] Kavan, L., Bargteil, A.W., Sloan, P.P.: Least Squares Vertex Baking. Computer Graphics Forum (Proceedings of EGSR 2011) **30**(4), 1319–1326 (2011) [32]

[60] Keller, A.: Instant radiosity. In: 24th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 49–56. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1997). URL http://doi.acm.org/10.1145/258734.258769 [45]

[61] Kessenich, J., Baldwin, D., Rost, R.: The OpenGL Shading Language. 3Dlabs, Inc. Ltd., 1.2.8 edn. (2006) [71, 99, 120]

[62] Klimaszewski, K.S., Sederberg, T.W.: Faster Ray Tracing Using Adaptive Grids. IEEE Computer Graphics and Applications **17**(1), 42–51 (1997). URL http://dx.doi.org/10.1109/38.576857 [50]

[63] Kontkanen, J., Laine, S.: Ambient occlusion fields. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 41–48. ACM, New York, NY, USA (2005). URL http://doi.acm.org/10.1145/1053427.1053434 [5, 35, 161]

Athanasios Gaitatzes

[64] Kristensen, A.W., Akenine-Möller, T., Jensen, H.W.: Precomputed local radiance transfer for real-time lighting design. ACM Transactions on Graphics **24**(3), 1208–1215 (2005). URL http://doi.acm.org/10.1145/1073204.1073334 [34]

[65] Křivánek, J., Konttinen, J., Pattanaik, S., Bouatouch, K., Žára, J.: Fast approximation to spherical harmonics rotation. In: ACM SIGGRAPH Sketches, p. 154. ACM, New York, NY, USA (2006). URL http://doi.acm.org/10.1145/1179849.1180042 [97]

[66] Lawlor, O.S., Kalée, L.V.: A voxel-based parallel collision detection algorithm. In: 16th International Conference on Supercomputing (ICS), pp. 285–293. ACM, New York, NY, USA (2002). URL http://doi.acm.org/10.1145/514191.514231 [110, 132]

[67] Loos, B.J., Sloan, P.P.: Volumetric obscurance. In: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D), pp. 151–156. ACM, New York, NY, USA (2010). URL http://doi.acm.org/10.1145/1730804.1730829 [37]

[68] MacDonald, D.J., Booth, K.S.: Heuristics for ray tracing using space subdivision. Visual Computer **6**(3), 153–166 (1990). URL http://dx.doi.org/10.1007/BF01911006 [50]

[69] Malley, T.J.V.: A shading method for computer generated images. Master's thesis, University of Utah (1988) [64]

[70] Malmer, M., Malmer, F., Assarsson, U., Holzschuch, N.: Fast Precomputed Ambient Occlusion for Proximity Shadows. Journal of Graphics Tools **12**(2), 59–71 (2007). URL http://artis.imag.fr/Publications/2007/MMAH07 [5, 36, 161]

[71] Mavridis, P., Gaitatzes, A., Papaioannou, G.: Volume-based Diffuse Global Illumination. In: International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing (CGVCVIP) (2010). URL http://graphics.cs.aueb.gr/graphics/docs/papers/vbgi2010.pdf [9]

[72] Mavridis, P., Papaioannou, G.: Global Illumination using Imperfect Volumes. In: International Conference on Computer Graphics Theory and Applications (GRAPP) (2011). URL http://graphics.cs.aueb.gr/graphics/docs/papers/GRAPP11_ImperfectVolumes.pdf [43, 47, 133, 144]

[73] McGuire, M.: Ambient occlusion volumes. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG), pp. 47–56. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2010). URL http://portal.acm.org/citation.cfm?id=1921479.1921488 [37, 110]

[74] McGuire, M., Luebke, D.: Hardware accelerated global illumination by image space photon mapping. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG), pp. 77–89. ACM, New York, NY, USA (2009). URL http://doi.acm.org/10.1145/1572769.1572783 [44]

[75] Mittring, M.: Finding next gen: CryEngine 2. In: ACM SIGGRAPH Courses, SIGGRAPH '07, pp. 97–121. ACM, New York, NY, USA (2007). URL http://doi.acm.org/10.1145/1281500.1281671 [37]

Athanasios Gaitatzes

[76] Naylor, B.: Constructing Good Partitioning Trees. In: Graphics Interface, pp. 181–191. Canadian Information Processing Society, Toronto, Ontario, Canada (1993) [50]

[77] Nichols, G., Wyman, C.: Interactive Indirect Illumination Using Adaptive Multiresolution Splatting. IEEE Transactions on Visualization and Computer Graphics **16**, 729–741 (2010). URL http://doi.ieeecomputersociety.org/10.1109/TVCG.2009.97 [46]

[78] Nicodemus, F.E., Richmond, J.C., Hsia, J.J., Ginsberg, I.W., Limperis, T.: Geometrical considerations and Nomenclature for Reflectance. Tech. rep. (1977) [20, 21]

[79] Nijasure, M., Pattanaik, S., Goel, V.: Real-time Global Illumination on the GPU. Journal of Graphics Tools **10**(2), 55–71 (2005) [46]

[80] Pantaleoni, J.: VoxelPipe: a programmable pipeline for 3D voxelization. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG), pp. 99–106. ACM, New York, NY, USA (2011) [42, 151]

[81] Papaioannou, G., Menexi, M.L., Papadopoulos, C.: Real-time Volume-Based Ambient Occlusion. IEEE Transactions on Visualization and Computer Graphics **16**, 752–762 (2010). URL http://dx.doi.org/10.1109/TVCG.2010.18 [110, 132]

[82] Parker, S., Martin, W., Sloan, P.P., Shirley, P., Smits, B., Hansen, C.: Interactive ray tracing. In: 1999 Symposium on Interactive 3D Graphics (I3D), pp. 119–126. ACM, New York, NY, USA (1999). URL http://doi.acm.org/10.1145/300523.300537 [50]

[83] Passalis, G., Theoharis, T., Toderici, G., Kakadiaris, I.A.: General Voxelization Algorithm with Scalable GPU Implementation. Journal of Graphics, GPU and Game Tools **12**(1), 61–71 (2007) [42]

[84] Penmatsa, R., Nichols, G., Wyman, C.: Voxel-space ambient occlusion. In: Symposium on Interactive 3D Graphics and Games (I3D). ACM, New York, NY, USA (2010). URL http://doi.acm.org/10.1145/1730971.1730989 [110]

[85] Pharr, M., Humphreys, G.: Physically Based Rendering: From Theory to Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004) [20]

[86] Policarpo, F.: Deferred Shading Tutorial. Techniques **31**, 32 (2005). URL http://www710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2007/Deferred_Shading_Tutorial_SBGAMES2005.pdf [30]

[87] Popov, S., Günther, J., Seidel, H.P., Slusallek, P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. Computer Graphics Forum **26**(3), 415–424 (2007). URL http://dx.doi.org/10.1111/j.1467-8659.2007.01064.x [51]

[88] Purcell, T.J.: Ray Tracing on a Stream Processor. Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA (2004) [50]

[89] Purcell, T.J., Buck, I., Mark, W.R., Hanrahan, P.: Ray tracing on programmable graphics hardware. In: 29th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), vol. 21, pp. 703–712. ACM, New York, NY, USA (2002). URL http://doi.acm.org/10.1145/566570.566640 [50]

Athanasios Gaitatzes

[90] Ramamoorthi, R., Hanrahan, P.: An efficient representation for irradiance environment maps. In: 28th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 497–500. ACM, New York, NY, USA (2001). URL http://doi.acm.org/10.1145/383259.383317 [120]

[91] Ritschel, T., Grosch, T., Kim, M.H., Seidel, H.P., Dachsbacher, C., Kautz, J.: Imperfect shadow maps for efficient computation of indirect illumination. In: ACM SIGGRAPH Asia Papers, vol. 27, pp. 1–8. ACM, New York, NY, USA (2008). URL http://doi.acm.org/10.1145/1409060.1409082 [46]

[92] Ritschel, T., Grosch, T., Seidel, H.P.: Approximating dynamic global illumination in image space. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 75–82. ACM, New York, NY, USA (2009). URL http://doi.acm.org/10.1145/1507149.1507161 [46]

[93] Rubin, S.M., Whitted, T.: A 3-dimensional representation for fast rendering of complex scenes. In: 7th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 110–116. ACM, New York, NY, USA (1980). URL http://doi.acm.org/10.1145/800250.807479 [50]

[94] Saito, T., Takahashi, T.: Comprehensible rendering of 3-D shapes. In: 17th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 197–206. ACM, New York, NY, USA (1990). URL http://doi.acm.org/10.1145/97879.97901 [32]

[95] Schwarz, M., Seidel, H.P.: Fast parallel surface and solid voxelization on GPUs. In: ACM SIGGRAPH Asia Papers, SIGGRAPH ASIA '10, pp. 179:1–179:10. ACM, New York, NY, USA (2010). URL http://doi.acm.org/10.1145/1866158.1866201 [40, 42]

[96] Segal, M., Akeley, K.: The OpenGL Graphics System: A Specification (Version 1.2). Computer (1998) [23]

[97] Shanmugam, P., Arikan, O.: Hardware accelerated ambient occlusion techniques on GPUs. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 73–80. ACM, New York, NY, USA (2007). URL http://doi.acm.org/10.1145/1230100.1230113 [36, 46]

[98] Shirley, P., Chiu, K.: A low distortion map between disk and square. Journal of Graphics Tools **2**(3), 45–52 (1997) [64]

[99] Slater, M.: Constant time queries on uniformly distributed points on a hemisphere. Journal of Graphics Tools **7**(1), 33–44 (2002) [63]

[100] Sloan, P.P., Kautz, J., Snyder, J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In: 29th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 527–536. ACM, New York, NY, USA (2002). URL http://doi.acm.org/10.1145/566570.566612 [33, 91, 120, 144]

[101] Snyder, J., Barr, A.H.: Ray tracing complex models containing surface tessellations. In: 14th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), vol. 21, pp. 119–128. ACM, New York, NY, USA (1987). URL http://doi.acm.org/10.1145/37401.37417 [50]


Athanasios Gaitatzes

[102] Software, P.: Video Card Benchmarks (2012). URL http://www.videocardbenchmark.net/high_end_gpus.html. [Online; accessed 18-April-2012] [151]

[103] Sud, A., Govindaraju, N., Gayle, R., Manocha, D.: Interactive 3D distance field computation using linear factorization. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 117–124. ACM, New York, NY, USA (2006). URL http://doi.acm.org/10.1145/1111411.1111432 [38]

[104] Sud, A., Otaduy, M.A., Manocha, D.: DiFi: Fast 3D Distance Field Computation Using Graphics Hardware. Computer Graphics Forum **23**(3), 557–566 (2004) [38]

[105] Sung, K., Shirley, P.: Ray tracing with the BSP tree. In: Graphics Gems III, pp. 271–274. Academic Press Professional, Inc., San Diego, CA, USA (1992) [50]

[106] Szirmay-Kalos, L., Umenhoffer, T., Toth, B., Szecsi, L., Sbert, M.: Volumetric Ambient Occlusion for Real-Time Rendering and Games. IEEE Computer Graphics and Applications **30**, 70–79 (2010). URL http://doi.ieeecomputersociety.org/10.1109/MCG.2010.19 [37, 110]

[107] Theoharis, T., Papaioannou, G., Platis, N., Patrikalakis, N.M.: Graphics and Visualization: Principles & Algorithms. A. K. Peters, Ltd., Natick, MA, USA (2007) [20]

[108] Thiedemann, S., Henrich, N., Grosch, T., Müller, S.: Voxel-based global illumination. In: Symposium on Interactive 3D Graphics and Games (I3D), pp. 103–110. ACM, New York, NY, USA (2011). URL http://doi.acm.org/10.1145/1944745.1944763 [42, 47, 110, 132, 133, 144]

[109] Thrane, N., Simonsen, L.O.: A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, University of Aarhus, Denmark (2005) [51]

[110] Wald, I., Slusallek, P., Benthin, C., Wagner, M.: Interactive Distributed Ray Tracing of Highly Complex Models. In: 12th Eurographics Workshop on Rendering Techniques (EGSR), pp. 277–288. Springer-Verlag, London, UK (2001) [50]

[111] Wald, I., Slusallek, P., Benthin, C., Wagner, M.: Interactive Rendering with Coherent Ray Tracing. Computer Graphics Forum **20**(3), 153164 (2001) [50]

[112] Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., Greenberg, D.P.: Lightcuts: a scalable approach to illumination. ACM Transactions on Graphics **24**(3), 1098–1107 (2005). URL http://doi.acm.org/10.1145/1186822.1073318 [45]

[113] Wang, R., Wang, R., Zhou, K., Pan, M., Bao, H.: An efficient GPU-based approach for interactive global illumination. In: ACM SIGGRAPH Papers, pp. 1–8. ACM, New York, NY, USA (2009). URL http://doi.acm.org/10.1145/1576246.1531397 [46]

[114] Ward, G.J., Rubinstein, F.M., Clear, R.D.: A ray tracing solution for diffuse interreflection. In: 15th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 85–92. ACM, New York, NY, USA (1988). URL http://doi.acm.org/10.1145/54852.378490 [46]

Athanasios Gaitatzes

[115] Whitted, T.: An improved illumination model for shaded display. Communications of the ACM **23**(6), 343–349 (1980). URL http://doi.acm.org/10.1145/358876.358882 [48, 49]

[116] Wikipedia: GeForce 256 — Wikipedia, The Free Encyclopedia (2002). URL http://en.wikipedia.org/wiki/GeForce_256. [Online; accessed 18-April-2012] [23]

[117] Wikipedia: Microsoft Direct3D — Wikipedia, The Free Encyclopedia (2002). URL http://en.wikipedia.org/wiki/Microsoft_Direct3D. [Online; accessed 18-April-2012] [23]

[118] Wikipedia: GeForce 3 Series — Wikipedia, The Free Encyclopedia (2003). URL http://en.wikipedia.org/wiki/GeForce_3_Series. [Online; accessed 18-April-2012] [23]

[119] Wikipedia: GeForce 200 Series — Wikipedia, The Free Encyclopedia (2009). URL http://en.wikipedia.org/wiki/GeForce_200_Series. [Online; accessed 18-April-2012] [151]

[120] Wikipedia: GeForce 400 Series — Wikipedia, The Free Encyclopedia (2010). URL http://en.wikipedia.org/wiki/GeForce_400_Series. [Online; accessed 18-April-2012] [23, 151]

[121] Wikipedia: Comparison of Nvidia Graphics Processing Units — Wikipedia, The Free Encyclopedia (2012). URL http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units. [Online; accessed 18-April-2012] [151]

[122] Zhang, L., Chen, W., Ebert, D.S., Peng, Q.: Conservative voxelization. Visual Computer **23**, 783–792 (2007). URL http://portal.acm.org/citation.cfm?id=1283953.1283975 [41]

[123] Zhou, K., Hu, Y., Lin, S., Guo, B., Shum, H.Y.: Precomputed shadow fields for dynamic scenes. ACM Transactions on Graphics **24**(3), 1196–1201 (2005). URL http://doi.acm.org/10.1145/1073204.1073332 [35]

[124] Zhukov, S., Iones, A., Kronin, G.: An Ambient Light Illumination Model. In: G. Drettakis, N. Max (eds.) 9th Eurographics Workshop on Rendering Techniques (EGSR), pp. 45–56. Springer-Verlag Wien New York (1998) [35]

Athanasios Gaitatzes

# Author Index

Akeley, Kurt . . . 23
Akenine-Möller, Tomas . . . 23, 34
Amit, Ben-David . . . 66
Andreadis, Anthousis . . . 9
Arbree, Adam . . . 45
Arikan, Okan . . . 36, 46
Arvo, James . . . 50
Assarsson, Ulf . . . 5, 36, 161

Bala, Kavita . . . 20, 45
Baldwin, Dave . . . 71, 99, 120
Bao, Hujun . . . 41, 42, 46
Bargteil, Adam W. . . . 32
Barr, Alan H. . . . 50
Barthe, Loic . . . 40, 41
Battaile, Bennett . . . 39
Bavoil, Louis . . . 37
Bekaert, Philippe . . . 20
Benthin, Carsten . . . 50
Bittner, Jiri . . . 50
Blythe, David . . . 23
Booth, Kellogg S. . . . 50
Bouatouch, Kadi . . . 97
Buck, Ian . . . 50
Bunnell, Michael . . . 36

Carr, Nathan A. . . . 50, 51
Cazals, Frédéric . . . 50
Chatelier, Pierre Y. . . . 39
Chen, Hongsheng . . . 6, 41, 94, 124, 161
Chen, Wei . . . 41, 42
Chiu, Kenneth . . . 64
Christen, Martin . . . 50
Chrysanthou, Yiorgos . . . 9
Clark, James H. . . . 50
Clear, Robert D. . . . 46
Cook, Robert L. . . . 49
Crane, Keenan . . . 41, 51, 110, 132
Crassin, Cyril . . . 40, 162

Athanasios Gaitatzes

Dachsbacher, Carsten . . . 7, 43, 45–47, 88, 110, 132, 133, 137, 157, 162, 163
Décoret, Xavier . . . . . . . . . 40–42, 94, 125, 137
Dimitrov, Rouslan . . . . . . . . . . . . . . . . . . . . . . 37
Dong, Zhao . . . . . . . . . . . . . . . . . . . . . . . . 41, 42
Donikian, Michael . . . . . . . . . . . . . . . . . . . . . . 45
Drettakis, George . . . . . . . . . . . . . . . . . . . . . . 50
Dutre, Philip . . . . . . . . . . . . . . . . . . . . . . . . . . 20

Ebert, David S. . . . . . . . . . . . . . . . . . . . . . . . . 41
Eisemann, Elmar . . . . 40–42, 94, 125, 137, 162
Engel, Wolfgang . . . . . . . . . . . . . . . . . . . 50, 133
Ernst, Manfred . . . . . . . . . . . . . . . . . . . . . . . . 50

Fang, Shiaofen . . . . . . . . . . . 6, 41, 94, 124, 161
Fernandez, Sebastian . . . . . . . . . . . . . . . . . . . 45
Foley, Tim . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Forest, Vincent . . . . . . . . . . . . . . . . . . . . . 40, 41
Fujimoto, Akira . . . . . . . . . . . . . . . . . . . . . . . . 50

Gaitatzes, Athanasios . . . . . . . . . . . 9, 152, 153
Gayle, Russell . . . . . . . . . . . . . . . . . . . . . . . . . 38
Geiss, Ryan . . . . . . . . . . . . . . . . . . . . . 110, 132
Gigahertz-Optik, Inc. . . . . . . . . . . . . . . . . . . . 20
Ginsberg, I. W. . . . . . . . . . . . . . . . . . . . . . 20, 21
Glassner, A. S. . . . . . . . . . . . . . . . . . . . . . . . . 50
Goel, Vineet . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
Goldsmith, Jeffrey . . . . . . . . . . . . . . . . . . . . . . 50
Goral, Cindy M. . . . . . . . . . . . . . . . . . . . . . . . . 39
Govindaraju, Naga . . . . . . . . . . . . . . . . . . . . . . 38
Green, Robin . . . . . . . . . . . . . . . . . . . . . . . . . 120
Greenberg, Donald P. . . . . . . . . . . . . . . 39, 44, 45
Greger, Gene . . . . . . . . . . . . . . . . . . . . . . . 39, 44
Greiner, Günther . . . . . . . . . . . . . . . . . . . . . . . 50
Grosch, Thorsten . . . 42, 46, 47, 110, 132, 133, 144
Günther, Johannes . . . . . . . . . . . . . . . . . . . 51, 66
Guo, Baining . . . . . . . . . . . . . . . . . . . . . . . . . . 35

Haines, Eric . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
Hall, Jesse D. . . . . . . . . . . . . . . . . . . . . . . . . . 50
Hanrahan, Pat . . . . . . . . . . . . . . . 50, 51, 74, 120
Hansen, Charles . . . . . . . . . . . . . . . . . . . . . . . . 50
Harada, Takahiro . . . . . . . . . . . . . . . . . . 110, 132
Hart, John C. . . . . . . . . . . . . . . . . . . . . . . . 50, 51
Havran, Vlastimil . . . . . . . . . . . . . . . . . . . . . . . 50
Henrich, Niklas . . . . 42, 47, 110, 132, 133, 144
Hoberock, Jared . . . . . . . . . . . . . . . . . . . . . . . 51
Hoffman, Naty . . . . . . . . . . . . . . . . . . . . . . . . . 23
Holzschuch, Nicolas . . . . . . . . . . . . . . 5, 36, 161

Athanasios Gaitatzes

Horn, Daniel Reiter . . . . . 51, 74
Houston, Mike . . . . . 51, 74
Hsia, J. J. . . . . . 20, 21
Hu, Yaohua . . . . . 35
Huang, Peijie . . . . . 38, 59
Hubbard, Philip M. . . . . . 39, 44
Humphreys, Greg . . . . . 20

Illingworth, Valerie . . . . . 20
Iones, Andrej . . . . . 35
Iwata, Kansei . . . . . 50

Jansen, Frederik W . . . . . 50
Jensen, Henrik Wann . . . . . 34, 43
Jevans, David . . . . . 50

Kajiya, James T. . . . . . 21, 90
Kakadiaris, Ioannis A. . . . . . 42
Kalée, Laxmikant V. . . . . . 110, 132
Kaplan, Michael R. . . . . . 51
Kaplanyan, Anton . . . . . xxii, 6, 7, 39, 43, 47, 88, 91, 94, 96, 99, 102, 110, 132, 133, 137, 138, 142, 144, 157, 161–163
Karabassi, Evaggelia-Aggeliki . . . . . 42
Karlsson, Filip . . . . . 50
Kautz, Jan . . . . . 33, 46, 91, 120, 144
Kavan, Ladislav . . . . . 32
Keller, Alexander . . . . . 45
Kessenich, John . . . . . 71, 99, 120
Kim, M. H. . . . . . 46
Klimaszewski, Krzysztof S. . . . . . 50
Kontkanen, Janne . . . . . 5, 35, 161
Konttinen, Jaakko . . . . . 97
Kristensen, Anders Wang . . . . . 34
Kronin, Grigorij . . . . . 35
Krupkin, Anton . . . . . 35
Křivánek, Jaroslav . . . . . 97

Laine, Samuli . . . . . 5, 35, 161
Lawlor, Orion Sky . . . . . 110, 132
Lefebvre, Sylvain . . . . . 40
Limperis, T. . . . . . 20, 21
Lin, Stephen . . . . . 35
Llamas, Ignacio . . . . . 41, 110, 132
Loos, Bradford James . . . . . 37
Luebke, David . . . . . 44

MacDonald, David J. . . . . . 50
Malgouyres, Rémy . . . . . 39
Malley, Thomas J. V. . . . . . 64
Malmer, Fredrik . . . . . 5, 36, 161

Athanasios Gaitatzes

Malmer, Mattias . . . . . . . . . . . . . . . . . . 5, 36, 161
Manocha, Dinesh . . . . . . . . . . . . . . . . . . . . . . . 38
Mark, William R. . . . . . . . . . . . . . . . . . . . . . . . 50
Martin, William . . . . . . . . . . . . . . . . . . . . . . . . 50
Mavridis, Pavlos . 9, 43, 47, 133, 144, 152, 153
McGuire, Morgan . . . . . . . . . . . . . . . 37, 44, 110
Menexi, Maria Lida . . . . . . . . . . . . . . . 110, 132
Mittring, Martin . . . . . . . . . . . . . . . . . . . . . . . 37
Müller, Stefan . . . . . 42, 47, 110, 132, 133, 144

Naylor, Bruce . . . . . . . . . . . . . . . . . . . . . . . . . 50
Neyret, Fabrice . . . . . . . . . . . . . . . . . . . . 40, 162
Nichols, Greg . . . . . . . . . . . . . . . . . . . . . . 46, 110
Nicodemus, F. E. . . . . . . . . . . . . . . . . . . . . 20, 21
Nijasure, Mangesh . . . . . . . . . . . . . . . . . . . . . 46

Otaduy, Miguel A. . . . . . . . . . . . . . . . . . . . . . 38

Pan, Minghao . . . . . . . . . . . . . . . . . . . . . . . . . 46
Pantaleoni, Jacopo . . . . . . . . . . . . . . . . . 42, 151
Papadopoulos, Charilaos . . . . . . . . . . . 110, 132
Papaioannou, Georgios . 9, 20, 42, 43, 47, 110, 132, 133, 144, 152, 153
Parker, Steven . . . . . . . . . . . . . . . . . . . . . . . . . 50
Passalis, Georgios . . . . . . . . . . . . . . . . . . . . . 42
Patrikalakis, N. M. . . . . . . . . . . . . . . . . . . . . . 20
Pattanaik, Sumanta . . . . . . . . . . . . . . . . . . 46, 97
Paulin, Mathias . . . . . . . . . . . . . . . . . . . . . 40, 41
Peng, Qunsheng . . . . . . . . . . . . . . . . . . . . . 41, 42
Penmatsa, Rajeev . . . . . . . . . . . . . . . . . . . . . 110
Pharr, Matt . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
Platis, N. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
Policarpo, Fabio . . . . . . . . . . . . . . . . . . . . . . . 30
Popov, Stefan . . . . . . . . . . . . . . . . . . . . . . . 51, 66
Puech, Claude . . . . . . . . . . . . . . . . . . . . . . . . . 50
Purcell, Timothy John . . . . . . . . . . . . . . . . . . . 50

Ramamoorthi, Ravi . . . . . . . . . . . . . . . . . . . . 120
Richmond, J. C. . . . . . . . . . . . . . . . . . . . . . 20, 21
Ritschel, Tobias . . . . . . . . . . . . . . . . . . . . . . . . 46
Rost, Randi . . . . . . . . . . . . . . . . . . . . . 71, 99, 120
Rubin, Steven M. . . . . . . . . . . . . . . . . . . . . . . . 50
Rubinstein, Francis M. . . . . . . . . . . . . . . . . . . 46

Sainz, Miguel . . . . . . . . . . . . . . . . . . . . . . 37, 162
Saito, Takafumi . . . . . . . . . . . . . . . . . . . . . . . . 32
Salmon, John . . . . . . . . . . . . . . . . . . . . . . . . . . 50
Sbert, Mateu . . . . . . . . . . . . . . . . . . . . 35, 37, 110
Schwarz, Michael . . . . . . . . . . . . . . . . . . . . 40, 42
Sederberg, Thomas W. . . . . . . . . . . . . . . . . . . 50
Segal, Mark . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

Athanasios Gaitatzes

Seidel, Hans-Peter ......... 40, 42, 46, 51, 66
Shanmugam, Perumaal ............... 36, 46
Shirley, Peter ................. 39, 44, 50, 64
Shum, Heung-Yeung ..................... 35
Simonsen, Lars Ole ...................... 51
Slater, Mel ........................... 63
Sloan, Peter-Pike . 32, 33, 37, 50, 91, 120, 144
Slusallek, Philipp ................. 50, 51, 66
Smits, Brian .......................... 50
Snyder, John ............ 33, 50, 91, 120, 144
Software, PassMark .................. 151
Stamminger, Marc ....... 45, 46, 88, 133, 137
Sud, Avneesh ......................... 38
Sugerman, Jeremy ................... 51, 74
Sung, Kelvin ......................... 50
Szecsi, László ................... 37, 110
Szirmay-Kalos, László .............. 37, 110

Takahashi, Tokiichiro .................. 32
Tanaka, Takayuki ..................... 50
Tariq, Sarah ................... 41, 110, 132
Theoharis, Theoharis ................ 20, 42
Thiedemann, Sinje . 42, 47, 110, 132, 133, 144
Thrane, Niels ......................... 51
Toderici, George ....................... 42
Torrance, Kenneth E. .................. 39
Toth, Balazs ..................... 37, 110

Umenhoffer, Tamas ................ 37, 110

Vogelgsang, Christian .................. 50
Žára, Jiří ............................. 97

Wagner, Markus ....................... 50
Wald, Ingo ........................... 50
Walter, Bruce ......................... 45
Wang, Rui ........................... 46
Wang, Wencheng .................. 38, 59
Ward, Gregory J. ...................... 46
Whitted, Turner .................... 48–50
Wikipedia ....................... 23, 151
Wu, Enhua ........................ 38, 59
Wyman, Chris .................... 46, 110
Wyvill, Brian ......................... 50

Yang, Gang ....................... 38, 59

Zhang, Hongxin ................... 41, 42
Zhang, Long .......................... 41
Zhou, Kun ........................ 35, 46
Zhukov, Sergej ........................ 35

Athanasios Gaitatzes