Habilitationsschrift

Real-Time Rendering

ausgeführt in den Jahren 2001–2007
zum Zwecke der Erlangung der *venia docendi* (Lehrbefugnis)
im Habilitationsfach "Praktische Informatik"

eingereicht im Mai 2007
an der Technischen Universität Wien, Österreich,
Fakultät für Informatik,

von Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer,

Gersthofer Strasse 140/1/7, A-1180 Wien, Österreich,
geboren am 23. Dezember 1973 in Wien,
`wimmer@cg.tuwien.ac.at`.

Wien, im Mai 2007.

Contents

1 **Introduction** 1
1.1 Overview 1
1.2 Introduction to Real-Time Rendering 1
1.2.1 What is Real Time? 2
1.2.2 Improving Performance 3
1.2.3 Improving Quality 5
1.2.4 Point-Based Rendering 7
1.2.5 Conclusion 7
1.3 List of Selected Papers 8
1.4 Overview of the Selected Papers and Contributions of the Author 9
1.4.1 Chapter 2: Rendering Time Estimation for Real-Time Rendering 9
1.4.2 Chapter 3: Light Space Perspective Shadow Maps 10
1.4.3 Chapter 4: Hardware Occlusion Queries Made Useful 10
1.4.4 Chapter 5: Guided Visibility Sampling 10
1.4.5 Chapter 6: Instant Points 11

2 **Rendering Time Estimation for Real-Time Rendering** 13
2.1 Introduction 15
2.2 Previous Work 17
2.3 Rendering Hardware Overview 18
2.4 The rendering time estimation framework 19
2.4.1 The refined rendering time estimation function 19
2.4.2 System tasks 20
2.4.3 CPU tasks 22
2.4.4 Idle time 22
2.4.5 GPU tasks 24
2.5 Methods for rendering time estimation 27
2.5.1 View-cell sampling 28
2.5.2 Per-object sampling 28
2.5.3 Mathematical heuristics 29
2.6 Hardware extensions for a soft real-time system 31
2.6.1 The timing accuracy problem 31

2.6.2 The time-stamping extension 31
2.6.3 The estimation accuracy problem 32
2.6.4 The conditional branching extension 33
2.6.5 Soft real-time system 33
2.7 Implementation and results 34
2.7.1 Test setup and models 34
2.7.2 Comparing the per-viewpoint estimation methods 35
2.7.3 Comparing the per-view-cell estimation methods 37
2.7.4 Hardware extension 37
2.7.5 Discussion 37
2.8 Conclusions 38

3 Light Space Perspective Shadow Maps **39**
3.1 Introduction 41
3.2 Previous work 43
3.3 Light space perspective shadow maps 44
3.3.1 Motivation 44
3.3.2 Overview 44
3.3.3 Focussing the shadow map 45
3.3.4 The perspective frustum in light space 46
3.3.5 Choosing the free parameter *n* 47
3.3.6 Applying the perspective frustum 47
3.4 Analysis and optimal parameter estimation 47
3.4.1 What is perspective aliasing 48
3.4.2 Logarithmic shadow mapping 50
3.4.3 Analysis of light space perspective shadow maps 50
3.4.4 General case 53
3.4.5 Discussion 54
3.5 Results 56
3.6 Conclusions and future work 57

4 Hardware Occlusion Queries Made Useful **59**
4.1 Introduction 61
4.2 For Which Scenes Are Occlusion Queries Effective? 61
4.3 What Is Occlusion Culling? 62
4.4 Hierarchical Stop-and-Wait Method 64
4.4.1 The Naive Algorithm, or Why Use Hierarchies at All? . . 64
4.4.2 Hierarchies to the Rescue! 64
4.4.3 Hierarchical Algorithm 65
4.4.4 Problem 1: Stalls 65
4.4.5 Problem 2: Query Overhead 66
4.5 Coherent Hierarchical Culling 66
4.5.1 Idea 1: Being Smart and Guessing 66
4.5.2 Idea 2: Pull Up, Pull Up 67

4.5.3 Algorithm 68
4.5.4 Implementation Details 69
4.5.5 Why Are There Fewer Stalls? 70
4.5.6 Why Are There Fewer Queries? 72
4.5.7 How to Traverse the Hierarchy 73
4.6 Optimizations 73
4.6.1 Querying with Actual Geometry 73
4.6.2 Z-Only Rendering Pass 74
4.6.3 Approximate Visibility 74
4.6.4 Conservative Visibility Testing 74
4.7 Conclusion 75

5 Guided Visibility Sampling **77**
5.1 Introduction 79
5.2 Overview 82
5.2.1 Problem Statement 82
5.2.2 Algorithm Overview 83
5.3 Visibility Sampling 83
5.3.1 Random Sample Generator 83
5.3.2 Adaptive Border Sampling 84
5.3.3 Reverse Sampling 86
5.3.4 Combining the Different Sampling Algorithms 87
5.3.5 Termination criteria 88
5.4 Results 88
5.4.1 Overview 88
5.4.2 Asymptotic behavior 89
5.4.3 Practical results 91
5.4.4 Comparison to hardware sampling 92
5.4.5 Comparison to exact visibility 93
5.5 Related Work, Discussion and Applications 94
5.5.1 Exact Visibility 94
5.5.2 Conservative Visibility 94
5.5.3 Aggressive Visibility 95
5.5.4 Algorithm Analysis 95
5.5.5 Limitations and Future Work 97
5.5.6 Applications 97
5.6 Conclusion 98

6 Instant Points **101**
6.1 Introduction 103
6.2 Previous Work 105
6.3 The Instant Points Rendering System 106
6.4 Memory Optimized Sequential Point Trees 106
6.4.1 Sequential Point Trees Revisited 106

6.4.2 Screen Splat Error Metric 107
6.4.3 SPT overhead 107
6.4.4 Memory optimization 108
6.4.5 Rendering MOSPTs 109
6.4.6 MOSPT Creation 110
6.5 Nested Octrees 111
6.5.1 Motivation and Definition 111
6.5.2 Hierarchy Creation 113
6.5.3 Rendering 114
6.6 Results 115
6.7 Conclusions and Future Work 117

Bibliography **119**

Acknowledgements **133**

Chapter 1

Introduction

1.1 Overview

This thesis presents a subset of my research work in the field of real-time rendering. The current chapter serves as an introduction to and survey of this field, providing the context for the papers presented in the following chapters (Section 1.2). Then the papers making up the body of the thesis are listed (Section 1.3) and briefly described (Section 1.4), with an emphasis on the contribution the author made to each individual work.

1.2 Introduction to Real-Time Rendering

Computer graphics is an ever evolving field. Years ago, researchers concentrated on rendering appealing still images. Ray tracing and radiosity were developed to create images which mimick real photographs, giving rise to “photorealistic rendering”. Applications of photorealistic rendering are manifold, including lighting simulation of planned buildings, creating artwork, and not the least of them is rendering special effects (e.g., explosions, atmospheric effects and many more) and whole image sequences in the movie industry.

However, the rapid increase in computational power made it possible to look towards interactive image generation, which was important for many application areas. This led to a new research direction, which, in its most general definition, concerns itself with techniques for the interactive generation of computer graphics images: “real-time rendering”. Flight simulation, driving simulation, architectural walkthroughs, interactive modeling packages, virtual reality and computer games are among the applications that rely on or were made possible in the first place by the advances made in real-time rendering.

Real-time rendering is a term that is nowadays commonly connected with commodity hardware available to every consumer. It should be noted, however, that not too long ago, 3D graphics on a consumer PC was plainly impossible. It was the introduction of the 3DFX Voodoo graphics card in 1997 [Leht00] which started an enormous development frenzy, so that nowadays, 3D consumer graphics cards from NVIDIA and AMD (formerly ATI) surpass the capabilities of most dedicated graphics workstations. In many ways, during these 10 years we have witnessed the coming of age of the discipline of computer graphics. Interestingly enough, it was the entertainment industry, foremost of all computer games, that made this development possible, by opening the consumer market to this new technology.

There are two possible directions for real-time rendering research, to both of which this thesis makes contributions:

- improving the performance of rendering algorithms, so that more complex scenes can be displayed, and
- improving the quality of the images that are displayed.

1.2.1 What is Real Time?

A topic that encompasses both of the above mentioned research directions is the question what computer graphics researcher understand by the term “real time.” In the context of real-time computing, a system is said to be real-time if the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed. In a hard or immediate real-time system, the completion of an operation after its deadline is considered useless—ultimately, this may lead to a critical failure of the complete system. A soft real-time system on the other hand will tolerate such lateness, and may respond with decreased service quality (e.g., dropping frames).

Obviously, a computer graphics application can hardly fulfill the requirements of a hard real-time system, as it depends on many factors outside of the programmer’s control (for example non-real-time operating system and drivers). Instead, we have to be satisfied with a soft real-time system in which glitches in response time can occur.

A definition that is more adapted to the goals of computer graphics is that real-time rendering is concerned with the display of computer-generated images at rates which let a human observer believe that she is looking at a smooth animation. The most important factor in this respect is therefore the *frame rate*, i.e., the number of displayed images per second. The frame rate is tightly coupled to the update rate of the display device. TV screens run at either 50 Hz (PAL) or 60 Hz (NTSC), and although the human eye can perceive flicker at this frequency and will weary from prolonged watching, many applications are targeted towards systems using TV screens (e.g., arcade and console games). Computer CRTs usually run at a comfortable 85 Hz, which is well above the flicker limit of the human eye. While on the one hand, these frequencies set an upper limit on the frame rates a real-time rendering system has to achieve, and on the other hand, it is generally agreed that little more than 20 frames per second (fps) are sufficient to generate the illusion of motion (24 fps is used in motion theaters, for example), it has been discovered that frame rates below the display update rate lead to significant artifacts. If the rendering system cannot provide a new image for every display refresh, the same image will be displayed several times before being changed. This results in unnatural choppy motion and confuses the human visual system into seeing distracting “ghost images”, especially noticeable at sharp edges in the image (like, for example, a building border) [Helm94]. These artifacts are perceivable mainly in display technologies which are inherently capable of such high refresh rates. Alternative display systems, mainly LCDs, have significantly lower refresh rates than CRTs. Moreover, the persistence of images on an LCD is much larger than on a CRT, so that smearing and blurring artifacts occur in continuous motion. For this reason, traditional LCD screens are not really suitable for real-time applications. However, since LCDs are quickly becoming the prevalent TV technology, manufacturers invest enormous resources into improving these aspects of the LCD technology, and solutions like 120 Hz LCDs and displays with pulsed backlights are already in prototype stage.

Apart from the value of the frame rate itself, the consistency of the frame rate plays an equally important role in providing a high-quality real-time rendering system, as frame-rate drops are easily perceivable as stutters and jerks, which are often more annoying to the user than a generally lower overall frame rate. For this reason, significant research has been conducted into the area of providing real-time rendering systems with bounded frame rates [Funk93], which typically make use of level-of-detail techniques (see next section). Chapter 2 extends this research by handling the crucial question of how to predict the frame rate of a given scene, which is an important prerequisite to being able to control the scene complexity in a way to reach a desired frame rate.

Finally, it should be noted that the growth in scene complexities desired by the users is so quick that for any state-of-the art real-time system a scene can be found that overwhelms the capabilities of the system. Thus, real-time rendering will stay an important research topic in the foreseeable future.

1.2.2 Improving Performance

The improvement of rendering performance is maybe the core area of real-time rendering from its beginning. Faster rendering allows for more complex scenes and more detailed objects, which also leads to improved quality.

Visibility culling. Starting from the description of a complex 3D model which cannot be rendered at interactive frame rates on a current rendering system, the first obvious step is to reduce the amount of data that has to be processed. Fortunately, in most applications of real-time rendering, the observer can only see a small subset of the whole scene at any given time. Most of the scene is usually hidden by occluding geometry like building walls or terrain. Therefore, significant processing power can be saved by identifying those parts of the scene which do not contribute to the final image and remove them from the rendering process. This is called *visibility culling*. It is important that the effort invested into visibility culling does not surpass the gains by not rendering invisible geometry. Therefore, a promising strategy is to *precompute* information about visibility and only do constant-time lookups at runtime (presented in Chapter 5). Alternatively, we show a method to exploit a graphics hardware feature to practically eliminate the visibility culling costs, making the algorithm feasible for runtime computation (presented in Chapter 4). Visibility culling has been one of the main research foci in the career of the author, and besides the two contributions presented in this thesis, he has helped advance the field through many other publications [Wonk00, Wonk01, Bitt01, Bitt04, Bitt05, Matt06, Matt07].

Levels of detail. In many applications, the amount of geometry that remains after visibility culling and is therefore actually visible to the observer still overwhelms the capabilities of current rendering systems. Fortunately, the complexity of the scene can be reduced: due to the perspective foreshortening effect in perspective cameras, many objects project to a small area on screen, and therefore do not need to be displayed using their full geometric description (which is often designed to allow even close-up views). This leads to the idea of *level-of-detail* (LOD) rendering, in which objects have several geometric representations at different levels of detail. While the author has worked on level-of-detail methods [Wimm98, Gieg07c], this thesis will not cover this specific area of real-time rendering.

Image-based rendering. Geometric LODs are very good at reducing the detail in highly detailed geometric objects, e.g., objects acquired using laser scanners or modeled using freeform surfaces. However, there are scenes consisting of many smaller, disconnected objects, or objects that defy geometric simplification due to their topological complexity. For these cases it has been suggested to use *image-based rendering* techniques, which work by creating sampling-based representations of the objects to be simplified. The complexity of these sampling-based representations is independent of the original object, and only depends on the desired output resolution. Image-based rendering is therefore well suited to achieve a desired frame rate independent of the complexity of the objects. A commonly used technique is to replace a geometric object by a quadrilateral polygon, a so-called *impostor*, that is mapped with an image (texture) of the object. Impostors can be generated on the fly or precomputed, in which case storage cost quickly becomes an issue. Alternatively, it is very popular to use image-based rendering already at the modeling stage. For example, a highly detailed geometric object is simplified using LOD techniques, and the simplified version is mapped with a texture that encodes the response of the surface to lighting conditions, thereby allowing to simulate surface detail by shading effects. These techniques gain in importance with respect to simple impostor methods, because the increasing use of advanced shading effects in current applications makes it difficult to compute faithful image-based representations that capture these effects at runtime. The field of image-based rendering is vast, and although the author has contributed to this field [Wimm99a, Wimm99b, Wimm01, Jesc02a, Jesc02b, Jesc05, Jesc07], but the area is not topic of this thesis.

1.2.3 Improving Quality

Originally, interactive rendering systems were limited to very simple appearance models that were hard-coded into the available graphics accelerators. The Phong illumination model [Phon75] allowed the modeling of diffuse and specular reflections, giving most computer-generated images a characteristic look of shiny plastic. This was mostly combined with a single texture map to capture fine surface structures. Furthermore, illumination was completely *local*, i.e., the shading (computed color) at a point depends only on the surface properties of the point itself and on a fixed number of light sources.

A recent paradigm shift in graphics hardware has laid the groundwork for fundamental changes in the quality of real-time rendering. Instead of hard-wiring shading models in the so-called fixed function pipeline, graphics hardware has gradually (partially in 2001, more so in 2003) started to offer full programmability for all its functional units.

This immediately opens the door for *more realistic local illumination models*, allowing the rendering of skin, fur, metallic surfaces etc., replacing the original Phong model. The author has contributed in the development of a new model for the illumination of plant leaves, allowing for realistic shading due to the direct illumination from the sun [Habe07] (not in this thesis).

Furthermore, especially the capability to perform arbitrary calculations for each pixel on screen, led to the development of full-screen postprocessing algorithms. Most well-known are so-called *high dynamic range* (HDR) rendering effects, in which the full dynamic range (many orders of magnitude) of natural illumination is taken into account in the computation of illumination Only in a final step, called *tone mapping*, this range is mapped to the output range of the display device, taking into account eye accommodation. The author has contributed to tone-mapping research [Artu03, Čadí06], but it is beyond the scope of this thesis. Another interesting HDR effect nowadays typically found in computer games is glare (pixels of very high light intensity “bleed” into neighboring pixels).

Another development that is more related to the general performance increase of graphics hardware than to programmability is that non-local effects are increasingly taken into account in real-time rendering. Reflections and refractions are just two examples where additional renderings of the scene, combined with reasonable assumptions (usually that the object is small with respect to the distance to the reflected/refracted part of the scene) can be used to simulate a global effect.

Shadows. By far the most studied global effect in real-time rendering is the appearance of shadows, which occur whenever there is an object in the light path between a point to be illuminated and a light source. We distinguish between hard shadows, which are due to an ideal point (or directional) light source, and soft shadows, which are due to real area (or volumetric) light sources. Hard shadows show a discontinuous shadow boundary, because the decision whether a point is in shadow or not is boolean. Soft shadows, on the other hand, show a gradual transition from the umbra region (fully shadowed) to the fully illuminated region. The area of this transition is called penumbra, and includes all points from which the light source can only be seen partially. Soft shadows are significantly more complex to compute than hard shadows, and interactive soft shadow algorithms have only been presented very recently [Guen06].

Hard shadows, on the other hand, are relatively well explored, although they still present important challenges to be solved. There are two main algorithms to compute hard shadows in dynamic scenes: *shadow volumes* [Crow77] and *shadow maps* [Will78]. Shadow volumes are calculated in object space and therefore produce pixel-exact shadow boundaries. However, they place relatively high demands on both the main processor and on the graphics card, and are also more difficult to implement. In contrast, shadow maps are very simple to implement and often have negligible overhead in rendering time, since the cost is only one additional rendering pass without shading. However, shadow maps operate in image space and are therefore prone to aliasing artifacts, evident as staircase effects at the shadow boundaries. The recent years have seen an influx in approaches to reduce these artifacts, among them the algorithm presented in Chapter 3 in this thesis, and other approaches also proposed by the author and coworkers [Wimm06b, Gieg06, Gieg07a, Gieg07b, Sche07].

Towards global illumination in real time. There is increasing interest in carrying out a full global illumination simulation in real time. The main characteristic of global illumination is that each surface point that is illuminated by a light source in turn becomes a light source itself and contributes to the illumination of the scene. Obviously this leads to an unmanageable amount of light interactions. While diffuse illumination can be precomputed using arbitrarily complex algorithms for static scenes, e.g., using Radiosity [Sill94], dynamic scenes and non-diffuse illumination require more sophisticated solutions. A popular technique is to precompute the radiance transfer of an object [Sloa02], which allows interactive changes to the lighting environment, while all direct and indirect illumination effects within the object are taken into account. This thesis does not deal with real-time global illumination, and will therefore not explore this topic further.

1.2.4 Point-Based Rendering

The idea of point-based rendering is that highly complex objects can be better represented using points instead of polygons, since at that level of detail the exact topology of the object surface becomes irrelevant [Pfis00, Rusi00]. These ideas became popular before the wide-spread availability of actual point-sampled geometry, and therefore research concentrated on higher-quality rendering [Bots05] based on the assumption that each point actually represents a small disk, with a normal vector and a disk radius stored with it.

Meanwhile, 3D range scanners are starting to become affordable at least for research labs. It turns out that especially for datasets acquired using long-range scanners (i.e., with a range from 2 to several 100 meters), it is not obvious how to generate a robust surface approximation from the dataset, since the density of points varies greatly depending on the distance to the scan position, and natural objects often have features that fall below the sampling density, making surface reconstruction impossible. On the other hand, the sheer amount of data to be processed presents an interesting new challenge for real-time rendering research. Before thinking about rendering a point cloud that takes 4 Gigabyte on disk using an accurate surface approximation, it is important to solve the problem of how to display the model at all. Chapter 6 introduces a hierarchical data structure that is amenable to out-of-core rendering of such huge point clouds. Such an algorithm is important to provide quick scan visualizations, and to allow manipulation of points and management of the dataset without having to rely on an accurate surface reconstruction.

It is to be expected that point-based rendering methods will gain more and more importance, since there is a huge number of applications where real objects need to be digitized and then visualized or even modified. Interactive modification of such huge datasets is a research topic the author is currently investigating.

1.2.5 Conclusion

This section has provided a very abridged overview of the research area of real-time rendering. Not all topics relevant in real-time rendering were touched, instead the focus was put on those areas which are relevant to understand the papers in this thesis in a wider context.

The papers in this thesis are related in the sense that they highlight different important aspects of real-time rendering. Chapter 2 lays the groundwork by providing general methods to approach the ideal goal of real-time rendering. Chapter 3 shows how to improve the quality of real-time rendering by providing better-looking shadow rendering, one of the most popular research topics in real-time rendering. Chapters 4 and 5, on the other hand, deal with increasing the performance through two methods for visibility culling, one for runtime computation and one for preprocessing. Finally, Chapter 6 extends real-time rendering from traditional polygon rendering to a new type of dataset that has recently gained importance, namely point clouds. In that sense, it is also a method for improving performance, but also highlights the additional system aspect of out-of-core rendering, which appears especially with the huge datasets that are common in point-based rendering.

1.3 List of Selected Papers

This thesis contains the following papers [Wimm03, Wimm04, Wimm05, Wonk06, Wimm06a]:

1. Michael Wimmer and Peter Wonka:
   Rendering Time Estimation for Real-Time Rendering
   In Per Christensen and Daniel Cohen-Or, editors, *Rendering Techniques 2003 (Proceedings Eurographics Symposium on Rendering)*, pages 118–129. Eurographics, Eurographics Association, June 2003. ISBN 3-905673-03-7.

2. Michael Wimmer, Daniel Scherzer, and Werner Purgathofer:
   Light Space Perspective Shadow Maps
   In Alexander Keller and Henrik W. Jensen, editors, *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, pages 143–151. Eurographics, Eurographics Association, June 2004. ISBN 3-905673-12-6.

3. Michael Wimmer and Jiří Bittner:
   Hardware Occlusion Queries Made Useful
   In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, March 2005. ISBN 0-32133-559-7.

4. Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov:
   Guided Visibility Sampling
   ACM Transactions on Graphics, 25(3):494–502, July 2006. Proceedings ACM SIGGRAPH 2006, ISSN 0730-0301.

5. Michael Wimmer and Claus Scheiblauer:
   Instant Points
   In *Proceedings Symposium on Point-Based Graphics 2006*, pages 129–136. Eurographics, Eurographics Association, July 2006. ISBN 3-90567-332-0.

The papers included in this thesis appear unmodified in their original, published form, except for the typesetting, which has been adapted to conform to the style of this thesis. No modifications to the text have been done. The bibliography sections have been joined into a single bibliography at the end of this thesis.

1.4 Overview of the Selected Papers and Contributions of the Author

This section gives a short overview on each of the papers in this thesis. These papers represent a sample of the research work the author carried out in the years 2001–2007. For this time, the author also lists more than 20 additional refereed scientific articles, papers and book chapters in his publication list. Some of them are listed with the appropriate chapters as related work, but a number of topics in which the author was heavily involved, such as image-based rendering, level-of-detail rendering, real-time rendering of natural phenomena, and urban modeling, cannot be shown due to the required scope of such a thesis.

In general, computer graphics research is a collaborative effort, where cooperations with diploma- and PhD students and other PostDocs are necessary to spread the effort usually required to implement complex real-time rendering systems. Therefore none of the papers in this thesis is a single-author paper by Michael Wimmer alone—actually, such papers are the exception, not the rule, in the field of computer graphics. However, as traditionally evidenced by the first-author position on all but one of them, the author made a significant contribution to these papers, typically by developing the initial idea, guiding the implementation, and writing the article. The following sections contain more details about the contributions of the author to each of these papers.

1.4.1 Chapter 2: Rendering Time Estimation for Real-Time Rendering

This chapter sets a baseline for the subsequent work in real-time rendering. It deals with the all-important question what the term “real-time” actually means in the context of computer graphics. As a result, it offers several methods to estimate (i.e., predict) the rendering time of a set of graphics primitives, which is crucial when designing rendering systems with bounded update rates. The paper also contains suggestions for several extensions to graphics hardware to facilitate the design of more predictable real-time graphics systems, most of which have meanwhile already been adopted in current graphics hardware or drivers.

The author has developed the original idea and refined it in discussions with Peter Wonka. He has implemented the rendering system and produced results. He has also written the complete text of the article, while most figures were provided by Peter Wonka. The article was published at the EUROGRAPHICS Symposium on Rendering, which is, after ACM SIGGRAPH, the second most important publication venue for rendering research in computer graphics (including real-time rendering).

1.4.2 Chapter 3: Light Space Perspective Shadow Maps

This chapter introduces a new mathematical technique to significantly improve the apparent quality of shadows in real-time rendering. The algorithm exploits theoretical findings about perspective transformations, and provides quality improvement at no cost versus the reference algorithm.

The implementation for this work was carried out by diploma student Daniel Scherzer under the direction and guidance of the author. Michael Wimmer initiated and guided the project, created the theoretical framework and wrote the complete paper. Daniel Scherzer provided the implementation and results. This paper has also been published at the EUROGRAPHICS Symposium on Rendering.

1.4.3 Chapter 4: Hardware Occlusion Queries Made Useful

This chapter deals with one of the most prominent topics in real-time rendering, *visibility culling*. The presented technique exploits a graphics hardware feature, the so-called occlusion queries, to predict the visibility of large parts of the scene on the fly. This allows much faster rendering since invisible scene parts do not need to be processed by the graphics hardware. Due to its simplicity and efficiency, this algorithm has become the de-facto standard for rendering large scenes with occlusion culling.

This chapter has been completely written by the author, after helpful discussions with Jiří Bittner, who also had an important contribution to the original Coherent Hierarchical Culling technique [Bitt04], on which this chapter is based.

In contrast to the other included papers, this is a chapter from an actual published book. The *GPU Gems* book series describes computer graphics research results and also techniques that fall under the category of “graphics tools,” with the requirement that they be applicable to real-time rendering with graphics hardware. This book series has gained a lot of attention also in the academics community, and has become an important publication venue that reaches both academic researchers as well as practitioners. Note that in order to appeal to practitioners, the style of presentation is much more colloquial than a journal article, however this should not distract from the scientific value of the technique.

1.4.4 Chapter 5: Guided Visibility Sampling

In contrast to on-the-fly visibility culling as described in the previous section, Guided Visibility Sampling exploits various sampling techniques to precompute an accurate visibility solution offline. This saves runtime processing time and allows advanced real-time algorithms like prefetching and network transmission, but only works for static scene parts. The algorithm is extremely robust and general since it is based on sampling, but it is also very accurate because it uses sampling strategies that adapt to the geometry present in the scene.

Initial ideas for this algorithm were discussed by Peter Wonka and Stefan Maierhofer, who also implemented together with Gerd Hesina a first prototype system, which was later improved by Kaichi Zhou. The author contributed significantly to refining the algorithm, and together with Peter Wonka reimplemented the whole system based on the new ideas in a more efficient manner. The author also wrote the complete paper and created all results, while figures were done by Peter Wonka. Alexander Reshetov provided the integration with Intel's fast MLRTA ray tracer [Resh05]. This paper was published at the ACM SIGGRAPH conference, which is the most important publication venue in computer graphics.

1.4.5 Chapter 6: Instant Points

The wide availability of 3D range scanning devices has recently opened a new research field inside real-time rendering: the efficient display of huge point clouds. This chapter describes a system capable of rendering many Gigabytes of point data at interactive rates through an advanced hierarchical out-of-core rendering algorithm.

This project was initiated and supervised by the author, while the implementation was carried out by diploma student Claus Scheiblauer under the guidance of the author. Michael Wimmer also developed the theoretical framework and wrote the complete paper. Claus Scheiblauer provided the implementation and results. This paper was published at the Symposium of Point-Based Graphics, which is a relatively recent venue, but has quickly become the main venue for presenting point-based rendering and modeling research.

Chapter 2

Rendering Time Estimation for Real-Time Rendering

Published as:

- Michael Wimmer and Peter Wonka:
  Rendering Time Estimation for Real-Time Rendering
  In Per Christensen and Daniel Cohen-Or, editors, *Rendering Techniques 2003 (Proceedings Eurographics Symposium on Rendering)*, pages 118–129. Eurographics, Eurographics Association, June 2003. ISBN 3-905673-03-7.

Rendering Time Estimation for Real-Time Rendering

Michael Wimmer and Peter Wonka

Abstract

This paper addresses the problem of estimating the rendering time for a real-time simulation. We study different factors that contribute to the rendering time in order to develop a framework for rendering time estimation. Given a viewpoint (or view cell) and a list of potentially visible objects, we propose several algorithms that can give reasonable upper limits for the rendering time on consumer hardware. This paper also discusses several implementation issues and design choices that are necessary to make the rendering time predictable. Finally, we lay out two extensions to current rendering hardware which would allow implementing a system with constant frame rates.

2.1 Introduction

The quality of a real-time rendering application is determined by several aspects, including impressive models, visually attractive effects like shadows, reflections and shading, etc. In this paper we take a look at another component that largely contributes to high-quality graphics simulation as sought for in computer games, trade shows, and driving simulators. It is a factor that has often been overlooked in current products: fluent and continuous motion. For the impression of fluent motion, it is necessary to render at a fixed high frame-rate, mainly determined by the refresh rate of the display device (typically 60 Hz or more). Failing to do so results in distracting and visually displeasing artifacts like ghosting (Figure 2.7) and jerks [Helm94]. Furthermore, the predictability of frame times is crucial for the ability to schedule certain events and to coordinate input with simulation and display. Especially due to the use of hardware command buffers, the time span between the issuing of a rendering command and its actual execution can easily be over one frame. If the rendering time of the frame is not known in advance and frame times have a high variance, the apparent moving speed will change constantly (see Figure 2.6). This is most visible when the viewer is rotating.

To obtain a system with a fixed frame rate, it is necessary to guarantee that the rendering time does not exceed a certain time limit. One building block of such a system is a prediction of the rendering time for a given frame. While previous work [Funk93, Alia99] showed how to build a real-time rendering system with guaranteed frame rates when such a prediction function is given, the actual prediction function used for the rendering time estimation should be studied in greater detail.

In this paper, we undertake a more in-depth study of rendering time in order to show which simple heuristics can be used for this purpose and how successful they are. Among others, we present an approach based on sampling rendering times, and an improved mathematical heuristics based on a cost function. An important part of this work is the proposal of two hardware extensions—a time-stamping function and a conditional branch mechanism—which make rendering time estimations more robust. We show how to implement a soft real-time system providing fixed frame rates using these extensions. We also skirt practical issues and design choices encountered when implementing a real-time rendering system, and give hints and examples for those. In the long run, we aim to use the results of this paper to construct a soft real-time system with bounded frame times. Rendering time estimation can also be used to calculate efficient placement of impostors in complex scenes.

First, we set out by defining the rendering time function. As the most general form we propose

$$t = RT(SG, RA, HW, ST),$$

where SG is a scene graph, RA is the rendering action used for traversal, HW is the hardware, and ST is the current state of the hardware, software and the operating system. While this form is general enough to incorporate all important effects that influence the rendering time, it is complicated to use in practice. Therefore, we will use a simpler form, where the scene graph is an ordered set of objects $\mathbf{X} = (x_1, ..., x_n)$, with given geometry and attributes $x_i = (g_i, a_i)$. Furthermore, we assume that the rendering action is implicitly defined by the attributes of the objects. We thus obtain the following form:

$$t = RT(\mathbf{X}, HW, ST)$$

This formulation of the rendering time function is the basis for the framework which we will use to discuss different aspects of the rendering time estimation problem.

The rest of the paper is organized as follows. Section 2.3 explains the functionality of current rendering hardware, section 2.4 describes our framework to estimate rendering time and explains the different tasks and factors that contribute to the rendering time and how to estimate them. In section 2.5 we describe the crucial parts of the rendering time estimation in greater detail, and in section 2.6 we propose two hardware extensions necessary for a soft real-time system. Sections 2.2, 2.7 and 2.8 present previous work, results and conclusions.

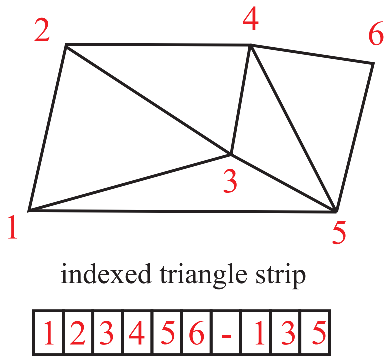


Figure 2.1: This figure illustrates different attributes of an indexed triangle strip. The strip consists of 6 indices, then a restart denoted by **-** and three more vertices. The number of polygons is 5, the number of indices is 9, the number of vertices is three times the number of polygons (15) and the number of actually transformed vertices is 6, i.e., each vertex is only transformed once, as they all fit into the vertex cache.

2.2 Previous Work

Funkhouser and Séquin [Funk93] demonstrate a real-time rendering system that bounds frame times through appropriate level-of-detail (LOD) selection. They assign a cost and a benefit to each LOD for each object. The optimization tries to select for each object such a LOD that the sum of the benefit values is maximized, given a maximum allowable cost. The cost metric used is equivalent to the following rendering time estimation:

$$RT(x) = max(c_1 * \#polys(x) + c_2 * \#v(x), c_3 * \#pix(x))$$

where x is the LOD in consideration, $\#polys$ is the number of polygons and $\#v$ is the number of vertices of the object, and $\#pix$ is the number of pixels in the projection. The parameters c_1, c_2 and c_3 are constants that are determined experimentally by rendering several sample objects.

While vertex and polygon counts are both reasonable estimates for geometric complexity, a better match for the way modern GPUs actually behave is the number of *actually transformed vertices*, i.e., the number of vertices that are not taken from the post-transform vertex cache used by such GPUs, but actually fetched and transformed. Hoppe uses this number as a basis for a cost function used for the creation of efficient triangle strips [Hopp99]. While Hoppe additionally considers the index transfer for indexed triangle strips, due to the small size of indices this factor plays only a limited role for rendering time estimation. See Figure 2.1 for an illustration of the different concepts.

Aliaga and Lastra [Alia99] construct a view-cell based real-time rendering system. They sample the view space with a large number of viewpoints, where each sample viewpoint defines a view cell around it. For each cell, they select objects for direct rendering according to a cost-benefit function. The remaining objects are replaced by layered depth images. The cost metric is based on the number of triangles and ignores the influence of rasterization. The proposed rendering time estimation is therefore

$$RT(x) = c_1 * \#tris(x)$$

where c_1 is determined by the triangle rate of the given hardware. The accuracy of these estimations will be compared with our estimations in section 2.7.

The problem of maintaining a specified frame rate has also been addressed in the Performer system [Rohl94], however based on a reactive LOD selection system. However, bounded frame rates can only be guaranteed using a predictive mechanism. Regan and Pose [Rega94] demonstrated a system capable of maintaining fixed frame rates with a special-purpose display controller for head-mounted displays based on just-in-time image composition.

2.3 Rendering Hardware Overview

This research focuses on consumer hardware, namely a PC with a state-of-the-art graphics card (current examples are NVIDIA's GeForce or ATI's Radeon products). Consumer hardware is not naturally geared towards the construction of a real-time rendering system, because hardly any tools are available to give a hard time limit for the execution of a given set of rendering commands. However, these systems are very wide spread and used by many interactive rendering applications. Therefore, it seems worthwhile to provide a best-effort estimation of the execution time, knowing that a hard real-time rendering system cannot be achieved. So we aim at the construction of a soft real-time rendering system working with statistical guarantees for the rendering time.

In the following we give a functional overview of the rendering system (see Figure 2.2). The rendering process uses the CPU and the GPU in parallel. The application is executed on the CPU and sends commands to the GPU via the graphics driver. We will use a typical frame of a simple rendering engine to illustrate this functionality (see also Figure 2.4). On the *CPU* side, the application issues a clear screen (clr) command to initialize the frame buffer. Then the application traverses all objects $x_i = (g_i, a_i)$. For each object, the application has to set the state of the graphics hardware according to the attributes a_i and then send the geometry g_i. The driver sends the commands to a *command buffer* (FIFO). The GPU reads commands from this buffer and executes them. State commands change the state of the pipeline according to the attributes of the primitives. Geometry is sent down

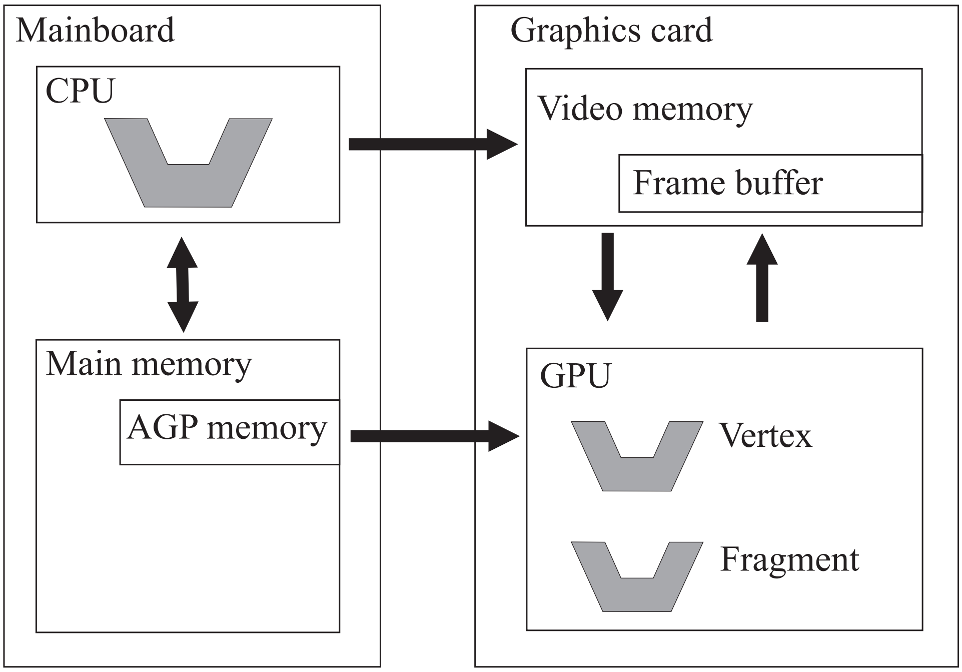


Figure 2.2: This figure shows an overview of the graphics architecture. The arrows indicate the most important flows of data.

the pipeline starting with the vertex processing unit, after which it is rasterized into fragments that in turn go to the fragment processing unit. The primitives, mainly indexed triangle strips, consist of vertex data and index data. For efficiency, both vertex and index data are either stored in AGP memory or in video (=graphics card) memory, both of which are directly accessible by the GPU.

2.4 The rendering time estimation framework

In this section we will propose a framework for analyzing the rendering time. The main idea is to assume a subdivision of the rendering process into a number of conceptually independent tasks which can be estimated separately. The remaining interdependencies that are not captured by our model can usually be subsumed under a so-called system task. Note that the test system used to obtain the empirical results shown in this section is described in section 2.7.

2.4.1 The refined rendering time estimation function

The refined rendering time estimation function RT which is used for the discussion in this section is made up of estimations ET for four major components (tasks),

- system tasks (ET_{system}),
- CPU tasks (ET^{CPU})
- idle time ($ET^{CPU}_{idle}, ET^{GPU}_{idle}$), and
- GPU tasks (ET^{GPU}),

in the following way:

$$RT = ET_{system} + max(ET^{CPU}, ET^{GPU})$$

with

$$ET^{CPU} = ET^{CPU}_{nr} + ET^{CPU}_{r} + ET^{CPU}_{mm} + ET^{CPU}_{idle}$$

and

$$ET^{GPU} = ET^{GPU}_{fs} + ET^{GPU}_{r} + ET^{GPU}_{mm} + ET^{GPU}_{idle}.$$

Here, the indices nr denote non-rendering code, fs frame setup, r rendering code, mm memory management, and $idle$ is idle time.

2.4.2 System tasks

The operating system and other applications use time for tasks like network services, indexing services, file system, memory managers etc., some of which cannot be totally eliminated even during the execution of a higher-priority process. Further, the graphics driver itself might schedule certain tasks like optimization routines, which are not documented and cannot be predicted. This time is denoted as ET_{system} in our framework, and is the main reason for variations in frame times for a single view.

To understand its influence on the rendering time, we conducted the following experiment. In a test scene we selected a number of individual viewpoints and measured the rendering time for 10,000 subsequent renderings for each viewpoint. All non-critical services were turned off (including indexing and other applications with file access), and the rendering thread was set to a higher priority, thus basically eliminating all non-critical operating system activity. Synchronization with the vertical retrace was also turned off in order to eliminate any dependency on the physical display device. We expected the rendering time variations to resemble a lognormal distribution, which would be typical for a process involving completion times (this has been observed, for example, in quality control engineering and

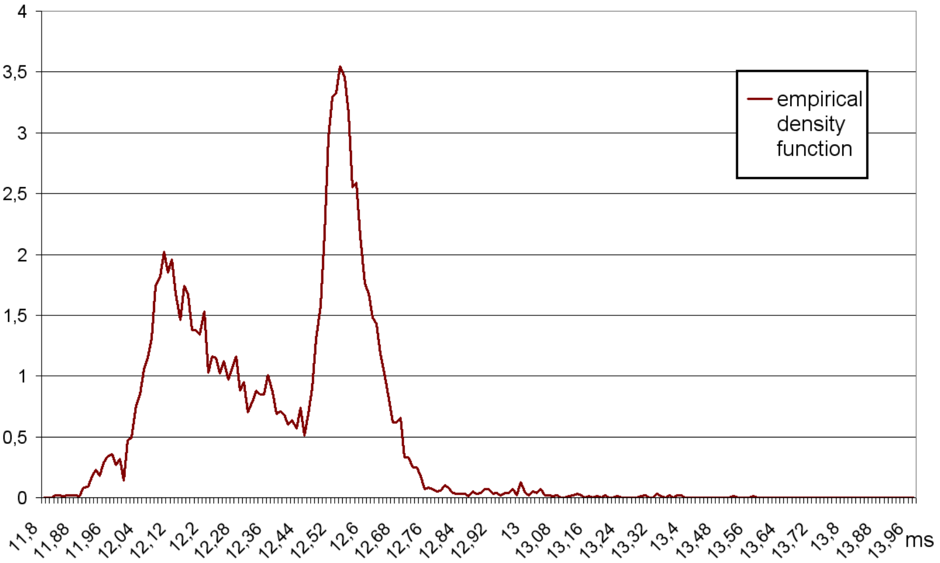


Figure 2.3: This graph shows an example for an empirical density function of the rendering time derived through 10,000 renderings from the same viewpoint.

traffic flow theory [Gart93]). However, there are variations that follow a quite peculiar pattern. Figure 2.3 shows the empirical probability density function for one of the chosen viewpoints. It does not resemble any of the well-known distribution function.

One option is to assume that the variation of this distribution represents the influence of system tasks. For example, the bimodal nature of the distribution suggests that there is a regular system task (e.g., the thread scheduler, related to the thread quantum) which is executed approximately every 2 to 3 frames and takes about 0.5 ms. The variation of the distribution also depends on the total execution time of the frame. Since we aim for a system with fixed frame times, we estimate the time of the system tasks as a constant c representing the maximum deviation of the minimum rendering time for a given confidence interval and target rendering time: $ET_{system,confidence} = c$. The constant c is determined by calculating the width of the given confidence interval with respect to the empirical probability distribution function of a test measurement. Although the distribution can vary greatly near the mean value, we found the extremal values and therefore the estimated constant to be quite consistent. For our test system, we used a confidence interval of 99%, from which we calculated c as 1.52ms.

2.4.3 CPU tasks

The CPU is responsible for several tasks. First, there is application code that does not directly contribute to rendering, like artificial intelligence, animation, collision detection, sound and networking (ET_{nr}^{CPU}). Second, there are rendering-related tasks like scene-graph traversal, view-frustum culling and dynamic state sorting (ET_{r}^{CPU}). Third, there are issues of memory management like the recalculation of dynamic vertex data and texture downloads (ET_{mm}^{CPU}). Fourth, there is the idle time when the GPU is too busy to accept further commands (ET_{idle}^{CPU}).

CPU memory management

Texture memory management. If not all textures for a given scene fit into video memory (and can therefore not be downloaded before rendering), a memory management routine is required which selects for each frame which textures need to be downloaded. For efficiency reasons [Carm00] and for predictability, this should not be left to the graphics driver. The texture management time $ET_{mm,tex}^{CPU}$ is given by the sum of download times for all textures selected for download for a given frame.

Geometry memory management. Similarly, *static geometry* needs to be managed by a geometry management routine if not all the static geometry in a scene fits into memory directly accessible by the GPU (video memory and AGP memory). The geometry management time is $ET_{mm,geom}^{CPU} = c * g$, where g is the amount of memory for the geometry scheduled for download in the current frame, and c is the memory copying speed. In the case of indexed geometry, indices are transferred either during the API call (in unextended OpenGL) or are managed together with the vertex data. *Dynamic geometry*, i.e. animated meshes where vertex positions change, should be written directly to AGP memory upon generation in order to avoid double copies. For a comparison of geometry download methods and their influence on the rendering time see also section 2.4.5.

2.4.4 Idle time

Ideally, CPU and GPU run in parallel. However, sometimes either the CPU or the GPU sits idle while waiting for the other part to supply data or complete a task. This (undesirable) time is called idle time and occurs in the *graphics driver*, which runs as a part of the user application on the CPU. The issue of idle time arises when the graphics driver writes commands for the graphics card to the command buffer and this buffer is either full or empty:

- When the buffer is full, the driver blocks the application and returns only when the buffer can be accessed again. This usually means that the CPU

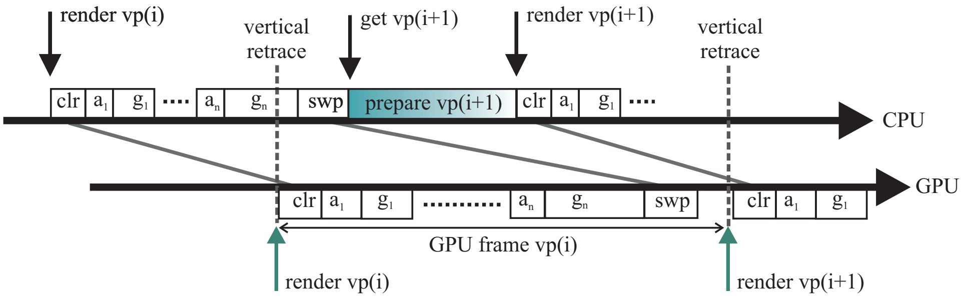


Figure 2.4: This figure shows how CPU and GPU work in parallel. When the GPU starts rendering the current frame ($vp(i)$), the CPU has already written most of the rendering commands for this frame to the command buffer. When the CPU is finished with the current frame, it can prepare the next frame ($vp(i+1)$), starting with CPU-intensive non-rendering tasks and memory management, while the GPU is still busy rendering the current frame.

is supplying graphics commands fast enough and could be used to do more complex non-rendering calculations.

- When the buffer is empty, the GPU is starved for rendering commands (this situation is therefore also known as back-end starvation). The CPU is not supplying graphics commands fast enough and the GPU sits idle, leaving some of its potential unused.

A rendering application which makes best use of the available graphics hardware should strive for maximum parallelism and avoid backend starvation (so that $ET_{idle}^{GPU} = 0$). If this cannot be achieved because the non-rendering code is causing the starvation and cannot be optimized, the unused time in the GPU can still be used to execute more complex shaders or draw more complex geometry without affecting the rendering time. In the following, we therefore assume a balanced system, where the GPU is always busy ($ET_{idle}^{GPU} = 0$) and the CPU might be occasionally idle. In our current implementation, the engine performs non-rendering code for the next frame when the GPU is still busy drawing the current frame. This is illustrated in Figure 2.4.

Note that some graphics commands require other buffers apart from the command buffer which can also become full and lead to idle time. The most important example are immediate-mode geometry commands in OpenGL, where the driver accumulates vertices in AGP or video memory buffers. These buffers are however usually quite small, so that the driver has to stall the CPU on most rendering commands and practically all parallelism between CPU and GPU is lost. Note that these commands are also undesirable because of the overhead imposed by the large number of API calls required. Geometry should therefore only be transferred under application control using extensions [NVID03b, ATI 03], which also has the advantage that static geometry need not be sent every frame (see section 2.4.3).

2.4.5 GPU tasks

The rendering tasks on the GPU typically constitute the most important factors for the rendering time. We can identify the following tasks [Prou01]:

Per-frame calculations. At the beginning of each frame, the frame buffer has to be cleared, and at the end of each frame, the back buffer has to be swapped with the front buffer. In a real-time setting, the swap should be synchronized with the vertical retrace of the screen in order to avoid tearing artifacts. While actual buffer swap times are negligible, clear times and buffer copy times (for windowed applications where buffer swapping is not possible) need to be taken into account in the estimation.

Per-primitive-group calculations. State setup including texture bind, material setup, vertex and fragment shader bind and shader parameters. The speed of this stage is determined by the number and type of state changes in a frame (with changes in vertex and fragment programs usually being the most expensive, followed by texture changes).

Per-primitive calculations are not fully developed in current graphics hardware. One example on current hardware is the adaptive refinement of triangles based on vertex normals [ATI 01].

Per-vertex calculations can be further broken down into index lookup (if an indexed rendering primitive is used) vertex fetching from video or AGP memory, and execution of the vertex shader. The time spent for these calculations is determined by the complexity of the vertex shader and the number of actually transformed vertices (see section 2.2), which can be determined by doing a FIFO-cache simulation for the geometry to be estimated. Note that one way to minimize the number of actually transformed vertices needed for a given geometry is to use a vertex-cache aware triangle stripper [NVID03a]. Note also that the vertex cache can only work for indexed primitives and when geometry is stored in AGP or video memory, therefore non-indexed primitives should only be used for geometry containing no shared vertices.

Triangle setup is the interface between per-vertex and per-fragment calculations. Triangle setup is usually not a bottleneck in current GPUs.

Per-fragment calculations or rasterization. These calculations are done by the fragment shader and subsequent stages. Examples include texture mapping, multitexturing, environment mapping, shadow mapping etc. The speed of this stage is determined by the complexity of the fragment shader, but also by the efficiency of early fragment z-tests present in newer cards, and texture memory coherence.

In the following, we discuss some other factors influencing rendering time, including bottlenecks, rendering order and the type of memory used.

	simple pixel	complex pixel	difference
simple vertex	4.969	7.35	2.381
complex vertex	9.859	11.856	1.997
difference	4.89	4.506	

Table 2.1: This table shows the rendering time for one viewpoint in the terrain scene. If the complexity of the vertex shader is increased, the rendering time increases (independent of the fragment shader complexity). If the complexity of the fragment shader is increased the rendering time increases as well (independent of the vertex shader complexity). This shows the lack of parallelism between the fragment and geometry stages.

The myth of the single bottleneck in an application

A common misunderstanding with regard to the rendering pipeline is that the speed of an application is defined by a single bottleneck. While this is true for each particular point in time, the bottleneck can change several times even during the rendering of one single object. Due to the small size of post-transform vertex caches, the fragment and the geometry stage can work in parallel only for one particular triangle size, which depends on the complexity of the vertex and the fragment shader. Triangles larger than this “optimal triangle” usually stall the pipeline, whereas smaller triangles will cause the fragment stage to sit idle.

While for some far-away objects the rendering time is determined only by the geometry stage, most objects consist of several triangles larger than the optimal triangle and several ones that are smaller. An example is shown in Table 2.1, where neither vertex nor fragment shader can be made more complex for free (as should be the case for a single bottleneck). This effect needs to be incorporated in rendering time estimation heuristics. Our results indicate that a sum of fragment- and geometry terms might be better suited to estimate rendering time than taking the maximum of such terms [Funk93]. A new heuristic based on this observation is introduced in section 2.5.3.

Rendering order

The rendering order can influence rendering time in two ways: First by the number of state changes required, which suggests that objects should be sorted by their rendering modes. Second by the effect of pixel occlusion culling, which suggests that geometry should be rendered front to back in order to reduce the amount of fragments that actually need to be shaded.

Table 2.2 illustrates that mode sorting improves rendering time especially in CPU-limited cases because state changes are CPU intensive. The other test scenes do not profit from mode sorting because they either contain no state changes (terrain scene), or for other reasons (forest scene, no improvement although 194 state

Sort order:	no sort	matrix/ texture	texture/ matrix	presorted text./mat.
texture changes	984	449	211	211
material changes	639	525	110	110
alpha changes	186	28	12	12
cpu limited:				
GPU time in ms	9.188	7.725	8.539	7.07
geometry limited:				
GPU time in ms	14.634	14.328	14.138	14.138

Table 2.2: Examples for mode sorting in the city test scene in a CPU-limited and a geometry-limited setting (with a more complex vertex shader). The column headers indicate the sort order used (e.g., in the second column, first by transformation matrix, then by texture). In the presorted case, there is no CPU overhead.

	normal	f2b	b2f
GPU time in ms	10.7	9.143	12.552

Table 2.3: Distance sorting (front-to-back and back-to-front) in the city scene in a fill-limited setting.

changes are reduced to 2), in which case dynamic mode sorting even increases rendering time.

Table 2.3 shows the effect of distance sorting in a strongly fill-limited setting, as compared to normal rendering (no sorting).

GPU memory management

There are several possible choices in which to send geometry to the graphics hardware, including whether to use indexed or non-indexed primitives, which type of memory to use (AGP or video), which memory layout to use (interleaved/non-interleaved vertex formats) and which routine to use for the memory copy. The graphics hardware and the driver are usually poorly documented and the optimal choice can only be found by intensive testing.

The fastest type of memory is generally video memory, but it only comes in limited quantities, and streaming geometry directly into video memory during rendering might reduce bandwidth for texture accesses. Table 2.4 shows that rendering from and copying to video memory can be done in parallel without the two influencing each other significantly. This means that geometry downloads to video memory don’t increase the overall rendering time if there is enough CPU time left. However, when rendering is already CPU limited (Test4), the total rendering time increases by the full time needed for the copy. Therefore, dynamic geometry transfers should be avoided in this case and as much geometry as possible stored statically in video memory. Test2 shows that in the case of simple vertex shaders,

	rendering		copying	
	[ms]	[tv/sec]	[ms]	[MB/sec]
Test1: terrain video memory				
independent	4.36	19.73	1.7	793
parallel	4.59	18.75	1.9	733
Test2: terrain AGP memory				
independent	4.95	17.38	1.4	952
parallel	6.27	13.72	3.3	420
Test3: terrain AGP memory, interleaved				
interleaved AGP				
independent	4.41	19.52	1.4	952
parallel	5.04	17.06	2.9	481
Test4: city video memory CPU limited				
independent	9.28	11.878	4.1	793
parallel	14.167	7.78	5.1	644

Table 2.4: The table shows rendering and memory copy times in two cases: independent shows the times for rendering and copying alone; parallel shows the slowdown when rendering and copying are done concurrently and compete for the same memory (tv/sec are the actually transformed vertices per second).

using AGP memory can slow down the GPU due to the limited bandwidth of the AGP bus, especially when geometry is also copied to AGP memory in parallel. This has to be considered when working with animated meshes. An interesting observation (Test3) is that the geometry layout can influence the rendering speed so that terrain rendering with interleaved geometry from AGP memory is almost as fast as rendering from video memory. Note also that these results are close to the maximum transformation capability of the used rendering hardware.

2.5 Methods for rendering time estimation

In this section, we give several heuristics which can be used to calculate the rendering time estimation function. These heuristics are then compared and evaluated in section 2.7.

To obtain an estimation for the rendering time, we have to choose a method in a spectrum that is spanned by the extremes of *measuring* and *calculating*. We propose three basic methods: one sampling method that is mainly defined by measurements (and gives RT directly), another hybrid method that is a tradeoff between sampling and heuristic calculations and a third method that uses a heuristic function based on the number of the actually transformed vertices and rendered pixels (the latter two methods estimate ET_r^{GPU}, i.e., the other terms of RT have to be estimated separately as described in section 2.4).

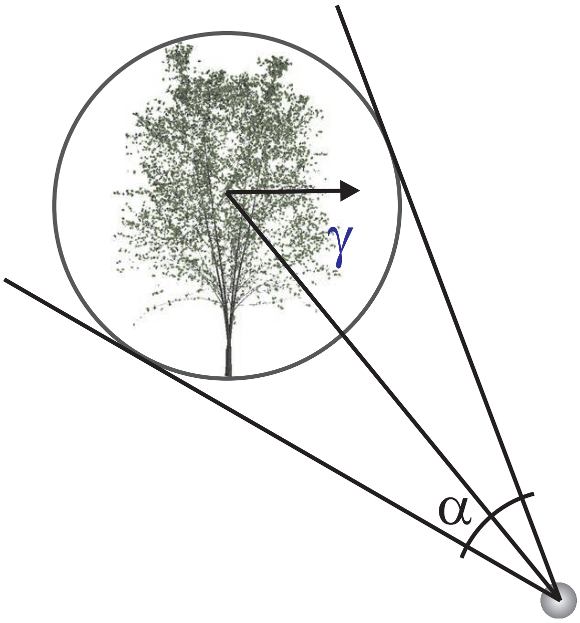


Figure 2.5: This figure shows precalculated per-object sampling. The rendering time estimation is parameterized with three angles, two of those are shown in this 2D figure.

2.5.1 View-cell sampling

The proposed sampling method works for a view-cell based system, where a potentially visible set (PVS) is stored for each view cell. For each view cell we discretize the set of view directions, randomly generate n views around each discretized direction and measure the rendering time for each view. The maximum rendering time of the n sample views is used as an estimation for the total rendering time RT of the direction and the view cell under consideration.

2.5.2 Per-object sampling

The hybrid method estimates the rendering time of a set of objects by adding the rendering time estimations of the individual objects. The assumption is that when two sets of objects are rendered in combination ($\mathbf{X}_1 \oplus \mathbf{X}_2$), the rendering time is at most linear with respect to the rendering times of the original sets $\mathbf{X}_1$ and $\mathbf{X}_2$.

$$ET_r^{GPU}(\mathbf{X}_1 \oplus \mathbf{X}_2) \leq ET_r^{GPU}(\mathbf{X}_1) + ET_r^{GPU}(\mathbf{X}_2)$$

To estimate the rendering time of a single object, we parameterize the rendering time estimation function by three angles $ET_r^{GPU}(x) = ET_r^{GPU}(x, \alpha, \gamma, \phi)$ (see Figure 2.5 for a 2D view). The angle α is the angle between the two supporting lines on the bounding sphere. This angle (which is related to the solid angle) is an estimate for the size of the screen projection. The angles γ and ϕ (for elevation) describe from which direction the object is viewed. In a preprocess, we sample this function using a regular sampling scheme and store the values in a lookup table together with the object. As the angle α becomes smaller, the rendering time will be geometry limited and not depend on the viewing parameters any more. We use this observation to prune unnecessary test measurements. This rendering time estimation can be used in two ways:

Per-viewpoint estimation: For an online estimation, the rendering time is looked up in the table stored with the object with respect to the current viewpoint and view direction. Care has to be taken not to make the estimation overly conservative: For objects straddling the view frustum, the angle alpha has to be clipped to the view frustum. This is especially important for nearby objects since they tend to cover a larger screen area, i.e., they usually contain several large rasterization-limited triangles which add significantly to the estimated time.

Per-view-cell estimation: For a view cell, the rendering time estimation is more involved. As in view-cell sampling (section 2.5.1), we discretize the set of possible viewing directions from a view cell. For each discretized direction, a conservative estimate of the rendering time is calculated separately in the following way: We seek for each object the point on the view-cell boundary where the object bounding sphere appears largest in the viewing frustum. This point lies either on a boundary vertex of the view cell, or on a boundary face such that the view frustum given by the viewing direction is tangent on the object bounding sphere. The rendering time estimation associated with this point is then looked up as in the per-viewpoint estimation, and added to the estimated rendering time for this viewing direction. Finally, the rendering time estimation for the whole view cell is calculated as the maximum of the rendering time estimations from the discretized viewing directions.

2.5.3 Mathematical heuristics

As the last model, we compare several mathematical heuristics for the rendering time estimation ET_r^{GPU}. Note that in previous work, the first 3 heuristics shown here were used alone, without regard for the other components of RT described in this paper. We propose to the presented heuristics within the complete rendering time estimation framework, which allows taking into account effects like texture and geometry management etc.

The first heuristic **H1** is the *triangle count* [Alia99]. The assumption on which this heuristic is based is that the ratio of actually transformed vertices to triangles is uniform over the whole scene, and that the rendering time is determined by the geometry stage.

$$ET_r^{GPU}(x) = c * \#tris$$

where $\#tris$ is the triangle count of an object, and c is the triangle rate for a given hardware.

The second heuristic **H2** is the actually transformed vertex count [Hopp99].

$$ET_r^{GPU}(x) = c * tv$$

where tv is the number of actually transformed vertices and c is the vertex rate for a given hardware. This heuristic reflects the geometry processing stage more accurately than **H1**, but still neglects the influence of rasterization on rendering time.

A more complete heuristic (**H3**) is Funkhouser's cost function [Funk93] with

$$ET_r^{GPU}(x) = max(c_1 * \#polys(x) + c_2 * \#v(x), c_3 * \#pix(x))$$

where x is the object under consideration, $\#polys$ is the number of polygons of the object, $\#v$ is the number of vertices of the object and $\#pix$ is the number of pixels in the projection.

As was discussed in section 2.4.5, the bottleneck in a rendering pipeline can shift several times even when rendering a single object. The *maximum* of geometry and rasterization terms as used in **H3** is actually a lower bound for the actual rendering time, whereas the *sum* of the two terms constitutes the upper bound, and is therefore a more conservative estimation. The experiments in section 2.4.5 also suggest that in practice, the actual rendering time often tends towards the sum. Furthermore, the factor which determines geometry transformation time is the number of actually transformed vertices and not just the number of vertices or polygons. Based on these two observations, we propose a new rendering time estimation heuristic **H4** which improves upon the previous ones:

$$ET_r^{GPU}(x) = c_1 * \#tv(x) + c_2 * \#pix(x)$$

The heuristics presented here will be compared in section 2.7. However, none of the rendering time estimation functions shown in this section is sufficient to build a soft real-time system, either because timing results are not accurate enough with current timing methods (for the sampling approaches) or because estimated rendering times can sometimes be exceeded in practice even if the prediction is very accurate (for the mathematical heuristics). In the next section, we will propose hardware extensions to overcome these problems.

2.6 Hardware extensions for a soft real-time system

In this section, we introduce two hardware extensions that make it possible to implement a soft real-time system. The extensions deal with two problems encountered when using rendering time estimation: the timing accuracy problem and the estimation accuracy problem.

2.6.1 The timing accuracy problem

All the heuristics presented in section 2.5—especially the per-object sampling method—rely to some extent on the ability to measure the rendering time for specific objects. While it is relatively easy to measure the time taken by a specific CPU task, it is very difficult to obtain such measurements for GPU-related tasks. CPUs provide an accurate time-stamping mechanism via an instruction that returns the current CPU clock cycle counter. Inserted before and after a number of instructions, the difference of the counters can be used to calculate the time required for executing the instructions.

The GPU, however, is a separate processing unit, and any timing mechanism implemented on the CPU will either give only information about the interaction with the command buffer, or include significant overhead if the GPU is explicitly synchronized before each timing instruction (e.g., via the OpenGL command `Finish`).

2.6.2 The time-stamping extension

Due to the large uncertainties that such inaccurate timing can introduce in the rendering time estimation, it seems useful to implement timing directly on the GPU and to extend current hardware with a *time-stamping function*, which can be used for acquiring the accurate measurements needed to set up the rendering time estimation functions in section 2.5 (either for sampling or for calibrating one of the heuristic formulae).

This function should operate similarly to the OpenGL occlusion-culling extension: A time-stamp token is inserted into the command buffer, and when this token is processed by the GPU, a time stamp representing the current GPU time (e.g., a GPU clock cycle counter) is stored in some memory location and can be requested by an asynchronous command. Due to the long pipelines present in current GPUs, there are two possible locations for setting the time stamp: (1) at the beginning of the pipeline (when the token is retrieved from the command buffer), and (2) at the end of the pipeline, which is the more accurate solution. The difference between two such time stamps can then be used to calculate the time required for the GPU to render an object much in the same way as for a CPU to execute some instructions. The communication paths between the backend of the GPU and the CPU necessary

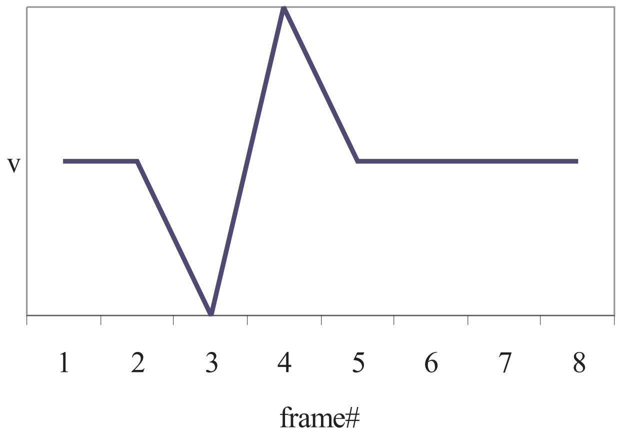


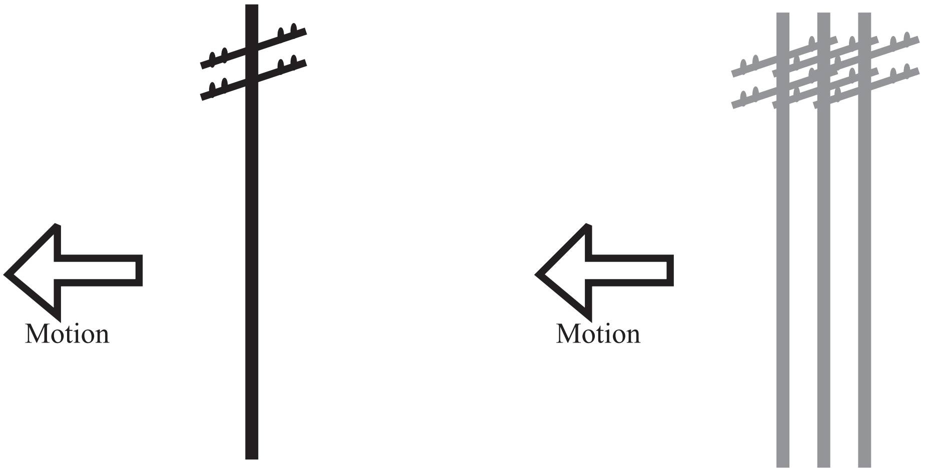
Figure 2.6: This diagram shows the strong variations in perceived movement speed during a frame skip (frame 2 is repeated).

for such an extension are already in place (they are used to transmit pixel counters used in occlusion queries).

2.6.3 The estimation accuracy problem

Funkhouser [Funk93] notes that for the metric **H3**, the actual rendering time does not deviate more than 10% from the predicted rendering time in 95% of frames. However, this means that in a real-time setting assuming a frame rate of 60 Hz, there would be up to 3 skipped frames per second. Figure 2.6 shows the strong variations in perceived movement speed caused by a single frame skip. In the accompanying video, we show that even one skipped frame every several seconds is unacceptable if a moderately smooth walkthrough is desired. The video also shows that switching to a lower frame rate (such as 30 Hz) for a longer duration is not acceptable either, due to ghosting artifacts which appear when the frame buffer is not updated at each screen refresh (see Figure 2.7).

For a quality real-time application we aim for an estimation accuracy higher than 99.9%, or a maximum of 1-2 frame skips per minute. Even the metric **H4** or the per-object sampling method using the time-stamping extension (both introduced in this paper) will not be able to provide such an accurate estimation. We therefore propose a hardware extension that can "fix" estimation errors during the rendering process.



Refresh Rate = Update Rate

Refresh Rate = 3 * Update Rate

Figure 2.7: Ghosting artifact when the frame rate doesn't match the screen refresh rate (recreated after an image from [Helm94]). Such artifacts are especially visible on sharp edges in the images, and when the viewer is rotating.

2.6.4 The conditional branching extension

We propose using a *conditional branch* in graphics hardware to switch to a coarser LOD if the remaining frame time is not sufficient. Such a conditional branching extension consists of a start token containing a sequence of numbers $t_{r_1}, \ldots, t_{r_n}$, and a branch token. When the GPU encounters the start token, it compares each t_{r_i} with the time remaining until the next vertical retrace. If t_{r_j} is the first number that is smaller than this time, all commands in the command buffer up to the j-th encountered branch token are skipped. Then all commands until the next branch token are executed, and finally all commands until the n-th branch token are skipped again.

The values t_{r_i} should be set to the result of the rendering time estimation function for all remaining objects for this frame, including the current object at a specific level of detail i. Since any tasks done by the driver on the CPU side will have to be executed for all conditional branches, such branches should only contain rendering commands which refer to geometry already stored in AGP or video memory, which can be accessed by the GPU directly.

2.6.5 Soft real-time system

Using the two proposed extensions, a soft real-time system can be implemented. The time-stamping extension guarantees accurate timings for the rendering time estimation function. Based on this function, appropriate LODs are selected for each object in each frame in such a way that the total frame time is not exceeded (there are several ways how to do that, but the LOD selection process is not topic of this paper). To guarantee that no frame is skipped even if the rendering time estimation fails, some objects (starting with those that (1) are already at a certain distance from the viewer, to reduce popping, and (2) still have some geometric complexity so that their rendering time is not negligible) are accompanied with a small number of coarser LODs, which are automatically selected by the graphics hardware if the remaining time is not enough to render the predetermined LOD. Note, however, that such a system is still a soft real-time system because frame skips can still occasionally occur (e.g., due to unforeseeable stalls caused by the operating system), but they will be reduced to a negligible number.

In order to guarantee that no geometry has to be transferred over the bus for objects that are not actually rendered, all LODs for all objects could be stored in GPU-accessible memory at the beginning of the walkthrough, if there is sufficient memory. In general, the geometric data for the LODs will be managed along with other geometric data as explained in section 2.4.3. We do not expect the additional data required by the LODs to be a significant burden on the bandwidth between CPU and GPU, since not all LODs for each object, but only a small number of additional LODs will be used. In an analogy to mip-mapping, the memory required for the lower LODs should not exceed the main model—this holds true for both discrete LODs (where successive levels should differ by a significant number of triangles) and progressive LODs (where lower levels are part of the higher levels).

2.7 Implementation and results

2.7.1 Test setup and models

The empirical values for all tables in this paper were obtained on an Athlon XP1900+ with an NVIDIA GeForce3 graphics card. The graphics API was OpenGL [Woo99] and the operating system Windows 2000. All timings were taken by synchronizing the pipeline (using the `glFinish` instruction) and reading the processor clock cycle counter both before and after rendering the object to be timed. The object was actually rendered several times between the two readings in order to minimize the influence of the synchronization overhead on the timing.

We will shortly describe the models that we used for the measurements in the paper (see also Table 2.5). The first scene is a model of a city (see Figure 2.8), our main benchmark. A PVS is calculated for each cell in a regular grid of 300x300 $10m^2$ view cells [Wonk00]. The second model is a textured terrain from another city (see Figure 2.9). The third model is a forest scene (see Figure 2.10) consisting of 157 trees with varying complexity. Each object in each scene consists of one or several lists of indexed triangle strips, where each strip can have a different texture. Visibility and view-frustum culling works on the object level. The ratio of actually

value	city	terrain	forest
#objects	5609	1	157
#triangles	3,453,465	86,400	579,644
#tv	6,419,303	86,000	1,028,978
#tv / tri	1.86	0.99	1.78
#textures	88	1	2
#state switches (sw)	14267	1	314

Table 2.5: Some descriptive values for the test scenes. #objects is the number of objects in the scene that are handled by the occlusion-culling or view-frustum culling algorithms. #tv is the number of actually transformed vertices.

	h1	h2	h3	**h4**	**pos**
city	3.27	0.7	1.12	**0.26**	1.41
forest	30.35	25.5	20.8	20.5	**5.41**

Table 2.6: Average squared errors for per-viewpoint estimations (ms^2). The newly introduced heuristics h4 and pos (per-object sampling) show improvements over the previous ones.

transformed vertices per triangle shows how well the scene is adapted to exploiting the vertex cache. State switches and triangles per state switch indicate the traversal overhead on the CPU and the GPU.

2.7.2 Comparing the per-viewpoint estimation methods

To compare the quality of a prediction function, we conducted the following test. We recorded a path through the test scenes and compared the prediction function for each viewpoint with the actual frame time. We compare the time for all four mathematical heuristics (h1–h4) and per-object sampling (pos). The constants in the mathematical heuristics were determined using linear regression for 10,000 test measurements of different objects.

For each frame we calculate the squared difference between estimated and measured frame time and take the average of these values as a criterion for the quality of the rendering time estimation (Table 2.6). We use the city and the forest scene for this test.

In the city scene we can see that the proposed per-object sampling heuristic provides reasonable results, but due to the timing inaccuracy problem it is inferior to all heuristics except for the triangle count. The forest scene is more challenging to predict and here per-object sampling performs much better than the other algorithms. The newly proposed heuristic h4 based on adding geometry and rasterization terms performs better than the previous mathematical heuristics in both scenes, and is significantly better than all other heuristics in the city scene.

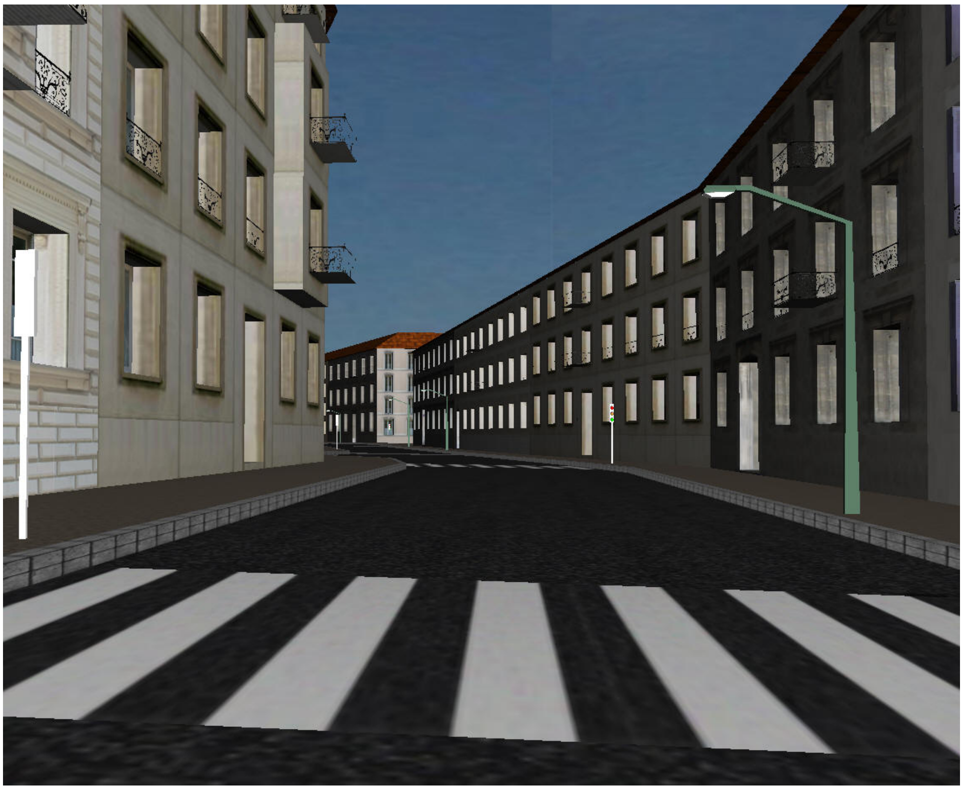


Figure 2.8: A snapshot of the city scene.

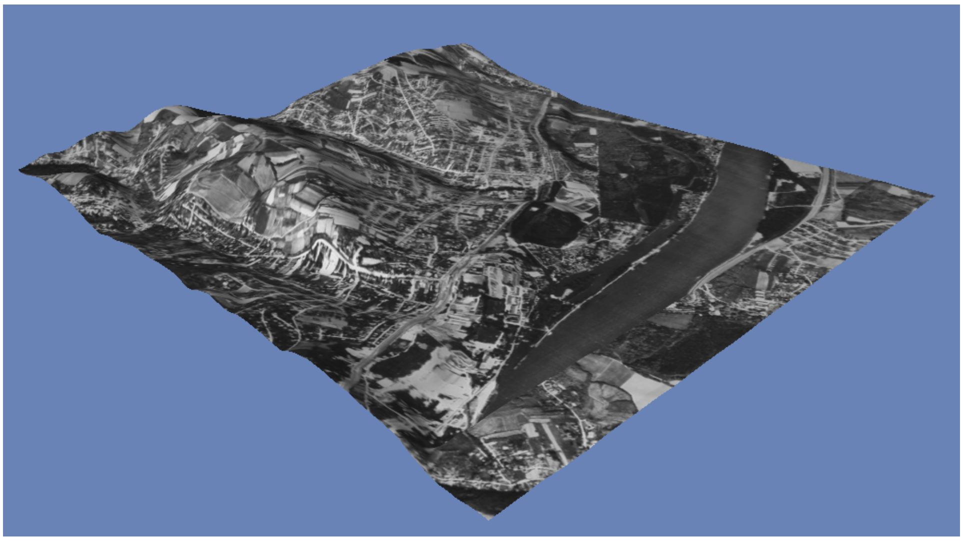


Figure 2.9: A snapshot of the terrain scene.

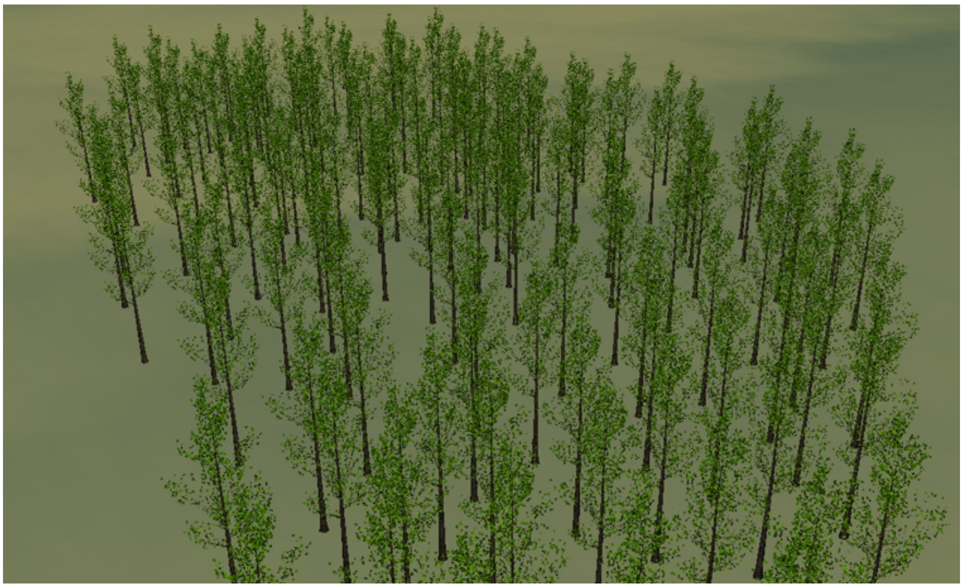


Figure 2.10: This figure shows a snapshot of the forest scene. Note that each tree consists of 6654 actually transformed vertices.

2.7.3 Comparing the per-view-cell estimation methods

For the city scene, we precalculated a PVS for a regular view-cell subdivision. Then we recorded a 1000 frame walkthrough for the per-view-cell sampling method and the per-object sampling method. Per-view-cell sampling underestimated 2 frames and resulted in an average squared error of 0.64. The per-object sampling method underestimated no frame, but due to the timing accuracy problem the average squared error was increased to 2.33.

2.7.4 Hardware extension

In the accompanying video, we simulated the conditional branching extension by randomly selecting some objects for some frames which receive a different LOD than the designated one, as would be the case if the rendering time estimation were incorrect. This effect is compared to the conventional frame-skip effect.

2.7.5 Discussion

The results obtained during our tests, especially for the per-viewpoint estimation methods, make it difficult to give a unique recommendation on which rendering time estimation to use. For example, the per-object sampling method (pos) gives better results than the mathematical heuristics in the forest scene, but is not so well suited for the city scene. The reason for this are the timing inaccuracies discussed in section 2.6.1. The individual objects in the forest scene consist of a much higher number of primitives than in the city scene, which reduces the influence of the inaccuracies and makes the estimations of the individual objects more precise. The mathematical heuristics, on the other hand, do not perform so well on the forest scene because this scene contains triangles of strongly varying screen projections, which influences the rendering time in ways that a single analytic formula is not able to capture.

Another factor which will influence the decision on a suitable rendering time estimation method is the involved computational effort. While the per-object sampling method can provide potentially superior results (see the forest scene), it also requires a costly sampling step for each individual object.

In summary, we recommend the per-object sampling method in cases when an exact estimation is critical and the mathematical heuristics fail. This is likely to happen for objects with many triangles of strongly varying screen projections. In all other cases, the newly proposed heuristic h4 presents an improvement over previous heuristics. However, when the hardware extensions proposed in this paper are available, the per-object sampling estimation will be significantly more attractive, if one is wiling to invest the effort in preprocessing.

2.8 Conclusions

In this paper, we introduced a framework for estimating the rendering time for both viewpoints and view cells in a real-time rendering system. We showed that previous work only captures certain aspects of the rendering time estimation and is only applicable under controlled circumstances. However, our aim is to achieve a system with a constant frame rate of at least 60 Hz in order to avoid annoying ghosting artifacts and frame skips. We propose several new rendering time estimation functions to be used in our framework, including one based on per-object sampling (pos), and one based on an improved mathematical heuristic (h4), and demonstrated their applicability in a walkthrough setting. While the new heuristics showed significant improvements over previous methods in some cases, we also observed that their effectiveness is hampered by limitations in current graphics hardware, which is not suited for constant frame rate systems. We therefore propose two hardware extensions that remedy this problem, one for accurate timing measurements, and one for conditional branches in the rendering pipeline.

Chapter 3

Light Space Perspective Shadow Maps

Published as:

- Michael Wimmer, Daniel Scherzer, and Werner Purgathofer:
  Light Space Perspective Shadow Maps
  In Alexander Keller and Henrik W. Jensen, editors, *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, pages 143–151. Eurographics, Eurographics Association, June 2004. ISBN 3-905673-12-6.

The following papers are extensions or alternative approaches related to this work [Wimm06b, Gieg07b, Gieg06, Gieg07a, Sche07]:

- Michael Wimmer and Daniel Scherzer:
  Robust Shadow Mapping with Light Space Perspective Shadow Maps
  In Wolfgang Engel, editor, *ShaderX 4 – Advanced Rendering Techniques*, volume 4 of *ShaderX*. Charles River Media, March 2006. ISBN 1-58450-425-0.
- Markus Giegl and Michael Wimmer:
  Queried Virtual Shadow Maps
  In Wolfgang Engel, editor, *ShaderX 5 – Advanced Rendering Techniques*, volume 5 of *ShaderX*. Charles River Media, December 2006. ISBN 1-58450-499-4.
- Markus Giegl and Michael Wimmer:
  Queried Virtual Shadow Maps
  In *Proceedings of ACM SIGGRAPH 2007 Symposium on Interactive 3D Graphics and Games*, pages 65–72, New York, NY, USA, April 2007. ACM Press. ISBN 978-1-59593-628-8.

- Markus Giegl and Michael Wimmer:
  Fitted Virtual Shadow Maps
  In *Proceedings of Graphics Interface 2007*, May 2007.

- Daniel Scherzer, Stefan Jeschke, and Michael Wimmer:
  Pixel-Correct Shadow Maps with Temporal Reprojection and Shadow Test Confidence
  In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering).* Eurographics, June 2007.

Light Space Perspective Shadow Maps

Michael Wimmer, Daniel Scherzer and Werner Purgathofer

Abstract

In this paper, we present a new shadow mapping technique that improves upon the quality of perspective and uniform shadow maps. Our technique uses a perspective transform specified in light space which allows treating all lights as directional lights and does not change the direction of the light sources. This gives all the benefits of the perspective mapping but avoids the problems inherent in perspective shadow mapping like singularities in post-perspective space, missed shadow casters etc. Furthermore, we show that both uniform and perspective shadow maps distribute the perspective aliasing error that occurs in shadow mapping unequally over the available depth range. We therefore propose a transform that equalizes this error and gives equally pleasing results for near and far viewing distances. Our method is simple to implement, requires no scene analysis and is therefore as fast as uniform shadow mapping.

3.1 Introduction

Shadows belong to the most important effects to convey realism in a computer-generated scene. One of the most popular shadow generation algorithms is *shadow mapping* [Will78], due to its simplicity, generality and high speed. With shadow mapping, the scene is first rendered from the view of the light, storing the depth values in a separate buffer. When the scene is then rendered from the normal viewing position, each pixel is transformed again into the light view, and its depth value compared to the depth value stored in the shadow map. If the depth value of the shadow map is nearer to the light source, the pixel is in shadow.

Like all image-space algorithms, shadow mapping suffers from aliasing artifacts due to quantization and perspective projection. Recently, several approaches have tried to reduce those aliasing artifacts. One particularly promising idea, *perspective shadow maps* [Stam02], is based on a perspective reparameterization of the shadow map. The original perspective shadow map technique suffers from a few drawbacks, which result mostly from the fact that the chosen perspective mapping is based on the observer projection:

- The lights have to be transformed into post-perspective space, and frequently change their type (from point to directional and vice versa, or from a normal

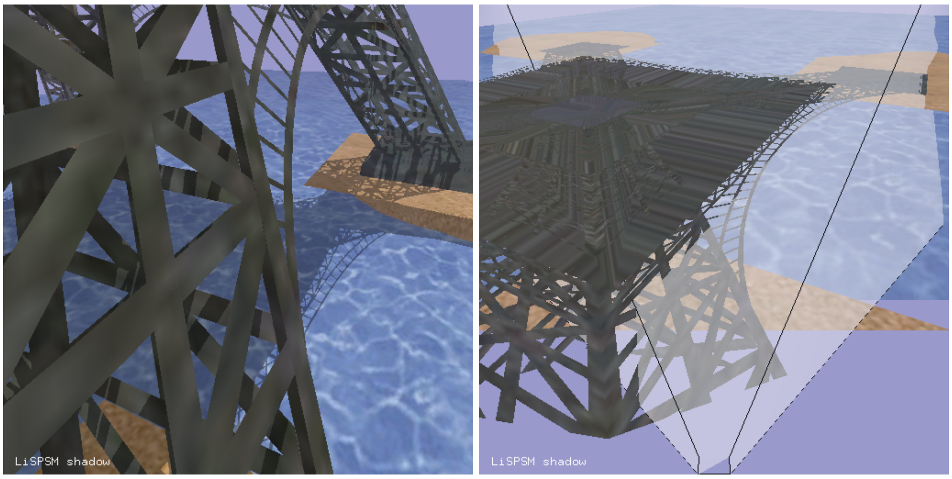


Figure 3.1: Light space perspective shadow maps (LiSPSM) (left) and the corresponding warped light view (including the eye frustum) (right). Note the high shadow detail both for near and distant objects.

to an “inverted” lightsource). Thinking in this particular post-perspective space is not intuitive.

- The mapping to post-perspective space has a singularity, which causes problems with shadow casters on or opposite the singularity. The practical solution to this problem is to “move” the viewpoint backwards for the shadow map generation until all relevant objects are included in front of the singularity. However, this reduces the shadow-map quality.
- Due to a number of special cases, the implementation is quite involved.
- The increase in shadow map resolution near the viewer comes at the expense of a drastic reduction of resolution for distant objects.

In this paper, we introduce *light space perspective shadow maps (LiSPSM)*, a new shadow mapping technique based on a variable perspective mapping specified in light space (see Figure 3.1 for an example). The advantage of the technique is that the chosen perspective mapping has no relevant singularities, allows treating all lights as directional lights and does not change the direction of the light sources. Therefore, most of the problems of perspective shadow maps are avoided. One of the most important contributions of this paper is a thorough error analysis of all shadow mapping techniques based on perspective reparameterizations. This analysis is then exploited to derive optimal parameters for our LiSPSM technique.

An important insight is that perspective shadow maps trade near resolution against large shadow errors in distant objects. In contrast, our method can be tuned to distribute the error equally or in a user-specified way among different distance regions. Light space perspective shadow maps are robust, as fast as standard shadow maps and simple to implement. Their major limitation is inherent to all methods based on reparameterizing the shadow plane, i.e., they can only deal with *perspective aliasing*. Dealing with more kinds of shadow aliasing usually requires a potentially costly scene analysis per frame.

The remainder of the paper is organized as follows: after a discussion of previous work in Section 3.2, we describe the LiSPSM technique in Section 3.3. In Section 3.4, we provide a thorough error analysis of uniform, perspective and light space perspective shadow maps and show how to optimally calculate the free parameter of LiSPSMs (n), which controls the distribution of aliasing error. Results are given in Section 3.5, and conclusions and ideas for future work in Section 3.6.

3.2 Previous work

The two most important categories of shadow algorithms are *shadow volumes* [Crow77] and *shadow mapping* [Will78]. Shadow volumes work in object space and are therefore potentially more accurate, but place a high burden on geometric and pixel processing and require well tesselated objects. Therefore, interest in shadow mapping techniques is large, since they work in image space and require only one additional rendering pass per light.

Most of the publications dealing with shadow maps try to solve the associated aliasing artifacts. Percentage closer filtering [Reev87] alleviates reprojection problems by filtering. A number of papers have tried to solve *perspective aliasing* due to the perspective view-frustum projection. The most prominent of these is the perspective shadow map method by Stamminger and Drettakis [Stam02], which tries to remove perspective aliasing by subjecting the shadow map to the same perspective transform as the viewer. Despite its drawbacks, this paper has inspired and opened the door to more general shadow map reparameterization approaches, of which we present one in this paper. Another way to reparameterize the shadow map is to tilt or warp the shadow plane directly [Chon04, Low03]. Recent approaches propose to combine shadow maps with shadow volumes or other primitives [Sen03, Govi03]. These techniques can potentially be combined with the light space perspective shadow maps presented in this paper.

Another approach to solve the aliasing problem are adaptive shadow maps [Fern01], where shadow maps are stored in a hierarchical fashion in order to provide more resolution where it is required due to different aliasing artifacts. However, the approach requires multiple readbacks and does not map well to current graphics hardware. Second depth shadow mapping [Wang94] can be used to reduce problems due to depth quantization and self occlusions. Brabec et al. [Brab02] improve uniform shadow map quality by focussing the shadow map to the visible scene.

As with all things related to real-time rendering, an excellent overview of shadow mapping and shadow algorithms in general can be found in Möller and Haines' Real-Time Rendering book [Möll02]. Finally, concurrent to our own work, there have been alternative developments to reduce shadow map aliasing [Kozl04, Mart04, Chon04, Aila04a].

3.3 Light space perspective shadow maps

3.3.1 Motivation

The goal of this work is to provide a fast, high-quality and robust shadow-map algorithm. Perspective shadow maps attempt to improve shadow quality by warping the shadow map according to a perspective transform given by the view transform. Our approach draws on this basic idea and improves upon it based on two main observations:

- While perspective transforms are valid tools to warp the shadow map, there is no reason that this perspective transform needs to be tied to the view frustum as in perspective shadow maps. In fact, any arbitrary perspective transformation could be used.
- Since the main goal of the perspective transform is to change the distribution of shadow map pixels, it is sufficient to use a warp that affects mainly the shadow map plane and not the axis perpendicular to the shadow map.

These two observations motivate a perspective transformation specified with respect to the coordinate axes of *light space*. In contrast to perspective shadow maps, this transformation has the important property that it does not change the direction of the light sources, and has no relevant singularities, because the view plane is parallel to the light vector. Directional lights remain directional lights in post-perspective space, while point lights are converted to directional lights as well. This results in a much more intuitive transformation, at the same time avoiding many of the problems found in perspective shadow maps.

Note however that our method only deals with the same aliasing errors as perspective shadow maps. In particular, projection errors due to surfaces parallel to light rays are not handled. An analysis of shadow mapping errors can be found in Section 3.4.

3.3.2 Overview

Light space perspective shadow maps are applied in the following steps:

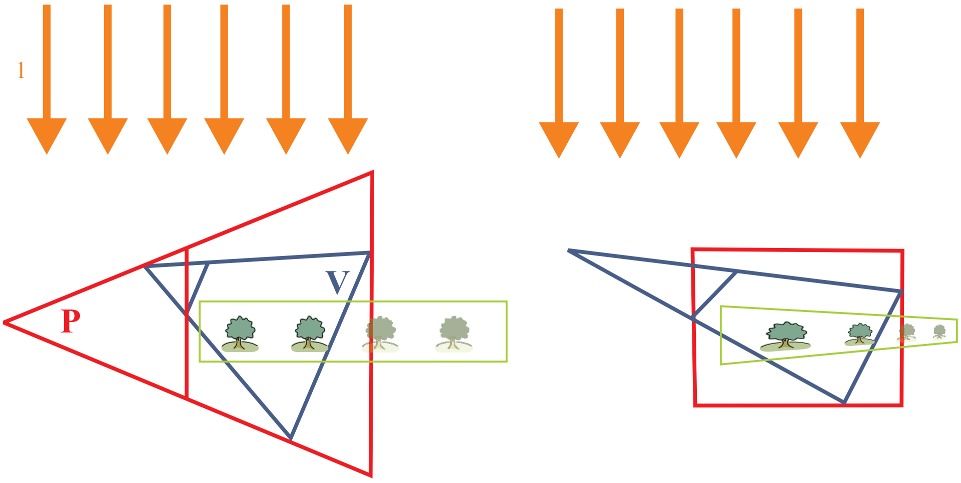


Figure 3.2: An example configuration of light space perspective shadow maps with view frustum V and the frustum defining the perspective transform P. Left: Note how the light rays l are parallel to the near and far plane of P. Right: After perspective transformation, the light direction is unchanged.

- Focus the shadow map on the convex body B that encloses all interesting light rays (i.e., the view frustum and all objects that can cast shadows into it).
- Enclose this body with an appropriate perspective frustum P that has a view vector parallel to the shadow map.
- By choosing the free parameter in P, the distance n of the projection reference point p to the near plane of the frustum, control the strength of the warping effect.
- Apply P both during shadow map generation rendering just as in standard shadow mapping.

Figure 3.2 shows an example configuration of light direction l, view frustum V and perspective frustum P, and the resulting warp.

3.3.3 Focussing the shadow map

The first step proceeds exactly as described by Stamminger and Drettakis [Stam02] for perspective shadow maps by *focussing* onto the convex body that is relevant for shadow calculation. This is done by calculating the convex hull of the view frustum and the light position (which is at infinity for a directional light) and intersecting this hull with the light frustum and the scene envelope (typically its bounding box). The result of this calculation is a convex body B described by a number of points.

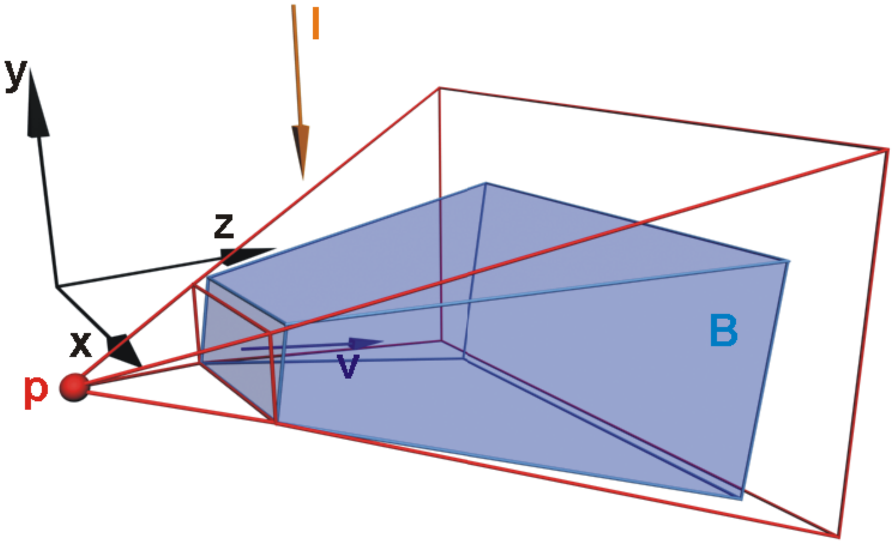


Figure 3.3: Construction of the perspective frustum P in 3D.

3.3.4 The perspective frustum in light space

The parameters for the perspective transform P can best be found in *light space*. We construct light space in the following way (see Figure 3.3): The y-axis is defined by the light vector l (but pointing towards the light). In the case of point lights, the spot direction vector is used as light vector (non-spot lights are usually not used for shadow mapping). The z-axis is defined to be perpendicular to the light vector and to lie in the plane containing the observer view vector v and the light vector. The x-axis complements the other two axes in order to form an orthogonal coordinate system. Note that in this definition of light space, the xz-plane is parallel to the shadow-map plane, and the z-axis corresponds roughly to the depth coordinate of the eye coordinate system. For illustration purposes, we will use a left-handed light space coordinate system in this paper, i.e., the z-axis has the same orientation as the view vector. In practice, the z-axis will usually be reversed to give a right-handed coordinate system, as is common for example in OpenGL.

For point lights, light space is defined as above, but *after* the perspective transform associated with the point light. There are no singularities in the combined perspective mapping, because the body B is completely in "front" of the light position and the view frustum (provided no objects straddle the near plane of the point light frustum). Thus, point lights can be treated as directional lights from this point on.

The *near* and *far planes* for the perspective frustum P are defined as planes parallel to the xy-coordinate plane, placed at the minimum and maximum light-space z-coordinate among the points of the body B.

The x and y coordinates of the *projection reference point* p for P are chosen so that the resulting frustum is roughly symmetrical, by taking the x-coordinate from the transformed viewpoint and the y-coordinate as the middle of the minimum and maximum y-coordinates of the body B.

3.3.5 Choosing the free parameter n

The *remaining free parameter* for P, the distance n of the projection reference point p to the near plane, influences how strong the shadow map will be warped. If it is chosen close to the near plane of P, perspective distortion will be strong, and the effect will resemble the original perspective shadow maps. If it is chosen far away from the far plane of P, the perspective effect will be very light, approaching uniform shadow maps. In Section 3.4, we will show that in the case of a view direction perpendicular to the light vector, the optimal choice for this parameter is $n_{opt} = z_n + \sqrt{z_f z_n}$, where z_n and z_f are the near and far plane distances of the eye view frustum. We will also show that in order to also give optimal results when the viewer is tilted towards the light or away from it, n has to be increased, so that it reaches infinity when the viewer looks exactly into the light or away from it. LiSPSMs then give exactly the same results as a uniform shadow map (which is optimal for this case).

Finally, the *frustum planes* are found by projecting all points of the body B onto the near plane of P and recording the maximum extents along the x and y axes of the near plane.

3.3.6 Applying the perspective frustum

Once the appropriate perspective frustum P has been found, its application is simple. The frustum combined with the usual projective mapping used for standard shadow maps. The mapping is used in two places, namely in the shadow map generation, and in the texture coordinate generation for shadow map rendering.

3.4 Analysis and optimal parameter estimation

In this section, we provide an analysis of shadow map aliasing errors, especially perspective aliasing, which is the error treated by our method. Our analysis is different from previous work in that we concentrate on worst-case errors in the center

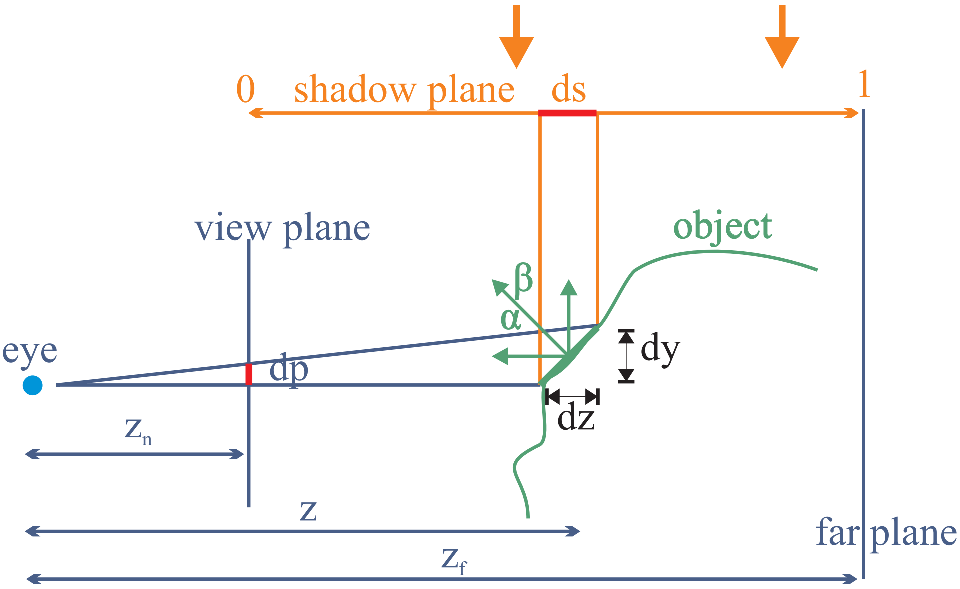


Figure 3.4: Aliasing in shadow mapping.

of view. We will show the ideal (yet impractical) logarithmic shadow map parameterization, and compare uniform, perspective and light space perspective shadow maps with regard to perspective aliasing. We will also show how the free parameter for light space perspective shadow maps can be chosen to provide optimal shadow resolution in the majority of cases.

In this section we will focus on directional light sources for two reasons: first, aliasing artifacts are often much worse for uniform light sources due to the wide range they have to cover (e.g., outdoor lighting), and second, point lights are mapped to directional lights in our approach.

3.4.1 What is perspective aliasing

We will briefly reiterate the main causes for shadow map aliasing for directional lights. Figure 3.4 shows a configuration for a small edge.

A pixel on the shadow map represents a shaft of light rays passing through it and has the size $ds \times ds$ in the local parameterization of the shadow map. Note that we assume a local parameterization of the shadow map which goes from 0 to 1 between near and far planes of the viewer—this already assumes that the shadow map has been properly focussed to the view frustum, not wasting any resolution on invisible parts of the scene. Introducing a local parameterization at this point also has the advantage that we will be able to compare different parameterizations. In world space, the shaft of rays has the length $dz = (z_f - z_n)ds$ for uniform shadow maps as an example.

The shaft hits a small edge along a length of $dz/\cos\beta$. This represents a length of $dy = dz\frac{\cos\alpha}{\cos\beta}$ in eye space, projecting to $dp = dy/z$ on screen. Note that we assume that the small edge can be translated along the z-axis. The shadow map aliasing error dp/ds is then

$$\frac{dp}{ds} = \frac{1}{z}\frac{dz}{ds}\frac{\cos\alpha}{\cos\beta}. \tag{3.1}$$

Shadow map *undersampling* occurs when dp is greater than the size of a pixel, or, for a viewport on the near plane of height 1, when dp/ds is greater than $res_{shadowmap}/res_{screen}$. As already shown by Stamminger and Drettakis, this can happen for two reasons: *perspective aliasing* when $\frac{dz}{zds}$ is large, and *projection aliasing* when $\cos\alpha/\cos\beta$ is large.

Projection aliasing occurs usually for surfaces almost parallel to the light direction. Reducing this kind of error would require higher sampling densities in such areas, which can only be found through an expensive scene analysis for each frame. In this paper, however, we concentrate on an approach which works without feedback, just like uniform or in some cases perspective shadow maps.

Perspective aliasing, on the other hand, is caused by the perspective projection of the viewer. If the perspective foreshortening effect occurs along one of the axes of the shadow map, it can be influenced by the *parameterization of the shadow map*. For uniform shadow maps, dz/ds is constant, therefore dp/ds is large when $1/z$ is large, which happens close to the near plane. In order to reduce perspective aliasing, it is therefore useful to analyze different shadow map parameterizations.

Note that perspective aliasing is the only aliasing error that can be improved using a global transformation like a perspective transform. Another important point to note is that the reparameterization is most effective when the view direction is perpendicular to the light direction. Otherwise, the depth range that can be influenced by the reparameterization decreases, down to the limit case where the view direction is parallel to the light vector. In this case, there is no perspective aliasing, and no global reparameterization of the shadow map can improve shadow map quality. This means that any such reparameterization should converge to uniform shadow maps in this situation.

Other image quality errors in shadow maps include resampling aliasing, which can be solved by percentage closer filtering [Reev87]; “swimming” artifacts, i.e., shadows that seem to frequently change their shape (this happens for all shadow map methods when they are undersampled except for the original “unfocussed” uniform shadow maps, which stay fixed in world space); and self-occlusion artifacts due to depth quantization errors.

3.4.2 Logarithmic shadow mapping

As has been shown in the previous subsection, perspective aliasing is the only shadow mapping problem that can be improved with a global approach, namely by changing the shadow map parameterization. An *optimal parameterization* would make dp/ds constant $= 1$ over the whole available depth range. For the ideal case of view direction perpendicular to light direction, this is (constants notwithstanding) equivalent to

$$ds = \frac{dz}{z}, \text{ i.e., } s = \int_0^s ds = \int_{z_n}^z \frac{dz}{z} = \ln \frac{z}{z_n}.$$

This shows that the optimal parameterization for shadow mapping (at least for directional lights) is logarithmic. Unfortunately, such a parameterization is not practical for hardware implementation. The logarithm could be applied in a vertex program on modern programmable hardware, however, pixel positions and all input parameters for pixel programs are interpolated hyperbolically. This makes graphics hardware amenable to perspective mappings, but not logarithmic ones. Still, it would be interesting to implement logarithmic shadow maps either in software or in a vertex program for finely tesselated scenes in order to see their quality advantage.

3.4.3 Analysis of light space perspective shadow maps

We will expand the analysis from the previous subsections in order to compare light space perspective shadow maps to uniform and perspective shadow maps. We first discuss the case in which the view direction is perpendicular to the light direction. Note that this is also the ideal case for any method that tries to reduce perspective aliasing, since the whole depth range in the view frustum is available to influence the shadow map parameterization.

Figure 3.5 shows the setup for light space perspective shadow maps. The view frustum V (which is assumed to be identical to B here) is enclosed by the perspective frustum P as described in Section 3.3.4. The values z_n, z_f and z describe the distance of the near plane, the far plane and an arbitrary point respectively to the viewpoint, whereas n, f and z' represent the distances of the same entities to the projection reference point p. In order to analyze the aliasing error dp/ds, we need to find the shadow map parameterization $s = s(z)$ that is induced by the perspective transform P. The effect of P on s is more easily described using the z'-coordinate (when using, for example, the OpenGL `glFrustum` command to generate P):

$$s = \frac{1}{2} + \frac{f+n}{2(f-n)} + \frac{fn}{z'(f-n)}. \tag{3.2}$$

After substituting $z' = z - z_n + n$, differentiation gives:

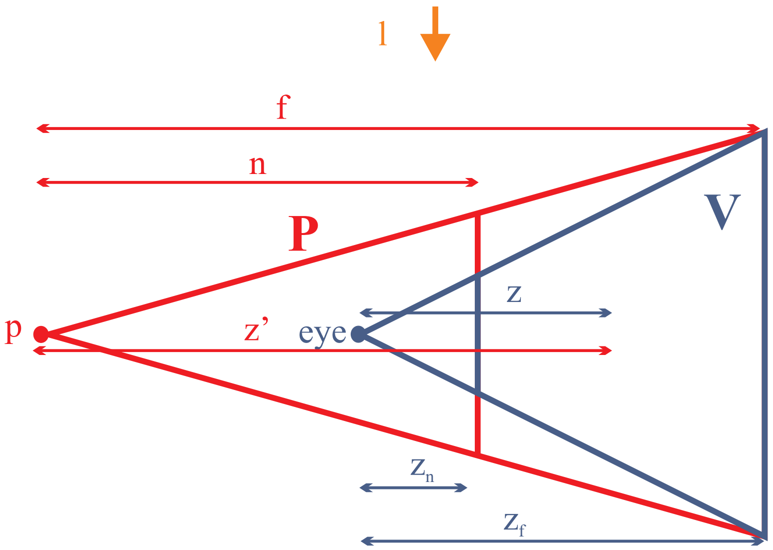


Figure 3.5: The parameterization of light space perspective shadow maps (shows the yz-plane in light space). The parameter n is free and can vary between z_n (perspective shadow mapping) and infinity (uniform shadow mapping).

$$\frac{ds}{dz} = \frac{fn}{(z - z_n + n)^2 (f - n)}.$$

Substituting $f = n + z_f - z_n$ and plugging this into equation (3.1) (assuming projection aliasing $= 1$) gives:

$$\frac{dp}{ds} = \frac{(z - z_n + n)^2}{z} \frac{(z_f - z_n)}{n(n + z_f - z_n)}. \tag{3.3}$$

Equation (3.3) can now be used to analyze all three shadow mapping methods. Sending n to infinity corresponds to uniform shadow maps and gives $(z_f - z_n)/z$, i.e., the error decreases hyperbolically with increasing distance, which is of course to be expected. More interesting is the case for perspective shadow maps, where $n = z_n$. Surprisingly, the result is a linear dependence on z:

$$\frac{dp}{ds} = z \frac{z_f - z_n}{z_n z_f}.$$

This means that for perspective shadow maps, the perspective aliasing error is very small at the near plane. However, with increasing distance, the error increases rapidly. This can be seen in Figure 3.6, which plots the perspective errors of uniform and perspective shadow maps for $z_n = 1$ and $z_f = 100$. See Section 3.4.5 for an explanation why this is consistent with the claim that PSMs are optimal for this

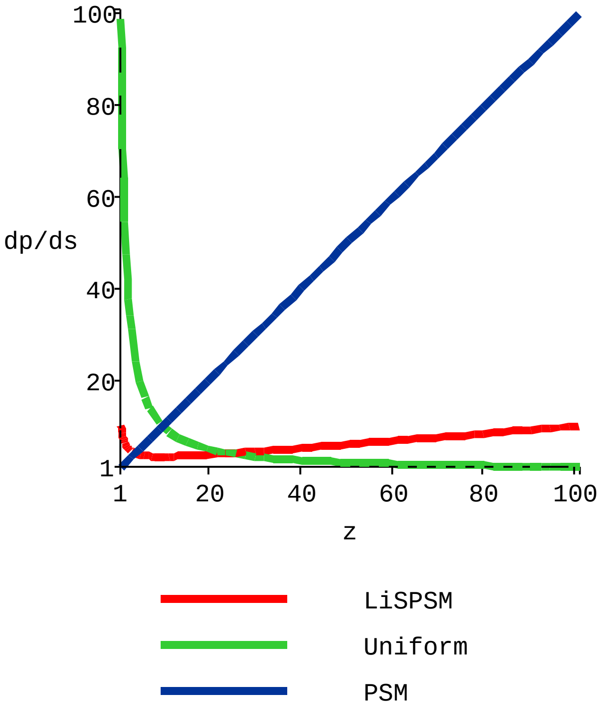


Figure 3.6: Perspective aliasing errors plotted against z-coordinate for different shadow mapping techniques.

situation, given a different error analysis. Note again that uniform shadow maps are assumed to be focussed.

Light space shadow maps lie between these two extremes of uniform and perspective shadow maps, depending on the parameter n. There are many ways to choose this parameter, including user choice or sampling several positions on screen. However, we opted for a choice which gives the optimal error distribution along the whole depth range with respect to the $\mathbf{L}_{max}$ norm. First, analyzing equation (3.3) shows that the function has only one positive local extremum, a minimum at location $n - z_n$. Therefore, the maxima within the relevant z-range $[z_n, z_f]$ are located at the boundaries of the range. Consequently, the minimal $\mathbf{L}_{max}$ norm is achieved when both maxima are equal. Substituting first z_n and then z_f into equation (3.3) and solving for n yields the desired parameter value for the ideal case of view direction perpendicular to light direction:

$$n_{opt} = z_n + \sqrt{z_f z_n}.$$

The associated maximum error rises roughly like $\sqrt{z_f}$, which is much better than both uniform and perspective shadow maps, which have a maximum error of

z_f. In addition to the perspective errors for uniform and perspective shadow maps, Figure 3.6 also shows the error for light space perspective shadow maps (LiSPSM) with $n = n_{opt}$. It can be seen that the error for LiSPSMs decreases quickly to the local minimum at $\sqrt{z_n z_f}$, then rises practically linearly until reaching again the maximum error at the far plane. One could say that LiSPSMs inherit the hyperbolic falloff of uniform shadow maps at the near plane, and the linear increase from perspective shadow maps from the minimum on, but both with far smaller error maxima.

Finally, this treatment has only dealt with perspective aliasing in the z-direction. In the x-direction, the situation is slightly different. The problematic behavior of perspective shadow maps for the z-direction arises from the fact that the z coordinate is projected into the shadow map s coordinate, which is subject to foreshortening. The x-coordinate, however, is projected into the shadow map t coordinate which is orthogonal to s, and this coordinate is *not* subject to perspective foreshortening. On the contrary, t undergoes the same perspective transformation as x, and is therefore ideal in the perspective shadow map approach, i.e., dp/dt is constant.

For light space perspective shadow maps, the situation is a bit different. The perspective aliasing error dp/dt evaluates to

$$\frac{ds}{dt} = \frac{z - z_n + n}{z} = 1 + \frac{\sqrt{z_n z_f}}{z},$$

where n has been replaced by n_{opt} in the right term. So, the error starts with a maximum about the same order of magnitude than dp/ds and then falls off hyperbolically. This means that error is distributed fairly between the x and z direction, whereas in perspective shadow maps, the x direction is ideal and the z direction can show dramatic errors. Note that the unequal treatment of errors in the z and x direction is inherent to all approaches that reparameterize the shadow plane.

3.4.4 General case

So far, the discussion has only dealt with the ideal case of a view direction perpendicular to the light direction. In practice, the light will rarely come directly from above, and the observer will also want to look up and down. As discussed in Section 3.4.1, this decreases the available depth range, and in the limit case of the light direction parallel to the view direction, no reparameterization of the shadow map can improve its quality, so the parameterization should converge to uniform shadow mapping.

In the general case, the eye space z-coordinate does not correspond directly to the light space z-coordinate. This has to be taken into account in the derivation of n_{opt} via the tilt angle γ, i.e., the angle between the light direction and the view vector. The main difference in the derivation of the perspective aliasing error dp/ds

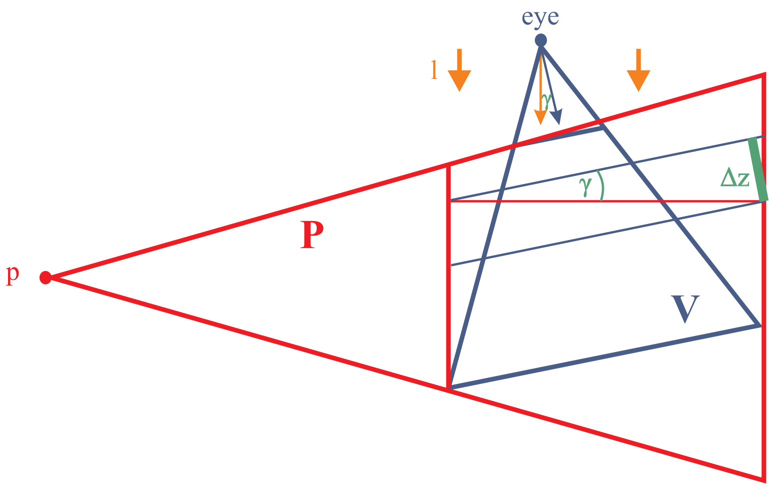


Figure 3.7: The z-range affected by the reparameterization is small when the view direction gets near the light direction. For perspective shadow maps, V and P would be identical, but the effect of the warp on the z-distribution would be the same.

is that in equation (3.2), z' has to be assumed as $z' = (z - z_n)/\sin\gamma + n$ instead of the formula given there. Leading this through, the optimal parameter value only has to be adjusted slightly as $n'_{opt} = n_{opt}/\sin\gamma$.

This choice causes the perspective warping effect to decrease with increasing tilt angle. For $\gamma = 0$, n'_{opt} goes to infinity, totally removing the perspective transform. This means that LiSPSMs converge to uniform shadow maps as desired. For $\gamma = \pi/2$, on the other hand, the original formula results.

There are certain additional intricacies involved in the general case. Due to the tilt of the view frustum with respect to the light direction, the warping effect of the shadow map reparameterization is also tilted with respect to the view frustum, which can reduce the range of z values that are actually affected by the warp. Figure 3.7 shows such a constellation. In such cases, it is advisable to replace in the calculation of n'_{opt} z_f by $z_n + \Delta z$, where $\Delta z = (f - n)\sin\gamma$. This will cause the mapping to converge faster to the uniform shadow map.

3.4.5 Discussion

Uniform and perspective shadow maps (PSMs) are at the opposite ends of a spectrum of error distributions: while uniform shadow maps show most error close to the viewer and then gain rapidly in quality, PSMs have the best quality near the viewer and then lose quality. The good results reported for PSMs seem surprising seeing that in Figure 3.6, uniform shadow maps surpass PSMs quite quickly in quality, even though this is supposed to be the ideal case for PSMs. The main reason for this is that our error analysis is more robust than the one used by Stamminger and Drettakis.

Our analysis of dp/ds (here discussed again for the case of view direction perpendicular to light direction) is based on a small edge at the center of the viewport that is translated in z-direction. This occurs for example when the viewpoint or an object is translated in z-direction, or for objects in a scene that have varying depths (see Figure 3.8). To extend the analysis to a general edge, it can be shown that the term $1/z$ (together with the trigonometric term) in equation (3.1) needs to be replaced by $k/z - y/z^2$, where k is the slope of the edge. For $y = 0$ and $k = 1$, our analysis results. In contrast, the error analysis done by Stamminger and Drettakis assumes small edges that are distributed within a single line (in 2D). This corresponds to restricting y to lie on a line with the same slope as the edge: $y = kz - y_0$ with y_0 arbitrary. The term $1/z$ from our analysis then becomes y_0/z^2, which, if used for the derivation of equation (3.3), indeed gives the claimed constant error dp/ds along the chosen line for PSMs (Figure 3.8). However, the problem for PSMs is that the value of this constant can be quite large—it essentially behaves as shown in Figure 3.6 when translated in z-direction. This can also be seen in typical light views for PSMs, where almost no area is reserved for far objects. In order to somewhat mitigate this quick increase in error, the authors push the near plane as far as possible into the scene. Unfortunately, this requires a readback from graphics hardware, which we avoid for LiSPSMs.

In essence, LiSPSMs are more robust because they minimize the worst-case perspective aliasing error, which occurs near the center of the viewport, by blending smoothly between uniform and perspective shadow maps. This is in contrast to PSMs, which equalize the error along any given plane, but this error can be quite large depending on the distance of this plane. Our optimal parameter choice limits the error of LiSPSMs to about the square root of the maximum error of the other two approaches (which both have the same maximum error) in our error analysis. Note that $\mathbf{L}_{max}$ with respect to the z-coordinate might not be the best error criterion, as nearby objects are far more important visually. On the other hand, the ratio between errors in near and far regions can be easily modified, giving, for example, even more weight to nearby objects at the expense of distant objects.

We also want to stress that the analysis in this section was based on directional light sources. While point lights are mapped to directional lights automatically by our approach, the perspective transform native to the point light may invalidate some of these findings for the case of point lights. However, we do not consider this a big limitation of our analysis. On the one hand, point lights in indoor environments or other “near”, i.e., very “perspective” point lights are not as prone to perspective aliasing anyway, and uniform shadow maps might be sufficient in these cases. On the other hand, “far” point lights behave more like directional lights and should therefore be covered well by our analysis. Furthermore, there is a growing interest in using directional lights for large outdoor environments, for example in games.

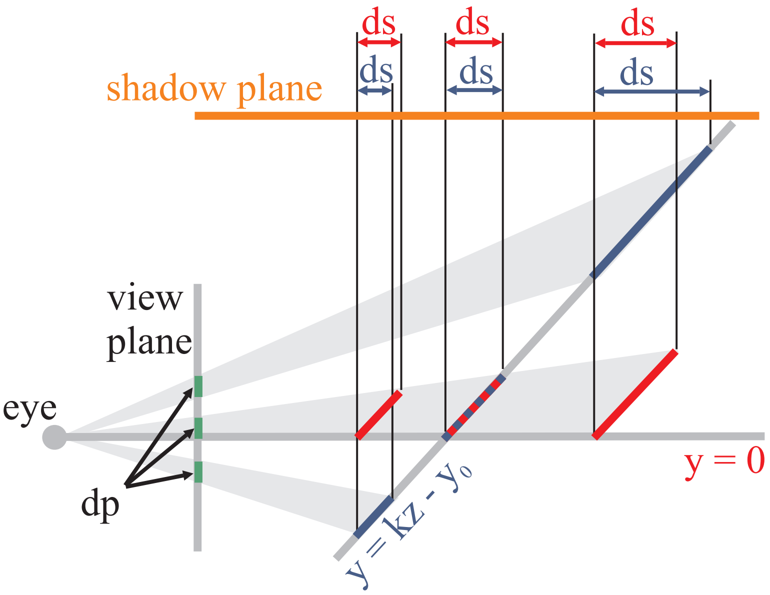


Figure 3.8: Shows the lines along which the perspective error is analyzed. PSMs have constant error on small edges along the sloped line, whereas LiSPSMs distribute error evenly for small edges along the line with $y = 0$. The warping effect for PSMs is much stronger, which is in general not desirable.

3.5 Results

We have implemented the light space perspective shadow map algorithm, and for comparison also uniform shadow mapping and perspective shadow mapping. The implementation runs on GeForce3-class or higher hardware and uses only OpenGL ARB extensions. Performance was not measured separately, as LiSPSMs run as fast as standard shadow mapping (i.e., with at least 75 fps for all tested scenes on an Intel Pentium4 2.4GHz and an NVIDIA GeForceFX 5900 graphics card). All images presented in the following (Figure 3.9) were captured using a 512x512 pixel viewport and a shadow map resolution of 1024x1024 pixels, using varying directional light directions. Bilinear filtering was enabled, but we didn't implement percentage-closer filtering. The view frustum field of view was set to 60°, the near plane to 1.0, and the far plane to 400, which corresponds approximately to the the scene extent.

Uniform shadow maps were always focussed on the relevant scene parts (unfocussed uniform shadow maps are practically unusable for larger scenes). For perspective shadow maps, we did not implement a read-back of the frame buffer as suggested by the authors, but pushed the near plane to the nearest intersection with an object bounding box in the view.

The images demonstrate our findings that uniform shadow maps work well

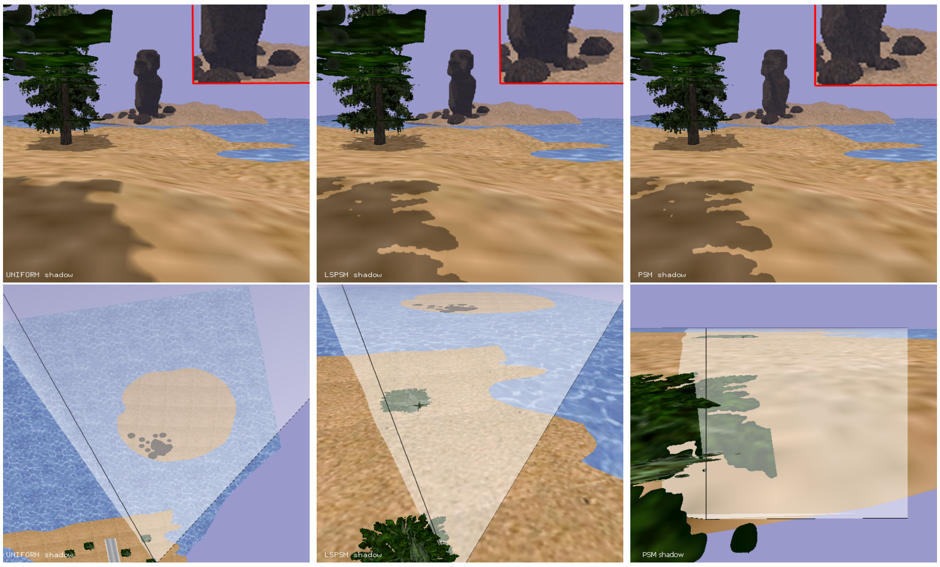


Figure 3.9: Uniform (left), light space perspective (middle) and perspective (right) shadow maps. The second row shows images rendered from the the warped light views, which also include the eye frusta (shaded transparently). The view positions show shadows for near, medium and distant (zoomed in the inset) objects. Uniform shadow maps capture the distant shadow best, perspective shadow maps the near shadow, and LiSPSM work well for both. Models courtesy of www.3dcafe.com. For more results see also the attached high-resolution images and videos. A sample implementation with source code is available at the author's webpage.

for distant objects, perspective shadow maps for close objects, whereas LiSPMs distribute the perspective error equally among near and distant objects, providing good results for both. Note that in some cases, projection aliasing can be seen, which cannot be solved with any technique relying on reparameterization.

3.6 Conclusions and future work

We have presented light space perspective shadow maps, a practical shadow mapping technique that combines the advantages of perspective and uniform shadow maps, provides overall high shadow quality and avoids the pitfalls of perspective shadow maps. The algorithm is robust, simple to implement and as fast as standard shadow maps.

We have also conducted a thorough error analysis of perspective aliasing errors in different shadow mapping techniques and shown our algorithm to be in a certain sense optimal among perspective shadow map reparameterizations, and not far from the ideal, logarithmic, parameterization.

In terms of future work, it would be interesting to investigate logarithmic and hierarchical parameterizations. Another idea is using multiple shadow maps for different depth regions. Figure 3.6 practically invites such a partition into two depth ranges, for example at the crossover point between uniform and perspective shadow map errors.

Chapter 4

Hardware Occlusion Queries Made Useful

Published as:

- Michael Wimmer and Jiří Bittner:
  Hardware Occlusion Queries Made Useful
  In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, March 2005. ISBN 0-32133-559-7.

The following paper is related to this work [Bitt04]:

- Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer:
  Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful
  Computer Graphics Forum, 23(3):615–624, September 2004. Proceedings EUROGRAPHICS 2004, ISSN 0167-7055.

Hardware Occlusion Queries Made Useful

Michael Wimmer and Jiří Bittner

4.1 Introduction

Hardware occlusion queries are one of the most eagerly awaited graphics hardware features in a long time. This feature makes it possible for an application to ask the 3D API (OpenGL or Direct3D) whether or not any pixels would be drawn if a particular object was rendered. With this feature, applications can check to see if the bounding boxes of complex objects are visible or not, skipping them if their bounds are occluded. Hardware occlusion queries are appealing to today's games because they work in completely dynamic scenes.

However, now that this feature has been available for more than two years (via the `NV/ARB_occlusion_query` OpenGL extension and by the `IDirect3DQuery9` interface in Direct3D), there is not yet widespread adoption of this feature for solving visibility culling in rendering engines. This is due to two main problems related to naive usage of the occlusion query feature: the overhead caused by issuing the occlusion queries themselves (since each query adds an additional draw call), and the latency caused by waiting for the query results.

In this chapter, we present a simple but powerful algorithm that solves these problems [Bitt04]: it minimizes the number of issued queries and reduces the delays due to the latency of query results.

To achieve this, the algorithm exploits the spatial and temporal coherence of visibility by reusing the results of occlusion queries from the last frame in order to initiate and schedule the queries in the next frame. This is done by storing the scene in a hierarchical data structure (such as a k-d tree, an octree, and so on [Cohe03], processing nodes of the hierarchy in a front-to-back order, and interleaving occlusion queries with the rendering of certain previously visible nodes.

The algorithm is smart about the queries it actually needs to issue: most visible interior nodes of the spatial hierarchy and many small invisible nodes are not tested at all. Furthermore, previously visible nodes are rendered immediately without waiting for their query result, which allows filling the time needed to wait for query results with useful work.

4.2 For Which Scenes Are Occlusion Queries Effective?

Let's recap briefly for which scenes and situations the algorithm presented in this chapter is useful.

Occlusion queries work best for large scenes where only a small portion of the scene is visible in each frame—for example, a walkthrough in a city. More generally, the algorithm presented here works if in each frame there are large occluded interior nodes in the hierarchy. These nodes should contain many objects, so that skipping them if they are occluded saves geometry, overdraw, and draw calls (which are a typical bottleneck in today's rendering systems [Wloka 2003]). An important advantage the algorithm has over most other techniques used nowadays in games (such as portal culling) is that it allows completely dynamic environments.

Some games—for example, shader-heavy ones—use a separate initial rendering pass that writes only to the depth buffer. Subsequent passes test against this depth buffer, and thus expensive shading is done only for visible pixels. If the scenes are complex, our occlusion-culling algorithm can be used to accelerate establishing the initial depth buffer and at the same time to obtain a complete visibility classification of the hierarchy. For the subsequent passes, we skip over whole objects or groups of objects that are completely invisible at virtually no cost.

Finally, if there is little or no occlusion in your scene—for example, a flyover of a city—there is no benefit to occlusion culling. Indeed, occlusion queries potentially introduce an overhead in these scenes and make rendering slower, no matter how the occlusion queries are used. If an application uses several modes of navigation, it will pay to switch off occlusion culling for modes in which there is no or only very little occlusion.

4.3 What Is Occlusion Culling?

The term *occlusion culling* refers to a method that tries to reduce the rendering load on the graphics system by eliminating (that is, culling) objects from the rendering pipeline if they are hidden (that is, occluded) by other objects. There are several methods for doing this.

One way to do occlusion culling is as a preprocess: for any viewpoint (or regions of viewpoints), compute ahead of time what is and is not visible. This technique relies on most of the scene to be static (so visibility relationships do not change) and is thus not applicable for many interactive and dynamic applications [Cohe03].

Portal culling is a special case of occlusion culling; a scene is divided into cells (typically rooms) connected via portals (typically door and window openings). This structure is used to find which cells (and objects) are visible from which other cells—either in a preprocess or on the fly [Cohe03]. Like general preprocess occlusion culling, it relies on a largely static scene description, and the room metaphor restricts it to mostly indoor environments.

Online occlusion culling is more general in that it works for fully dynamic scenes. Typical online occlusion-culling techniques work in image space to reduce computation overhead. Even then, however, CPUs are less efficient than, say, GPUs for performing rasterization, and thus CPU-based online occlusion techniques are typically expensive [Cohe03].

Fortunately, graphics hardware is very good at rasterization. Recently, an OpenGL extension called `NV_occlusion_query`, or now `ARB_occlusion_query`, and Direct3D's occlusion query (`D3DQUERYTYPE_OCCLUSION`) allow us to query the number of rasterized fragments for any sequence of rendering commands. Testing a single complex object for occlusion works like this (see also [Seku04]:

1. Initiate an occlusion query.
2. Turn off writing to the frame and depth buffer, and disable any superfluous state. Modern graphics hardware is thus able to rasterize at much higher speed [NVID04].
3. Render a simple but conservative approximation of the complex object—usually a bounding box: the GPU counts the number of fragments that would actually have passed the depth test.
4. Terminate the occlusion query.
5. Ask for the result of the query (that is, the number of visible pixels of the approximate geometry).
6. If the number of pixels drawn is greater than some threshold (typically zero), render the complex object.

The approximation used in step 3 should be simple so as to speed up the rendering process, but it must cover at least as much screen-space area as the original object, so that the occlusion query does not erroneously classify an object as invisible when it actually is visible. The approximation should thus be much faster to render, and not modify the frame buffer in any way.

This method works well if the tested object is really complex, but step 5 involves waiting until the result of the query actually becomes available. Since, for example, Direct3D allows a graphics driver to buffer up to three frames of rendering commands, waiting for a query results in potentially large delays. In the rest of this chapter, we refer to steps 1 through 4. as "issuing an occlusion query for an object."

If correctness of the rendered images is not important, a simple way to avoid the delays is to check for the results of the queries only in the following frame [Seku04]. This obviously leads to visible objects being omitted from rendering, which we avoid in this chapter.

4.4 Hierarchical Stop-and-Wait Method

As a first attempt to use occlusion queries, we show a naive occlusion-culling algorithm and extend it to use a hierarchy. In the following section, we show our coherent hierarchical culling algorithm, which makes use of coherence and other tricks to be much more effective than the naive approach.

4.4.1 The Naive Algorithm, or Why Use Hierarchies at All?

To understand why we want to use a hierarchical algorithm, let's take a look at the naive occlusion query algorithm first:

1. Sort objects front to back
2. For each object
    (a) Issue occlusion query for the object (steps 1-4 from previous section)
    (b) Stop and wait for result of query
    (c) If number of visible pixels greater than 0
        i. Render the object

Although this algorithm calculates correct occlusion information for all of our objects, it is most likely slower than just rendering all the objects directly. The reason for this is that we have to issue a query and wait for its result for every object in the scene!

Now imagine, for example, a walkthrough of a city scene: We typically see a few hundred objects on screen (buildings, streetlights, cars, and more). But there might be tens of thousands of objects in the scene, most of which are hidden by nearby buildings. If each of these objects is not very complex, then issuing and waiting for queries for all of them is more expensive than just rendering them.

4.4.2 Hierarchies to the Rescue!

We need a mechanism to group objects close to one another so we can treat them as a single object for the occlusion query. That's just what a spatial hierarchy does. Examples of spatial hierarchies are k-d trees, BSP trees, and even standard bounding-volume hierarchies. They all have in common that they partition the scene recursively until the cells of the partition are "small" enough according to some criterion. The result is a tree structure with interior nodes that group other nodes, and leaf nodes that contain actual geometry.

The big advantage of using hierarchies for occlusion queries is that we can now test interior nodes, which contain much more geometry than the individual objects.

In our city example, hundreds or even thousands of objects might be grouped into one individual node. If we issue an occlusion query for this node, we save the tests of all of these objects—a potentially huge gain!

If geometry is not the main bottleneck, but rather the number of draw calls issued [Wlok03], then making additional draw calls to issue the occlusion queries is a performance loss. With hierarchies, though, interior nodes group a larger number of draw calls, which are all saved if the node is found occluded using a single query. So we see that in some cases, a hierarchy is what makes it possible to gain anything at all by using occlusion queries.

4.4.3 Hierarchical Algorithm

The naive hierarchical occlusion-culling algorithm works like this; it is specified for a node within the hierarchy and initially called for the root node:

1. Issue occlusion query for the node
2. *Stop and wait* for the result of the query
3. If the node is visible
   (a) If it is an interior node
      i. Sort the children in front-to-back order
      ii. Call the algorithm recursively for all children
   (b) If it is a leaf node
      i. Render the objects contained in the node

This algorithm can potentially be much faster than the basic naive algorithm, but it has two significant drawbacks.

4.4.4 Problem 1: Stalls

The first drawback it shares with the naive algorithm. Whenever we issue an occlusion query for a node, we cannot continue our algorithm until we know the result of the query. But the time until the query result becomes available may be prohibitive. The query has to be sent to the GPU. There it will sit in a command queue until all previous rendering commands have been issued (and modern GPUs can queue up more than one frame's worth of rendering commands!). Then the bounding box associated with the query must be rasterized, and finally the result of the query has to travel back over the bus to the driver.

During all this time, the CPU sits idle, and we have caused a CPU stall. But that's not all. While the CPU sits idle, it doesn't feed the GPU any new rendering

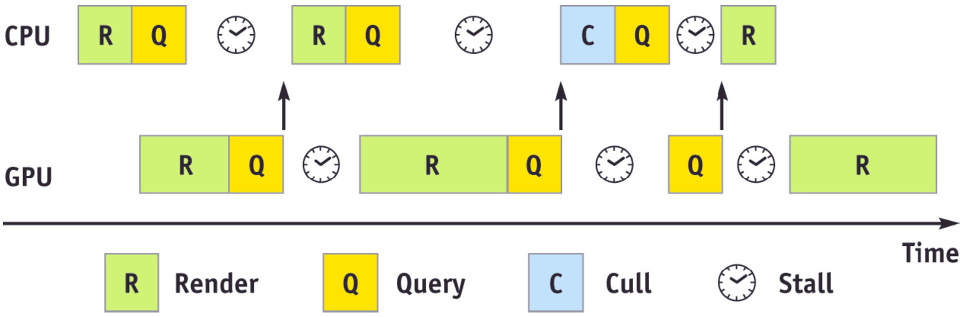


Figure 4.1: CPU stalls and GPU starvation caused by Occlusion Queries.

commands. Now when the result of the occlusion query arrives, and the CPU has finally figured out what to draw and which commands to feed the GPU next, the command buffer of the GPU has emptied and it has become idle for some time as well. This we call GPU starvation. Obviously, we are not making good use of our resources. Figure 4.1 sums up these problems.

4.4.5 Problem 2: Query Overhead

The second drawback of this algorithm runs contrary to the original intent of the hierarchical method. We wanted to reduce the overhead for the occlusion queries by grouping objects together. Unfortunately, this approach increases the number of queries (especially if many objects are visible): in addition to the queries for the original objects, we have to add queries for their groupings. So we have improved the good case (many objects are occluded), but the bad case (many objects are visible) is even slower than before.

The number of queries is not the only problem. The new queries are for interior nodes of the hierarchy, many of which, especially the root node and its children, will have bounding boxes covering almost the entire screen. In most likelihood, they are also visible. In the worst case, we might end up filling the entire screen tens of times just for rasterizing the bounding geometry of interior nodes.

4.5 Coherent Hierarchical Culling

As we have seen, the hierarchical stop-and-wait method does not make good use of occlusion queries. Let us try to improve on this algorithm now.

4.5.1 Idea 1: Being Smart and Guessing

To solve problem 1, we have to find a way to avoid the latency of the occlusion queries. Let's assume that we could "guess" the outcome of a query. We could

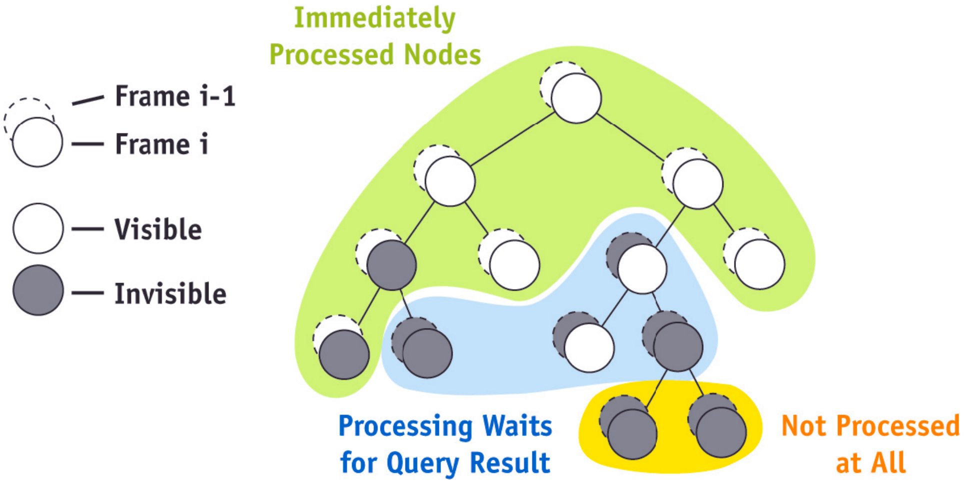


Figure 4.2: Processing Requirements for Various Nodes

then react to our guess instead of the actual result, meaning we don't have to wait for the result, eliminating CPU stalls and GPU starvations.

Now where does our guess come from? The answer lies, as it so often does in computer graphics, in temporal coherence. This is just another way of expressing the fact that in real-time graphics, things don't move around too much from one frame to the next. For our occlusion-culling algorithm, this means that if we know what's visible and what's occluded in one frame, it is very likely that the same classification will be correct for most objects in the following frame as well. So our "guess" is simply the classification from the previous frame.

But wait—there are two ways in which our guess can go wrong: If we assume the node to be visible and it's actually occluded, we will process the node needlessly. This can cost some time, but the image will be correct. However, if we guess that a node is occluded and in reality it isn't, we won't process it and some objects will be missing from the image—something we need to avoid!

So if we want to have correct images, we need to verify our guess and rectify our choice in case it was wrong. In the first case (the node was actually occluded), we update the classification for the next frame. In the second case (the node was actually visible), we just process (that is, traverse or render) the node normally. The good news is that we can do all of this later, whenever the query result arrives. Note also that the accuracy of our guess is not critical, because we are going to verify it anyway. Figure 4.2 shows the different cases.

4.5.2 Idea 2: Pull Up, Pull Up

To address problem 2, we need a way to reduce overhead caused by the occlusion queries for interior nodes.

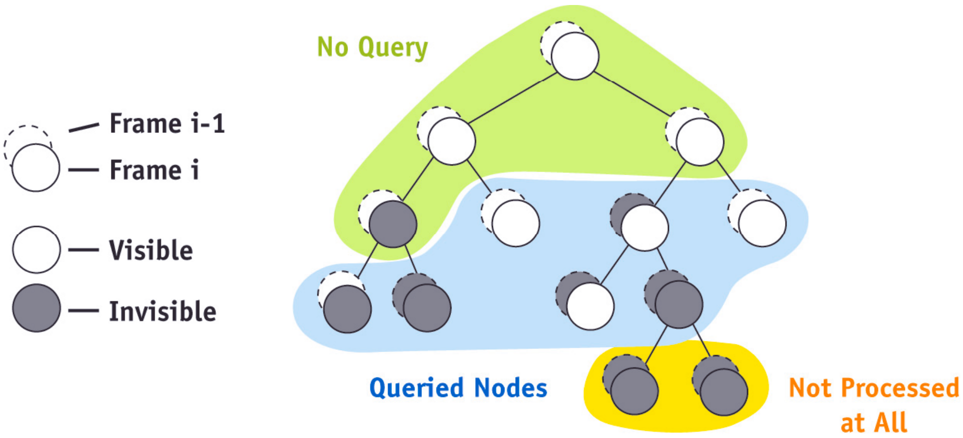


Figure 4.3: Occlusion Query Requirements for Various Nodes.

Luckily, this is easy to solve. Using idea 1, we are already processing previously visible nodes without waiting for their query results anyway. It turns out that we don't even need to issue a query for these nodes, because at the end of the frame, this information can be easily deduced from the classification of its children, effectively "pulling up" visibility information in the tree. Therefore, we save a number of occlusion queries and a huge amount of fill rate.

On the other hand, occlusion queries for interior nodes that were occluded in the previous frame are essential. They are required to verify our choice not to process the node, and in case the choice was correct, we have saved rendering all the draw calls, geometry, and pixels that are below that node (that is, inside the bounding box of the node).

What this boils down to is that we issue occlusion queries only for previously visible leaf nodes and for the largest possible occluded nodes in the hierarchy (in other words, an occluded leaf node is not tested if its parent is occluded as well). The number of queries issued is therefore linear in the number of visible objects. Figure 4.3 illustrates this approach.

4.5.3 Algorithm

We apply these two ideas in an algorithm we call "coherent hierarchical culling." In addition to the hierarchical data structure for the front-to-back traversal we already know, we need a "query queue" that remembers the queries we issued previously. We can then come back later to this queue to check whether the result for a query is already available.

The algorithm is easy to incorporate into any engine as it consists of a simple main traversal loop. A queue data structure is part of the C++ standard template

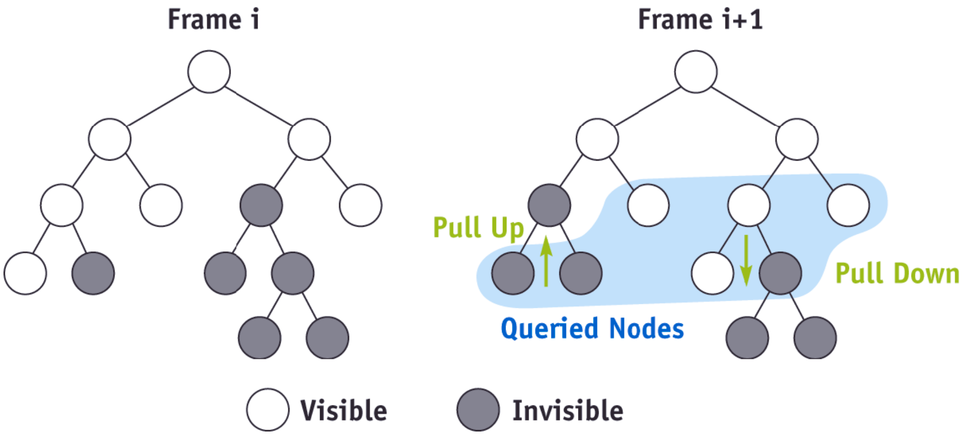


Figure 4.4: Visibility of Hierarchy Nodes in Two Consecutive Frames.

library (STL), and a hierarchical data structure is part of most rendering libraries anyway, for example, for collision detection.

The algorithm visits the hierarchy in a front-to-back order and immediately recurses through any previously visible interior node (idea 1). For all other nodes, it issues an occlusion query and stores it in the query queue. If the node was a previously visible leaf node, it also renders it immediately, without waiting for the query result.

During the traversal of the hierarchy, we want to make sure that we incorporate the query results as soon as possible, so that we can issue further queries if we encounter a change in visibility.

Therefore, after each visited node, we check the query queue to see if a result has already arrived. Any available query result is processed immediately. Queries that verify our guess to be correct are simply discarded and do not generate additional work. Queries that do not verify our guess are handled as follows: Nodes that were previously visible and became occluded are marked as such. Nodes that were previously occluded and became visible are traversed recursively (interior nodes) or rendered (leaf nodes). Both of these cases cause visibility information to be pulled up through the tree. See Figure 4.4, which also depicts a so-called “pull-down” situation, where a previously occluded node has become visible and its children need to be traversed

4.5.4 Implementation Details

The algorithm is easy to follow using the pseudocode, which we show in Figures 4.5 and 4.6. Let’s discuss some of the details in the code.

To maintain the visibility classification for all nodes over consecutive frames, we use a `visible` flag and a `frameID`, which increments every frame. Nodes visible in the previous frame are easily identified through `lastVisited = frameID - 1` and `visible = true`; all other nodes are assumed invisible. This way, we don't have to reset the visibility classification for each frame.

To establish the new visibility classification for the current frame, we set all nodes we encounter during the traversal to invisible by default. Later, when a query identifies a node as visible, we "pull up" visibility; that is, we set all its ancestors to visible as well.

Interior nodes that were visible in the last frames (that is, not `leafOrWasInvisible`) are nodes for which we skip occlusion queries altogether.

Traversing a node means rendering the node for leaf nodes or pushing its children onto the traversal stack so that they get rendered front to back.

We also assume that the `Render()` call renders an object only if it hasn't been rendered before (this can be checked using the `frameID`). This facilitates the code for previously visible objects: when we finally get the result of their visibility query, we can just process them again without introducing a special case or rendering them twice (since they have already been rendered immediately after their query was issued). Another advantage is that we can reference objects in several nodes of the hierarchy, and they still get rendered only once if several nodes of the hierarchy are visible.

Finally, we have to be careful with occlusion tests for nodes near the viewpoint. If the front faces of a bounding box get clipped by the near plane, no fragments are generated for the bounding box and the node would be classified as occluded by the test even though the node is most likely visible. To avoid this, we should check for each bounding box that passes the view-frustum test whether any of its vertices is closer than the near plane. In such a case, the associated node should be set visible without issuing an occlusion query. Note that for the occlusion queries, the actual node bounding boxes can be used instead of the (usually less tight) nodes of the spatial hierarchy.

4.5.5 Why Are There Fewer Stalls?

Let's take a step back and see what we have achieved with our algorithm and why.

The coherent hierarchical culling algorithm does away with most of the inefficient waiting times found in the hierarchical stop-and-wait method. It does so by interleaving the occlusion queries with normal rendering, and avoiding the need to wait for a query to finish in most cases.

If the viewpoint does not move, then the only point where we actually might have to wait for a query result is at the end of the frame, when all the visible geometry is already rendered. Previously visible nodes are rendered right away without waiting for the results.

```
TraversalStack.Push(hierarchy.Root);
while ( not TraversalStack.Empty() or
        not QueryQueue.Empty() )
{
  //-- PART 1: process finished occlusion queries
  while ( not QueryQueue.Empty() and
         (ResultAvailable(QueryQueue.Front()) or
          TraversalStack.Empty()) )
  {
    node = QueryQueue.Dequeue();
    // wait if result not available
    visiblePixels = GetOcclusionQueryResult(node);
    if ( visiblePixels > VisibilityThreshold )
    {
      PullUpVisibility(node);
      TraverseNode(node);
    }
  }

  //-- PART 2: hierarchical traversal
  if ( not TraversalStack.Empty() )
  {
    node = TraversalStack.Pop();
    if ( InsideViewFrustum(node) )
    {
      // identify previously visible nodes
      wasVisible = node.visible and
                   (node.lastVisited == frameID - 1);
      // identify nodes that we cannot skip queries for
      leafOrWasInvisible = not wasVisible or IsLeaf(node);
      // reset node's visibility classification
      node.visible = false;
      // update node's visited flag
      node.lastVisited = frameID;
      // skip testing previously visible interior nodes
      if ( leafOrWasInvisible )
      {
        IssueOcclusionQuery(node);
        QueryQueue.Enqueue(node);
      }
      // always traverse a node if it was visible
      if ( wasVisible )
        TraverseNode(node);
    }
  }
}
```

Figure 4.5: Algorithm pseudocode, part 1.

```
TraverseNode(node)
{
  if ( IsLeaf(node) )
    Render(node);
  else
    TraversalStack.PushChildren(node);
}

PullUpVisibility(node)
{
  while (!node.visible)
  {
    node.visible = true;
    node = node.parent;
  }
}
```

Figure 4.6: Algorithm pseudocode, part 2.

If the viewpoint does move, the only dependency occurs if a previously invisible node becomes visible. We obviously need to wait for this to be known in order to decide whether to traverse the children of the node. However, this does not bother us too much: most likely, we have some other work to do during the traversal. The query queue allows us to check regularly whether the result is available while we are doing useful work in between.

Note that in this situation, we might also not detect some occlusion situations and unnecessarily draw some objects. For example, if child A occludes child B of a previously occluded interior node, but the query for B is issued before A is rendered, then B is still classified as visible for the current frame. This happens when there is not enough work to do between issuing the queries for A and B. (See also Section 4.5.7, where we show how a priority queue can be used to increase the chance that there is work to do between the queries for A and B.)

4.5.6 Why Are There Fewer Queries?

We have already seen that a hierarchical occlusion-culling algorithm can save a lot of occlusion queries if large parts of the scene are occluded. However, this is paid for by large costs for testing interior nodes.

The coherent hierarchical culling algorithm goes a step further by obviating the need for testing most interior nodes. The number of queries that needs to be issued is only proportional to the number of visible objects, not to the total number of objects in the scene. In essence, the algorithm always tests the largest possible nodes for occluded regions.

Neither does the number of queries depend on the depth of the hierarchy, as in the hierarchical stop-and-wait method. This is a huge win, because the rasterization of large interior nodes can cost more than occlusion culling buys.

4.5.7 How to Traverse the Hierarchy

We haven't talked a lot about traversing the hierarchy up to now. In principle, the traversal depends on which hierarchy we use, and is usually implemented with a traversal stack, where we push child nodes in front-to-back order when a node is traversed. This basically boils down to a depth-first traversal of the hierarchy.

However, we can gain something by not adhering to a strict depth-first approach. When finding nodes that have become visible, inserting these nodes into the traversal should be compatible with the already established front-to-back order in the hierarchy.

The solution is not to use a traversal stack, but a priority queue based on the distance to the observer. A priority queue makes the front-to-back traversal very simple. Whenever the children of a node should be traversed, they are simply inserted into the priority queue. The main loop of the algorithm now just checks the priority queue instead of the stack for the next node to be processed.

This approach makes it simple to work with arbitrary hierarchies. The hierarchy only needs to provide a method to extract its children from a node, and to compute the bounding box for any node. This way, it is easy to use a sophisticated hierarchy such as a k-d tree [Cohe03], or just the bounding volume hierarchy of the scene graph for the traversal, and it is easy to handle dynamic scenes.

Note also that an occlusion-culling algorithm can be only as accurate as the objects on which it operates. We cannot expect to get accurate results for individual triangles if we only ever test nodes of a coarse hierarchy for occlusion. Therefore, the choice of a hierarchy plays a critical role in occlusion culling.

4.6 Optimizations

We briefly cover a few optimizations that are beneficial in some (but not all) scenes.

4.6.1 Querying with Actual Geometry

First of all, a very simple optimization that is always useful concerns previously visible leaf nodes [Seku04]. Because these will be rendered regardless of the outcome of the query, it doesn't make sense to use an approximation (that is, a bounding box) for the occlusion query. Instead, when issuing the query as described in

Section 4.3, we omit step 2 and replace step 3 by rendering the actual geometry of the object. This saves rasterization costs and draw calls as well as state changes for previously visible leaf nodes.

4.6.2 Z-Only Rendering Pass

For some scenes, using a z-only rendering pass can be advantageous for our algorithm. Although this entails twice the transformation cost and (up to) twice the draw calls for visible geometry, it provides a good separation between occlusion culling and rendering passes as far as rendering states are concerned. For the occlusion pass, the only state that needs to be changed between an occlusion query and the rendering of an actual object is depth writing. For the full-color pass, visibility information is already available, which means that the rendering engine can use any existing mechanism for optimizing state change overhead (such as ordering objects by rendering state).

4.6.3 Approximate Visibility

We might be willing to accept certain image errors in exchange for better performance. This can be achieved by setting the `VisibilityThreshold` in the algorithm to a value greater than zero. This means that nodes where no more than, say, ten or twenty pixels are visible are still considered occluded. Don’t set this too high, though; otherwise the algorithm culls potential occluders and obtains the reverse effect.

This optimization works best for scenes with visible complex geometry, where each additional culled object means a big savings.

4.6.4 Conservative Visibility Testing

Another optimization makes even more use of temporal coherence. When an object is visible, the current algorithm assumes it will also be visible for the next frame. We can even go a step further and assume that it will be visible for a number of frames. If we do that, we save the occlusion queries for these frames (we assume it’s visible anyway). For example, if we assume an object remains visible for three frames, we can cut the number of required occlusion queries by almost a factor of three! Note, however, that temporal coherence does not always hold, and we almost certainly render more objects than necessary.

This optimization works best for deep hierarchies with simple leaf geometry, where the number of occlusion queries is significant and represents significant overhead that can be reduced using this optimization.

4.7 Conclusion

Occlusion culling is an essential part of any rendering system that strives to display complex scenes. Although hardware occlusion queries seem to be the long-sought-after solution to occlusion culling, they need to be used carefully, because they can introduce stalls into the rendering pipeline.

We have shown an algorithm that practically eliminates any waiting time for occlusion query results on both the CPU and the GPU. This is achieved by exploiting temporal coherence, assuming that objects that have been visible in the previous frame will remain visible in the current frame. The algorithm also reduces the number of costly occlusion queries by using a hierarchy to cull large occluded regions using a single test. At the same time, occlusion tests for most other interior nodes are avoided.

This algorithm should make hardware occlusion queries useful for any application that features a good amount of occlusion, and where accurate images are important. For example, Figure 4.7 shows the application of the algorithm to the walkthrough of a city model, with the visibility classification of hierarchy nodes on the right. The orange nodes were found visible; all the other depicted nodes are invisible. Note the increasing size of the occluded nodes with increasing distance from the visible set. For the shown viewpoint, the algorithm presented in this chapter provided a speedup of about 4 compared to rendering with view-frustum culling alone, and a speedup of 2.6 compared to the hierarchical stop-and-wait method.

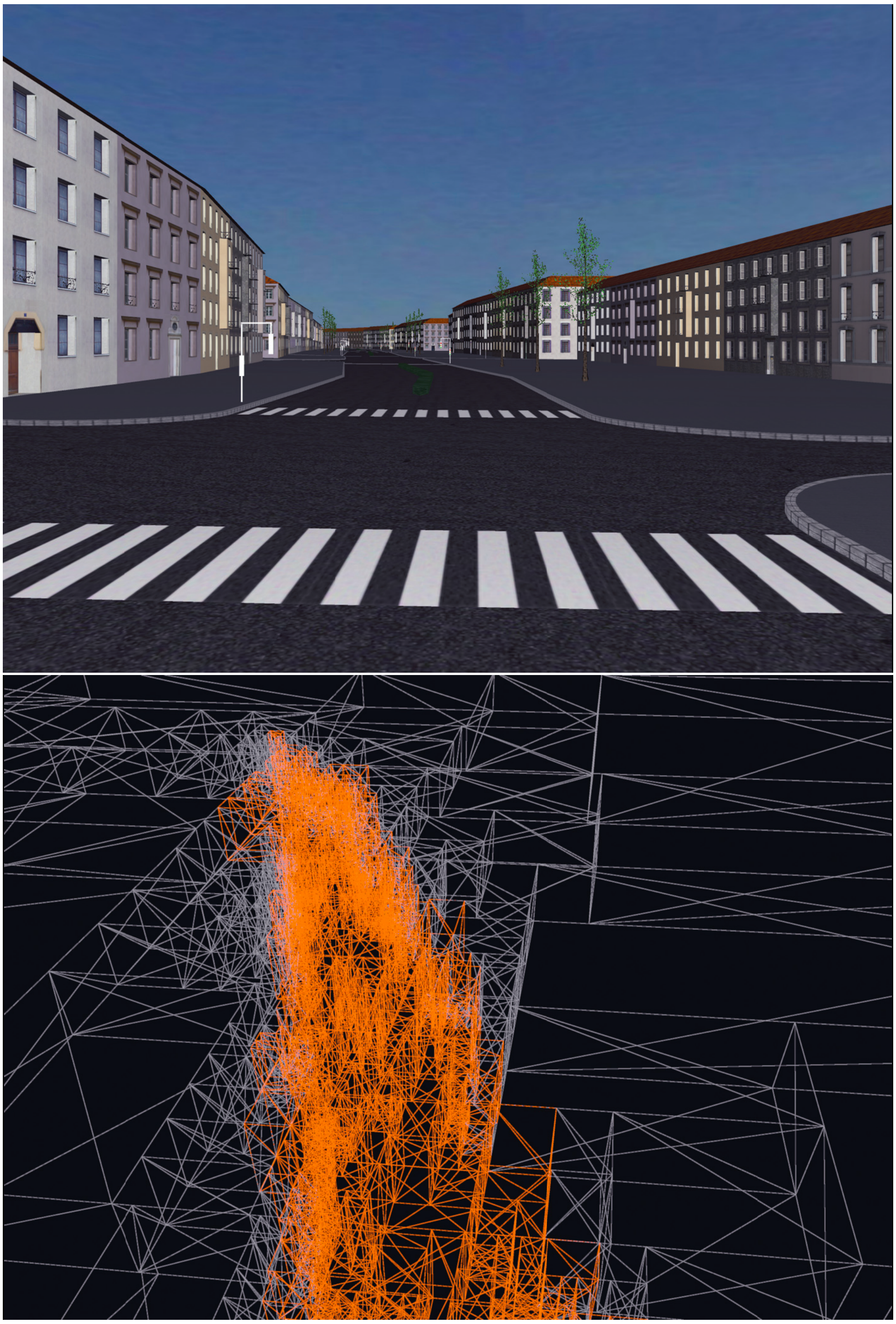


Figure 4.7: Visualizing the Benefits of Occlusion Queries for a City Walkthrough.

Chapter 5

Guided Visibility Sampling

Published as:

- Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov:
  Guided Visibility Sampling
  ACM Transactions on Graphics, 25(3):494–502, July 2006. Proceedings ACM SIGGRAPH 2006, ISSN 0730-0301.

The following papers are related to visibility preprocessing [Bitt01, Bitt05, Matt06, Matt07]:

- Jiří Bittner, Peter Wonka, and Michael Wimmer:
  Visibility Preprocessing for Urban Scenes using Line Space Subdivision
  In Bob Werner, editor, *Proceedings of Pacific Graphics 2001 (Ninth Pacific Conference on Computer Graphics and Applications)*, pages 276–284, Los Alamitos, CA, October 2001. IEEE Computer Society Press. ISBN 0-7695-1227-5.
- Jiří Bittner, Peter Wonka, and Michael Wimmer:
  Fast Exact From-Region Visibility in Urban Scenes
  In Kavita Bala and Philip Dutré, editors, *Rendering Techniques 2005 (Proceedings Eurographics Symposium on Rendering)*, pages 223–230. Eurographics, Eurographics Association, June 2005. ISBN 3-905673-23-1.
- Oliver Mattausch, Jiří Bittner, and Michael Wimmer:
  Adaptive Visibility-Driven View Cell Construction
  In Wolfgang Heidrich and Tomas Akenine-Moller, editors, *Rendering Techniques 2006 (Proceedings Eurographics Symposium on Rendering)*, pages 195–206. Eurographics, Eurographics Association, June 2006. ISBN 3-90567-335-5.

- Oliver Mattausch, Jiří Bittner, Peter Wonka, and Michael Wimmer:
  Optimized Subdivisions for Preprocessed Visibility
  In *Proceedings of Graphics Interface 2007*, May 2007.

Guided Visibility Sampling

Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov

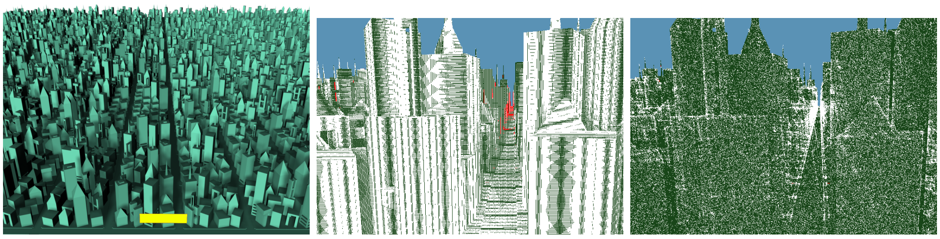


Figure 5.1: Visualization of sampling strategies (white pixels show a subset of the actual samples, missed geometry is marked red). Left: An urban input scene and a view cell (in yellow) for visibility sampling. Middle: Previous visibility sampling algorithms repeatedly sample the same triangles in the foreground while missing many smaller triangles and distant geometry. Right: Our solution is guided by scene visibility and therefore quickly finds most visible triangles while requiring drastically fewer samples than previous methods.

Abstract

This paper addresses the problem of computing the triangles visible from a region in space. The proposed aggressive visibility solution is based on stochastic ray shooting and can take any triangular model as input. We do not rely on connectivity information, volumetric occluders, or the availability of large occluders, and can therefore process any given input scene. The proposed algorithm is practically memoryless, thereby alleviating the large memory consumption problems prevalent in several previous algorithms. The strategy of our algorithm is to use ray mutations in ray space to cast rays that are likely to sample new triangles. Our algorithm improves the sampling efficiency of previous work by over two orders of magnitude.

5.1 Introduction

Visibility is a fundamental problem in computer graphics: visibility computations are necessary for occlusion culling, shadow generation, inside-outside classifications, image-based rendering, motion planning, and navigation, to name just a few examples. While visibility from a single viewpoint can be calculated quite easily, many applications require the potentially visible set (PVS) for a region in

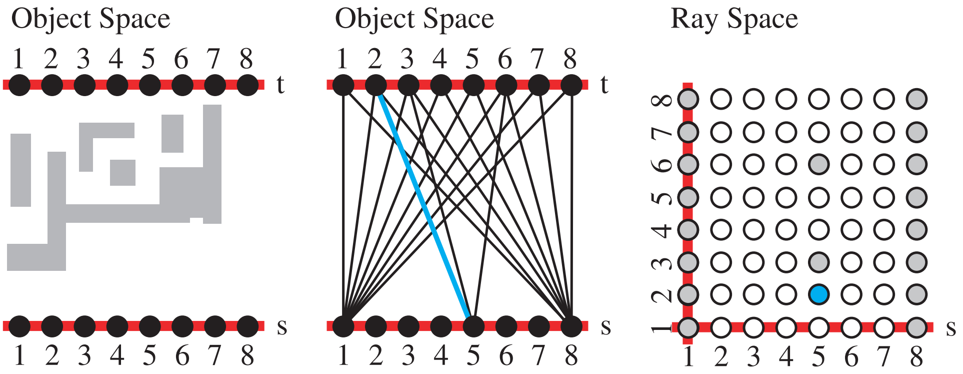


Figure 5.2: Sampling in object and ray space. Left: a scene with a set of objects. A view cell is shown as a line segment parameterized with s. We are interested in all rays that intersect the view cell and a second line segment parameterized with t. Middle: Shows a subset of the possible rays. One ray is highlighted in blue. Right: A depiction of the discrete ray space. Any ray in the middle figure corresponds to a point in ray space. The blue point corresponds to the blue ray in the middle figure.

space, which is, unfortunately, much more complicated. A number of excellent from-region visibility algorithms exist, but most of them are only applicable to a limited range of scenes, require complex computations, and sometimes significant amounts of memory. Therefore, sampling-based solutions have become very popular for practical applications due to their robustness, general applicability, and ease of implementation. In this paper we will improve upon previous sampling-based algorithms by significantly improving the sampling efficiency, i.e., the number of samples required to detect a certain set of visible polygons.

To motivate our design choices, we will look at two key aspects of any visibility algorithm: the behavior of the algorithm in ray space, and the data structure used to store and acquire visibility information.

Figure 5.2 illustrates the concept of ray space in 2D. Given a view cell, shown as edge parameterized with s, and a scene with objects shown in grey, we can compute visibility by considering all rays from the view cell to a plane behind the scene, parameterized with t. For a 2D scene, this is a 2D set of rays; for a 3D scene this is a 4D set of rays. If this set of rays is sampled densely enough, we will have a good visibility solution.

The inefficiency of a pure regular sampling approach as shown in Figure 5.2 is that the same surfaces are sampled over and over again (note that the definition of regular depends on the parameterization of ray space!). Therefore, it would be beneficial if we could only sample areas that have not been sampled before. This is shown in Figure 5.3, where after an initial orthogonal sampling, only few additional rays are needed to find all visible surfaces. In total, little more than a 1D subspace of the 2D ray space needs to be explored in this example. This is

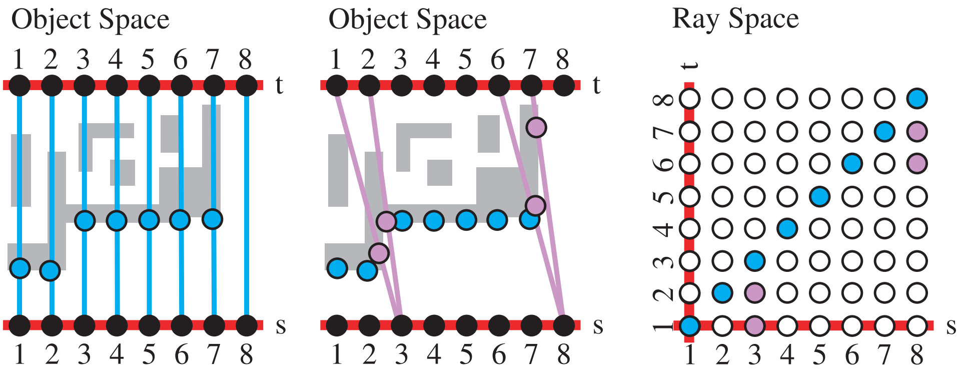


Figure 5.3: Left: The scene sampled orthogonally. Middle: Additional samples to capture oblique surfaces. Right: The rays used to sample the scene are shown in corresponding colors.

due to the spatial coherence of visibility. In this paper, we exploit this coherence: starting from stochastically sampled points, we grow lower-dimensional subspaces of ray space using the newly introduced strategies of *adaptive border sampling* and *reverse sampling*, which are guided by the properties of scene visibility.

The second key aspect of a visibility algorithm is what data structure is used to store visibility information. The most complete, but also complex, way is to store 4D ray space. For large scenes, this entails prohibitive levels of memory consumption. Conservative algorithms often store the shadow volume, whereas sampling algorithms use the volume of 3D space that has not been sampled yet (the so-called void volume, Figure 5.4); but these data structures still require several times the memory taken by the scene description itself. Alternatively, the boundary of the void volume (the void surface [Pito99]) can be used, which is easy to sample from one point in space, but difficult to manipulate. In this paper, we do not store visibility information beyond the PVS at all, relying on our new reverse sampling approach to penetrate the void surface based on the current sample only.

The key contribution of this paper is an intelligent sampling algorithm that drastically improves the performance of previous sampling approaches by combining random sampling with deterministic exploration phases. The algorithm requires little memory, is simple to implement, accepts any triangular test scene as input, and can be used as a general purpose visibility tool.

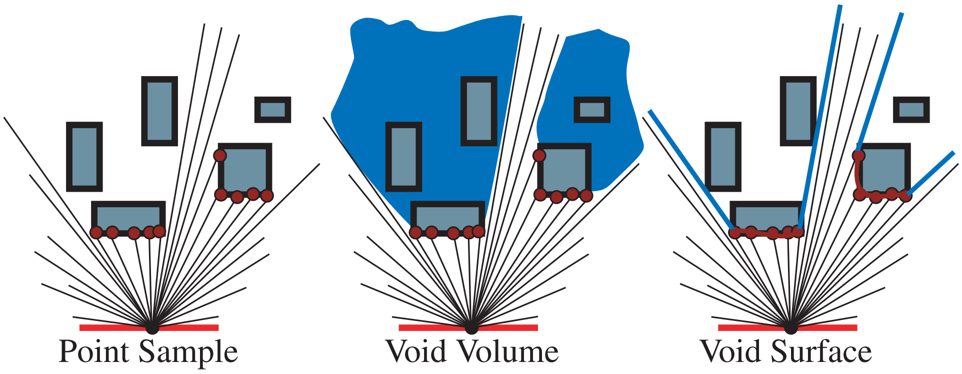


Figure 5.4: Representing visibility from a single point. Left: independent samples. Middle: the void volume. Right: the void surface.

5.2 Overview

5.2.1 Problem Statement

We consider visibility problems that are posed as follows: As first input we take a three-dimensional scene consisting of a set of triangles, TS. We do not rely on connectivity information, volumetric objects, or large polygons as potential occluders (such a set of triangles is often called triangle soup). As second input we consider a subset of ray space Ω, usually defined by the rays emanating within a 3D polyhedron called *view cell* and intersecting the bounding box of the scene. A ray can be defined by a starting point and a direction. Using TS and Ω, we can define a visibility function $v : \Omega \rightarrow TS$, so that each ray in Ω maps to the triangle in TS that it intersects first.

The exact solution of the visibility problem is the range of this function, $v(\Omega) \subseteq TS$, also called exact visible set EVS. Our algorithm is *aggressive* ([Nire02]), i.e., it calculates a potentially visible set $PVS \subseteq EVS$.

Our algorithm can be used to solve the visibility problem in different applications (see Section 5.5.6). A *usage scenario* to keep in mind for the following exposition is a visibility preprocessing system for real-time rendering: the *view space* (set of possible observer locations) is partitioned into view cells. In a *preprocessing* step, our algorithm is used to calculate and store a PVS for each view cell (note that only its boundary polygons are taken into account, since any ray leaving the view cell can be represented by a ray on the boundary). At runtime, the view cell corresponding to the current observer location is determined, and only the objects in the associated PVS are sent to the graphics hardware, leading to significant savings in rendering time.

5.2.2 Algorithm Overview

The algorithms in this paper are based on ray shooting and assume the capability to trace a ray x and compute its first intersection with a scene triangle $t \in TS$, i.e., to compute the triangle $t = v(x)$ (fast ray tracers include OpenRT [Wald03] and the recently presented MLRTA [Resh05]).

The idea of a sampling solution is to select a sequence of rays $\mathbf{X} = x_i$, trace the ray and add the triangle $v(x_i)$ to the visibility set PVS.

In this paper, we will address the problem on how to sample *efficiently*, that is how to improve the chances of finding new triangles. We will start with one of the most popular sampling strategies, random sampling (Section 5.3.1). Then we will show how to use visibility information from previous samples to construct intelligent sampling strategies based on ray mutation to complement random sampling:

Adaptive Border Sampling is an algorithm to quickly find nearby triangles by sampling along the borders of triangles previously found to be visible (Section 5.3.2).

Reverse Sampling is an algorithm to sample into regions in space that are likely to be near the boundaries of visible and invisible space, i.e., the void surface (Section 5.3.3).

In Section 5.3.4, we will show how to combine the different sampling algorithms in order to obtain *guided visibility sampling*, a complete hybrid random and deterministic sampling algorithm. The algorithm is called guided because both sampling strategies are guided by visibility information in the scene (see Section 5.5.4 for a more detailed discussion).

5.3 Visibility Sampling

All rays in the scene form a 5D space. A ray x has a starting point x_p (3D) and a direction x_{dir} (2D). A typical visibility query is to give a region R in 3D space and ask what is visible along the rays leaving the region (view cell). While this defines a 5D set of rays, we only need to consider a 4D set of rays in practice; the rays starting at the boundary δR of the viewing region. Additionally, all triangles intersecting R are classified as visible.

5.3.1 Random Sample Generator

The random (or pseudo-random) sampling algorithm selects a sequence of random samples $X = x_i$ from the scene. The probability distribution for each new sample $p(x_i)$ is independent of all previous samples $x_1, ..., x_{n-1}$. The question of sampling uniformity for random sampling has been explored in the context of form-factor computation [Sber93]. We sample the position and ray direction uniformly using the following formulae:

$$u = \xi_1, \quad v = \xi_2, \quad \phi = 2\pi\xi_3, \quad \theta = \arcsin\xi_4,$$

where the ξ_i are independent Halton sequences [Nied92], and (u, v) are the normalized coordinates on a view cell face. While random sampling alone suffers from similar inefficiencies as regular sampling (see Section 5.1), it will be used to seed the more efficient strategies described next.

5.3.2 Adaptive Border Sampling

This sampling algorithm is a deterministic ray mutation strategy that covers most of the ground work to make our system successful. This strategy leaves the ray starting point x_p on the view cell fixed while covering adjacent triangles in object space, practically constructing a local visibility map [Bitt02] from the selected view cell point.

The key idea of this sampling strategy is that it adapts the sampling rate to the geometric detail of the surface (see Figure 5.5). Therefore, it is unlikely that subpixel triangles are missed, which is a problem for methods that sample objects regularly. The method performs especially well for the most frequent case of a connected mesh, but does not assume or use any connectivity information. The connected regions are discovered in the random sampling step (therefore, scenes with many small disconnected meshes like trees remain a challenge for the approach).

The algorithm proceeds as follows. If a triangle $t = (p_1, p_2, p_3)$ is hit for the first time by a sample ray $x = (x_p, x_{dir})$, we enlarge t by a small amount to obtain an enlarged polygon t' , and adaptively sample along its edges (Figure 5.5).

For each edge, we use two rays x_l and x_r, and the corresponding samples $hit(x_l)$ and $hit(x_r)$ in world space. If the rays x_l and x_r hit different triangles, we recursively subdivide the edge, up to a given threshold. At this point, we also detect depth discontinuities between the new samples and the original sample on the triangle, which is already a part of reverse sampling as described in the next section.

The actual method used for *border enlargement* deserves attention. In order not to miss any adjacent triangles, the border polygon t' should be as tight as possible. On the other hand, if it is too tight, t will be hit again due to the numerical precision of ray shooting. If the enlargement were done in object space, this would happen for near edge-on or very distant triangles. We therefore enlarge t in ray space by rotating rays to the vertices of t to their new positions on t' by a small angle. This is more robust because it depends neither on the distance of the triangle nor on its orientation, but only on the numerical precision of the ray representation.

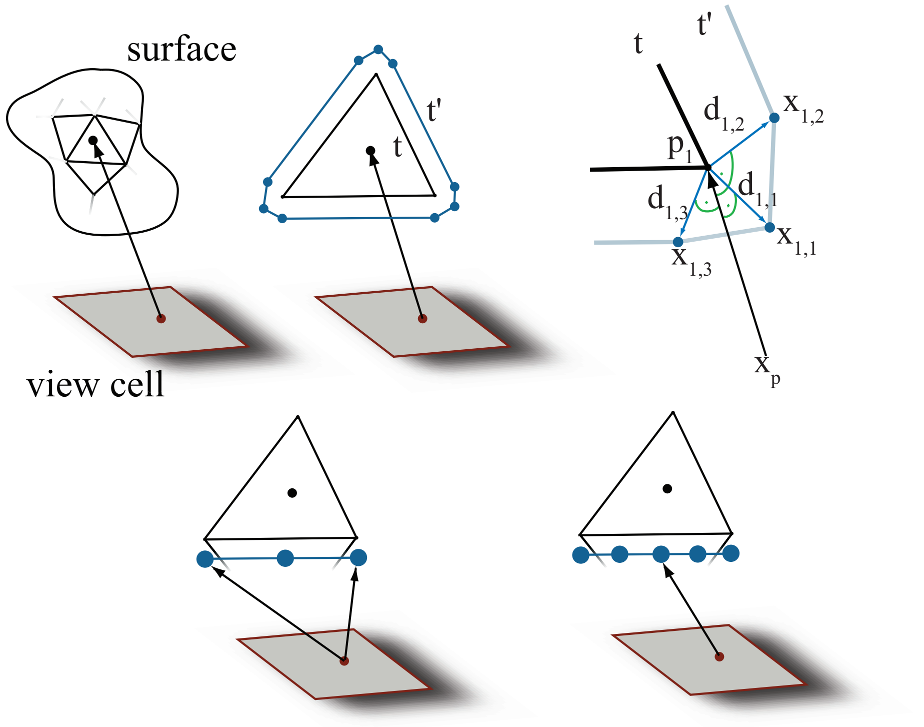


Figure 5.5: Adaptive border sampling: Top: If we hit a new surface, we sample nearby points on the border polygon t'. Bottom: Adaptive subdivision of an edge of t'.

In practice, this means that for each vertex, the new vertices are put on a plane perpendicular to the ray.

The shape of t' is chosen so that the ray space distance to t is fairly constant. This is not possible with only 3 vertices, since sliver triangles would lead to singularities. We therefore chose t' to be a polygon of 9 vertices. For each vertex p_i of t, three vertices $p_{i,j}$ on t' are generated. Two vertices are generated each on a vector $d_{i,j}$ perpendicular to the ray and to one of the adjacent edges, respectively. The third is the midpoint of the other two, pushed away from t along the angle bisector $d_{i,i}$:

$$\begin{aligned}
d_{i,i+1} &= N((p_i - x_p) \times (p_{i+1} - p_i)) \\
d_{i,i-1} &= N((p_i - x_p) \times (p_i - p_{i-1})) \\
d_{i,i} &= N(d_{i,i-1} + d_{i,i+1}) \quad \text{if} \quad d_{i,i-1} \cdot d_{i,i+1} > 0, \quad \text{else:} \\
& N((p_i - x_p) \times d_{i,i-1} + d_{i,i+1} \times (p_i - x_p)) \\
p_{i,j} &= p_i + \varepsilon \cdot |p_i - x_p| \cdot d_{i,j}
\end{aligned}$$

where $N(v)$ is the vector normalization operator. $d_{i,i}$ is chosen to be numerically

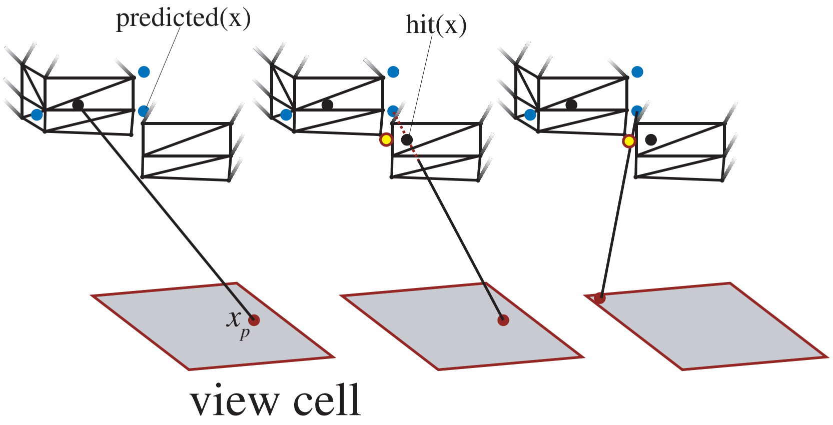


Figure 5.6: Reverse Sampling: Left: initial hit on triangle t. Middle: the new ray to $predicted(x)$ is blocked by a much closer triangle t'. Right: Reverse sampling mutates the starting point on the view cell so that the ray passes through p_{new} (yellow) and reaches $predicted(x)$.

robust. For backfacing triangles, the $d_{i,j}$ need to be inverted.

Adaptive border sampling efficiently explores connected visible areas of the input model from a single viewpoint along a 1D curve in ray space (see Section 5.5.4). However, it cannot penetrate into gaps visible only from other portions of ray space. This is handled by reverse sampling.

5.3.3 Reverse Sampling

This algorithm is a deterministic mutation strategy that allows penetrating into as yet uncovered regions of space. Note that this cannot be done perfectly: finding the actual void volume is equivalent to the original visibility problem. However, the adaptive sampling process gives good candidate locations for further sampling rays, namely at discontinuity locations. This strategy works by changing the starting point of the ray instead of its direction.

A discontinuity is detected during the adaptive sampling of an edge by comparing the distance of the ray origin to the actual hitpoint $|hit(x) - x_p|$ with the distance to a "predicted" hitpoint $|predicted(x) - x_p|$. The predicted hitpoint is just the intersection of the ray x with the plane of the original triangle t. If the new hitpoint is considerably (Δ) closer, i.e.,

$$|predicted(x) - x_p| - |hit(x) - x_p| > \Delta,$$

the ray is obviously occluded by a closer triangle. Note that we do not check the reverse case (jump from closer to farther triangle) as this will be detected when doing adaptive border sampling for the farther triangle. We calculate a mutated ray from a different view cell position to the predicted hitpoint so that it passes by the occluding triangle. For this, the plane $p = (x_p, hit(x), hit(x_{old}))$ is intersected with the newly found triangle (x_{old} is the previous ray from which x was generated). On the intersecting line, we select a point p_{new} which lies just outside of the new triangle. The mutated ray is now constructed with $x_{new,dir} = predicted(x) - p_{new}$ as direction vector, and $x_{new,p} = intersect(viewcell, line(p_{new}, predicted(x))$ as origin (see Figure 5.6). If the new ray is not contained in the ray space Ω (i.e., it does not intersect the view cell), however, it is discarded.

The new ray x_{new} is now treated as independent ray, and the triangle it intersects will be added for adaptive border sampling like any other triangle, but this time with the new view cell origin.

For the 2D example in Figure 5.3, reverse sampling corresponds to a horizontal movement in ray space.

5.3.4 Combining the Different Sampling Algorithms

The sampling strategies presented so far can be combined into an extremely efficient guided visibility sampling algorithm. Its two main components are a sample generator for exploring the ray space with independent random samples, and a sampling queue for propagating the ray using adaptive border sampling and reverse sampling. The algorithm is described by the following pseudocode:

```
main()
  while not finished
   ($x_p$,$x_{dir}$) = generate_random_ray()
   handle_ray(x)
   while not queue.empty()
     adaptive_border_sampling(
       queue.dequeue())

handle_ray(x)
  if v(x) notin PVS
    PVS += v(x)
    queue += x

adaptive_border_sampling(x)
  t' = enlarge(v(x), $\varepsilon$)
  for each p in t'
    handle_ray(($x_p$, p-$x_p$))
  for each edge ($p_l$, $p_r$) in t'
    subdiv_edge($p_l$, $p_r$)

subdiv_edge($p_l$, $p_r$)
  x = ($x_p$, $p_l$-$x_p$)
  y = ($x_p$, $p_r$-$x_p$)
  check_discontinuity(x)
  check_discontinuity(y)

  if v(x) = v(y) or |hit(x)-hit(y)| < $\varepsilon$
    return
  else
    p = ($p_l$ + $p_r$)/2
    handle_ray(($x_p$, p-$x_p$))
    subdiv_edge($p_l$, p)
    subdiv_edge(p, $p_r$)

check_discontinuity(x)
  if |predicted_hit(x) - $x_p$)| -
      |hit(x) - $x_p$| > thresh)
    $x_{new}$ = reverse_sampling(x)
    if start($x_{new}$) in view cell
      handle_ray($x_{new}$)
```

Scene	triangles	size	vc
CANYON	2,242,504	10x5x3 km	140x72 m
CITY	5,646,041	320x312x9 m	15x2.2 m
PPLANT	12,748,510	200x61x81 m	2x1.3 m
CUBES	24,000	100x100x100 m	1.5x1.5 m

Table 5.1: Statistics for all scenes. vc denotes the average size of view cells used in the model.

5.3.5 Termination criteria

Depending on the application requirements, there are several options regarding when to stop casting rays for a view cell: a) a fixed criterion, allocating a number of rays or an amount of time for the computation of each view cell, or b) an adaptive criterion, terminating if the number of newly found triangles per a certain number of samples falls below a threshold, or most preferably, c) a combination of both. A typical example for such a criterion is: stop the iteration when not more than 50 new triangles are found for 1M rays, or when a total of 10M rays has been shot, whichever comes first.

5.4 Results

5.4.1 Overview

To compare the efficiency of our algorithm to previous work, we use the following algorithms: GVS, our guided visibility sampling algorithm with adaptive border sampling (ABS) and reverse sampling (RS); and RAND, random sampling (in GVS, a value of epsilon of 5e-5 was used for enlarging triangles). We have dedicated separate subsections to the comparisons with NIR, the main other existing visibility sampling method published by Nirenstein and Blake [Nire04] (mainly because this algorithm has a slightly different goal than GVS); and EXACT, Bittner's [Bitt03] exact visibility algorithm.

The test scenes selected are (see Figure 5.7 and Table 5.1): PPLANT, the complete UNC Power Plant model; CITY, a city model of the ancient city of Pompeii generated using the CityEngine [Müll06]; CANYON, a dataset of the Grand Canyon; and CUBES, a simple scene of random cubes. The tests were conducted on an Intel Pentium4 3.2GHz with 4GB of main memory. The graphics card for NIR was an NVIDIA GeForce 7900GTX 512MB.

For GVS and RAND, we used Intel's multi-level ray tracer (MLRTA [Resh05]), which allowed sampling rates between about 800K/s and 1200K/s, with peaks up to several million samples/s. The sampling rate depends on the scene type (not so

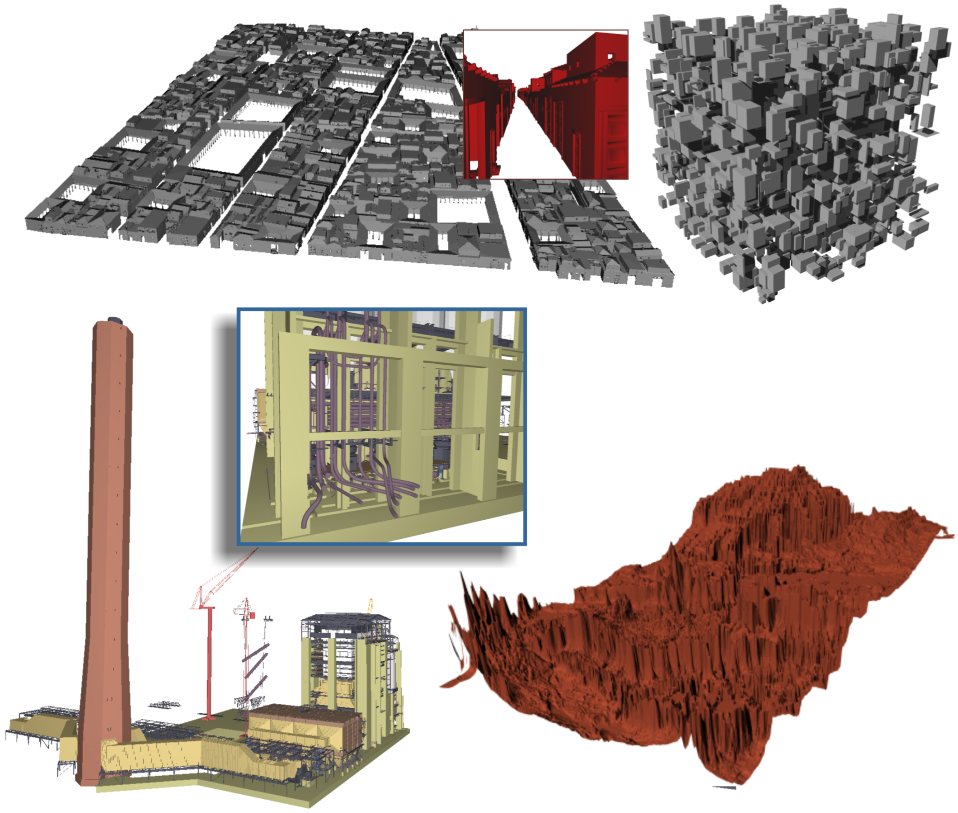


Figure 5.7: Top left: CITY. Top right: CUBES. Bottom left: PPLANT. Bottom right: CANYON. Inlays: view from a view cell.

much on the size—PPLANT had a higher sampling rate than CANYON, for example), and on the coherence of the rays (with random samples and ABS samples being faster depending again on the scene). The overhead of the sampling selection process varied between 5 and 15%, depending on the relative distribution of random, ABS and RS rays.

5.4.2 Asymptotic behavior

We first analyze the *theoretical properties* of the algorithms in terms of their sampling behavior, i.e., on a *sample-by-sample basis*, since this is the only comparison that does not depend on the individual implementation. Since we do not have an exact visibility algorithm that runs in reasonable time on larger scenes, we can only study their asymptotic behavior on a small number of view cells. Figure 5.8 provides a detailed analysis of the CANYON scene, graphing the pixel error (calculated by counting the false pixels in a large number of random renderings [Nire04]) and the number of triangles found over the number of samples for GVS and RAND.

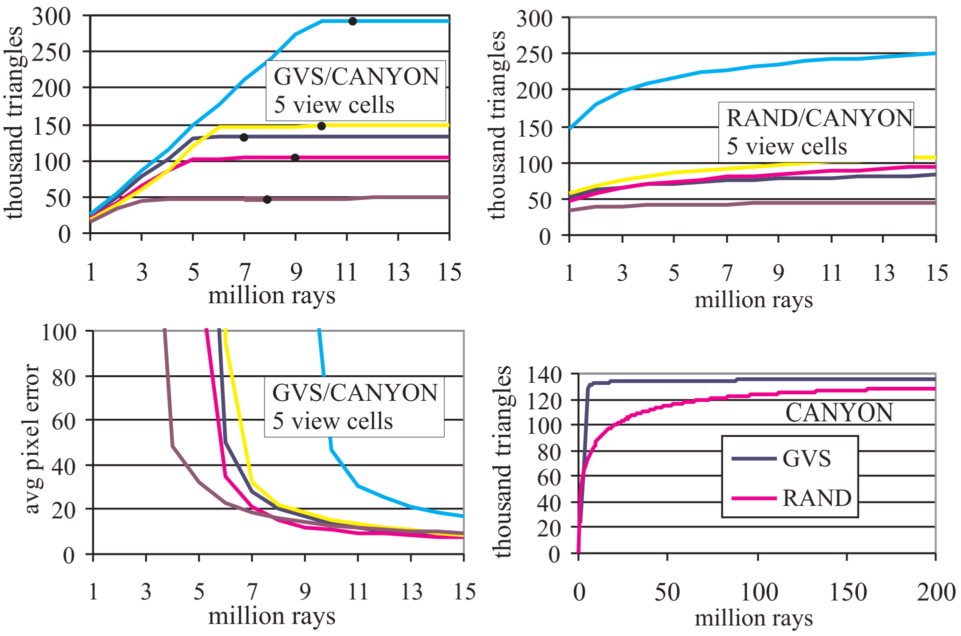


Figure 5.8: Detailed asymptotic analysis for 5 view cells for the CANYON model (see text for details). The pixel errors are measured on a 1000x1000 screen, equivalent to $10^{-4}\%$. The plots in the lower right image show the blue view cell from the other images.

The top left image shows that GVS converges linearly as long as the deterministic strategies (ABS and RS) can be used for most triangles. The black dot on each view cell curve shows when our termination criteria terminates the PVS search (we used 50 or less new triangles found per 1M samples). It can be seen that this happens in a fairly well converged state already. The graph also shows that the behavior is very similar for all view cells. The length of the linear segment only depends on the final PVS size.

The top right figure shows RAND in comparison. The convergence of RAND looks mainly logarithmic and has a very quick falloff after an initial strong phase. It is especially noteworthy that even at 15M samples, when GVS has already long converged, RAND is still 50K triangles behind GVS for most view cells. The bottom right figure analyzes this behavior on an even larger scale for the dark blue view cell from the other graphs. This figure confirms the quick convergence of GVS, and shows that even after 200M samples, RAND is still several thousand triangles behind GVS. It can be concluded that it would take RAND several orders of magnitude longer to find a PVS that GVS can find with about 7M to 8M samples.

Finally, the bottom left figure proves that the PVS size correlates strongly to average pixel error, and that the termination criterion discussed above works well in practice, bringing the average pixel error below 30 pixels on a 1000x1000 screen.

Due to the better distribution of initial samples, RAND shows lower average pixel error in the phase where GVS searches mainly deterministically. However, to reach the same pixel errors as provided by GVS in a converged state, RAND has to calculate a similar number of triangles in the PVS, leading to the same observation as before, that similar pixel error requires orders of magnitude more samples than with GVS.

5.4.3 Practical results

Next, we demonstrate that these findings generalize to a larger number of scenes, and provide a *practical analysis* including running times. Table 5.2 summarizes our findings.

	Alg.	Avg.Err.	Max.Err.	time	PVS
CANYON	GVS	35	239	7.9s	6.7%
	RAND	67	828	183s	6.3%
	NIR512	2,191	8,215	6.8s	5.8%
	NIR1024	519	2480	11.6s	6.3%
CITY	GVS	22	230	6.1s	1.1%
	RAND	70	625	69s	0.4%
	NIR512	1,292	8,655	5.4s	0.2%
	NIR1024	631	8,965	8s	0.4%
PPLANT	GVS	23	211	30s	0.8%
	RAND	69	825	129s	0.5%
	NIR512	3,225	17,169	24s	0.4%
	NIR1024	1,549	8,317	25.9s	0.6%

Table 5.2: Statistics for all scenes averaged over a number of view cells. We used a threshold of 50 or less triangles found per 1M samples to cut off computation for GVS. For RAND, we shot 150M rays for each test. Errors are number of false pixels on a 1000x1000 screen, which corresponds to $10^{-4}\%$. Results for NIR are given at 512x512 and 1024x1024 resolution. The intrinsic parameters had to be adjusted for each scene to obtain reasonable results (see the comments in Section 5.4.4). The last column shows the average size of the calculated PVS as a percentage of the whole model.

We used the same convergence criterion of 50 triangles per 1M samples for GVS, and a constant 150M rays for RAND. It can be seen that this results in very similar average and maximum errors for both algorithms. However, the running times differ by more than an order of magnitude, which reflects the good convergence behavior of GVS with respect to RAND shown above. The table also lists results for NIR, which are discussed in the following subsection. In addition to the error we also give the size of the PVS in terms of the whole model size (an

EVS was not available in reasonable time). A higher value means a more accurate solution.

5.4.4 Comparison to hardware sampling

Nirenstein and Blake [Nire04] recently published an interesting adaptive regular sampling algorithm which uses graphics hardware to adaptively sample hemi-cubes on the view cell. It is difficult to directly compare NIR and GVS. On the one hand, they are both based on the same atomic operation—taking a visibility sample. This is because sampling with graphics hardware and with a ray tracer is functionally practically equivalent due to the available sub-pixel precision (usually 12 bit) in current graphics hardware.

The time complexity, however, differs significantly between the two algorithms. The time complexity of ray casting is linear in the number of rays and, due to spatial data structures, logarithmic in the number of objects. In practice, we have also observed a strong dependence on the type of the scene and the implementation of the ray tracer, which makes general predictions on the scalability with respect to scene size very hard.

For graphics hardware, the basic operation is an item-buffer render. Depending on whether a particular view is mostly fill or geometry limited, the resolution of this item buffer has more or less impact on the rendering time. Our implementation of NIR rendered models from multiple vertex buffers stored directly in video memory, which provided triangle throughput near the theoretical maximum on the card we used (between 130 and 190M triangles/s, depending on how many vertices were shared in the model—note that some vertices had to be duplicated to allow item buffer rendering). Only on the CANYON model did we observe a fill rate limitation (9 vs. 12 hemicubes/s for 512 vs. 1024 resolutions), whereas CITY and PPLANT were geometry limited (7 and 2 hemicubes/s).

Efficient acceleration algorithms exist for both architectures, if a certain amount of preprocessing is tolerated. Of particular importance for visibility processing is that the complexity of scenes that can be handled by ray tracing is limited only by the available storage space, as ray casters can work efficiently out of core (e.g., Wald et al. [Wald04] have demonstrated that a 350 million polygon model can be ray cast at 2-3 frames per second). Furthermore, it should be pointed out that rasterization benefits from hardware acceleration, whereas ray tracing is still run in software. Recent advances in hardware for ray tracing [Woop05] promise a huge potential for improving the speed of sampling-based algorithms like GVS even further, once this technology becomes more commonplace.

However, the main difference between the algorithms is the principal goal. NIR aims to increase rendering speed by aggressively culling more objects than are actually occluded, the rationale being that large gains in rendering speed can be obtained if errors in the final image are tolerated. Indeed, NIR consistently underestimates the PVS, as shown in Table 5.2 (note that even for an error threshold of 0, a significant rest-error is reported for NIR [Nire04]).

While this approach is valuable for applications like quick previewing etc., where a resolution can be fixed, and an average of, for example, 1000 false pixels is tolerable, many applications require a more accurate PVS. This is where GVS excels. The GVS algorithm aims to provide the most accurate PVS possible with a minimum number of samples. Therefore, the performance metric for GVS is not the total percentage of culled objects, but the degree to which the actual PVS can be approximated. Our results show that GVS, using a limited number of samples, consistently finds the largest PVS, resulting in average pixel errors below 0.005%. This is important for any visualization application that relies on visibility preprocessing (especially if antialiasing is used or the output resolution is not fixed in advance), but also for a number of other applications where reliable (and practically exact) visibility is required, e.g., computational geometry, GI, and robotics.

It should be noted for the results in Table 5.2 that NIR results are derived through a PVS subdivision threshold, which works differently from the method used in GVS and RAND and can therefore not be compared directly. We found that this threshold was very sensitive to the type of the scene and had to be tuned so as not to lead to excessive subdivision or too early termination in each scene separately (for example, in once case the error for the 1024 resolution was significantly worse than for 512, due to premature termination). The reason for NIR's inability to pick up the complete PVS lies both in the regular sampling strategy, which forces a very fine subdivision on the view cell in order to pick up sub-pixel triangles, and in the thresholding for the adaptive subdivision, which can prematurely terminate the subdivision.

5.4.5 Comparison to exact visibility

We compared our algorithm to EXACT on the CUBES scene, from a view cell of about 1.5x1.5m. EXACT took 19s on a PIV 1.7GHz PC to find 3,743 visible triangles. To find the same number of triangles, GVS required about 3s. For GVS, a screenspace error of 0.001% was already reported after 2s. More interesting, however, is the fact that both GVS and RAND found significantly more visible triangles than EXACT if given enough samples. For example, 3,850 triangles were found after only 15s by GVS. Note that EXACT was used on an "as is" basis—better results could certainly be achieved by tuning numerical thresholds intrinsic to the method. This shows clearly that the accuracy of visibility algorithms, even exact ones, is ultimately limited by numerical issues.

5.5 Related Work, Discussion and Applications

A large volume of research has been devoted to visibility problems due to their importance in computer graphics, computer vision, robotics and other fields. This section compares various aspects of the proposed visibility sampling algorithm to a wider class of from-region visibility algorithms. For a general overview, we can recommend excellent surveys of visibility problems and algorithms [Dura99, Cohe03].

From-region visibility algorithms are usually classified as exact (potentially visible set PVS = exact visible set EVS), conservative (PVS $\supseteq$ EVS), aggressive (PVS $\subseteq$ EVS), or approximate (PVS $\sim$ EVS).

5.5.1 Exact Visibility

Exact solutions to compute visibility from a region in space have been rare [Dugu02, Dura99], but recently, two algorithms have been published [Nire02, Bitt03] and further improved upon [Haum05, Mora05] that are both exact and work for general scenes. While exact algorithms have been the holy grail of the visibility community for a long time, these two algorithms show that the complexity inherent in the visibility problem may be an obstacle to make exact visibility widely applicable. The high running times and high complexity of implementation are critical, and numerical robustness issues can actually make the solution as approximate as a sampling-based strategy (see [Bitt03]). We believe that sampling-based methods and exact methods complement each other, as they have different strengths and weaknesses.

5.5.2 Conservative Visibility

Several authors stress the importance of conservative visibility computations, i.e., never underestimating the visible set. Since this problem is almost as hard as the exact visibility problem, practically all published conservative from-region algorithms simplify the problem by imposing certain restrictions on the scene. Typical restrictions are the limitation to 2.5D visibility [Wonk00, Bitt01, Kolt01], architectural scenes [Aire90, Tell91], the restriction to volumetric occluders [Scha00], or the restriction to larger occluders close to the view cell [Leyv03, Dura00]—this last restriction is implied by the nature of the data structures used to store visibility information. While it can be argued that larger occluders can be synthesized from smaller ones [Andu00], this is not possible in general. The guarantee to include all visible geometry in the PVS may be important for some applications, but ultimately, sampling-based methods can be much more successful:

1. As opposed to the published conservative algorithms, they do not make any

assumptions about the scene, allowing them to handle a much larger variety of scenes.

2. Due to their ease of implementation and robustness, non-conservative algorithms are more practical for commercial products such as computer games [Aila04b], and are already used in this context.

3. Numerical issues often make conservative algorithms non-conservative in practice.

5.5.3 Aggressive Visibility

Since visibility is such a fundamental problem, general, robust and practical tools are important to complement the specialized algorithms discussed before. These tools are almost universally based on sampling. The two most popular solutions are to randomly select a large number of rays to sample visibility [Scha00, Aire90, Shad98], or to first sample the boundary of the view cell with points and then sample visibility from each of these points [Levo96, Stue99]. In the context of view planning for laser range scanners, sampling algorithms exist that store the void surface or the void volume to compute the next-best view [Pito99]. A similar algorithm was also used for the generation of textured depth meshes [Wils03]. Another option is to shoot rays from the scene triangles towards the view cell [Gots99], which leads to oversampling of ray space for most scenes.

Nirenstein and Blake [Nire04] were the first to realize the full potential of sampling for visibility computation. They proposed a new approach which uses graphics hardware for sampling. As discussed in Section 5.4.4, this algorithm aims to reduce the rendering time by culling even visible triangles as long as this does not result in significant rendering error. This is opposed to our algorithm, which always tries to find the best possible approximation of the exact visible set.

5.5.4 Algorithm Analysis

Ray space analysis. In the introduction in Figure 5.3, we have argued that it is desirable not to sample the ray space regularly. The right image in this figure shows that only an approximately 1D subspace of rays needs to be considered in this simple 2D example. Our new algorithm samples ray space more intelligently: random sampling places initial seed points in ray space to stochastically search for regions in ray space that have not been explored yet. To continue the example for 2D as in the figure, adaptive border sampling corresponds to a vertical expansion in 2D ray space (since the viewpoint remains fixed) which only proceeds into yet unexplored areas. A particular advantage of the adaptive border sampling method is that the sampling rate is adapted to the geometric complexity of the visible surfaces. Reverse sampling, on the other hand, is a movement in the horizontal direction (since the hitpoint remains fixed) in cases where these movements promise to lead to not yet explored regions.

For the full 3D case, it is instructive to study our algorithm in terms of the *visibility complex* [Dura99]. The visibility complex describes a partition of the 4D ray space into 4D regions of rays that hit the same object (note that ray space is strictly 4D because we are only interested in rays starting from the view cell). The 3D boundaries of this partition are called *tangency volume* and consist of rays tangent to scene objects. Samples placed along the object borders therefore correspond to samples near the tangency volume of the object in dual space. Since we keep the viewpoint (2 degrees of freedom) fixed during the deterministic ABS exploration phase, we need to sample a 1D set only. Without ABS, we would ignore the tangency volumes and have to sample the whole 2D subset of ray space defined by the chosen viewpoint.

Reverse sampling, on the other hand, looks for lines tangent to two scene edges. In ray space, these lines are near intersections of two tangency volumes. These intersections are called bitangents and are only 2D. For reverse sampling, the viewpoint is allowed to move along a plane (1D), so in total RS also samples a 1D set. The combined ABS and RS strategies therefore correspond to explorations of the 4D ray space along those 1D curves that are most likely to reveal new objects. This explains the high efficiency of the GVS algorithm.

Another useful interpretation of the ABS sampling strategy in 3D is based on the visibility map [Bitt02]. The visibility map is a structure that contains all visible line segments in a given view. These segments can be characterized mainly as flat and corner (interior edges of a mesh), or shadow (depth discontinuities). The ABS sampling strategy places samples at all edges of the visibility map (without explicitly constructing it). Samples on interior edges of a mesh serve to find connected sets of a mesh (trivially adjacent regions in the visibility complex). Samples at the shadow edges serve to discover depth discontinuities, where objects are partly occluded by other objects. Shadow edges are where the RS sampling strategy is used to refine the sampling (by finding the bitangents in the visibility complex).

Accuracy. The term conservative (or even exact) visibility is actually quite misleading. Most algorithms, though conservative in theory, are not conservative in practice due to numerical robustness problems. This is especially true for algorithms relying on graphics hardware. Furthermore, complex algorithms are prone to implementation problems. Due to the much improved sampling efficiency, the magnitude of error introduced by our algorithm is comparable to that of other error sources. Such errors are usually tolerated for conservative algorithms (see Section 5.4). Other algorithms that are often used in conjunction with visibility processing, like level-of-detail algorithms or shadow mapping, are an additional source of errors.

Scene complexity. One distinguishing feature of our sampling-based algorithm is that it can handle arbitrary types of scenes with high overall and visual complexity. It does not rely on occluder synthesis, and depends mostly on the size of the visible set, not on the total scene complexity.

5.5.5 Limitations and Future Work

Although guided visibility sampling generally finds the major part of the PVS very quickly, the fact that it is stochastic on the one hand and guided by the visibility in the scene on the other hand makes the final accuracy dependent on the structure of the scene. Therefore, we cannot give any hard guarantees for the pixel error of the calculated PVS. Also, the ability to explore connected ray space subsets in the far distance is limited by the numerical precision of the ray direction vector. For ABS, this means that triangles that have a solid angle of less than double precision accuracy when seen from the ray origin will most likely be missed.

The worst case of scene complexity is in scenes that consist of a large set of small disconnected triangles, such as forest scenes or synthetic scenes of random triangles. The visibility of such scenes is so complex that even sampling-based solutions will either have high error or take a long time to compute. Still, it is important to point out that sampling-based algorithms are the only ones that are able to even process these scenes.

In this respect, an avenue of future work is to incorporate geometric LOD into the sampling framework, similar to the vLOD system proposed by Chhugani et al. [Chhu05]. Geometric LODs could potentially increase the speed of the ray tracer, and make intersection computations more robust because small triangles in the distance get replaced by larger ones. However, robust geometric LOD is not available for all scenes, and integrating LODs into ray tracers is a current topic of research. Furthermore, the error metric used to create the LODs impacts the accuracy of the visibility algorithm and therefore the usable output resolutions.

5.5.6 Applications

One important strength of sampling-based methods is their ease of application. We will discuss a number of application scenarios for our algorithm.

Visibility preprocessing for real-time rendering and games. This is the scenario already described in the overview, and one of the most important applications for GVS. For example, the scenes of current computer games are becoming increasingly general, so that special purpose algorithms (cells and portals, and 2.5D solutions) cannot be used anymore, while exact algorithms are difficult to implement and error-prone. GVS can be used in all stages of game development: During level design, the number of rays can be limited so that a coarse solution can be provided almost instantaneously. For the final production, the PVS can be calculated with high accuracy. It is very important to create a PVS that is as close to the EVS as possible and not dependent on a particular output resolution, since the resolution the application will be run at is not known in advance. In addition, antialiasing methods (supersampling and multisampling) use information from subpixel triangles, so that the virtual resolution is even higher. Note that although scenes in computer games are inherently dynamic, the major part of the scene is still static, so huge gains in rendering speeds can be obtained. Furthermore, GVS works on arbitrary polyhedral view cells, so that the view space can be chosen freely.

Online and networked visibility. As shown in the results, a reasonable approximation to the EVS with low pixel error can be found in a second or less. Therefore, GVS can be used for online visibility culling by running it on a separate processor or over the network, as described in the Instant Visibility system [Wonk01]. In this case, transmitting the PVS on a per-object basis will improve results because it suffices for one triangle of an object to be found by GVS in order to classify the whole object as visible. Furthermore, a small modification to GVS makes the algorithm better suitable to progressive evaluation: instead of interleaving ABS and random samples from the beginning, create a certain number (e.g., 1M) of random samples in a startup phase, and only then use those to seed the ABS rays. This will give a better distribution of samples in the initial phase of the algorithm, since ABS systematically “flood fills” the PVS around its seed point, and it takes some time until all image regions have been reached.

Impostor generation. In many scenes, visibility culling is not sufficient to guarantee a high frame rate everywhere in the model. Therefore, image-based methods can be used to replace complex scene parts by so-called impostors. However, since impostors trade rendering speed against memory consumption, it is important to find the exact visible parts of the scene to avoid wasting impostor memory on invisible geometry [Jesc05]. GVS is ideally suited for this purpose since it provides accurate per-triangle visibility information, so that only those object parts that are actually visible need to be stored in an impostor.

Visibility as decision basis. Many practical applications require accurate visibility information as part of a decision making process. Examples include visibility analysis in urban planning (does the new skyscraper impact old town?), military applications (line of sight culling, tactical battlefield management [McDe87]), telecommunications (visibility of emitters), robotics and many more. GVS is advantageous for these problems because it is general purpose and does not have any parameters to tweak, and does not depend on any special properties of the scene.

5.6 Conclusion

We have presented a visibility sampling algorithm to compute a full 3D visibility solution from a region in space. The proposed algorithm improves the efficiency of previous sampling strategies by over two orders of magnitude, thereby allowing visibility solutions with negligible error to be computed in reasonable time. The proposed algorithm works on arbitrary so-called polygon soups and does not require any memory beyond that used by the ray caster. Due to the new sampling strategies employed in the algorithm, its accuracy is competitive even with exact and conservative approaches, while it is also extremely simple to implement.

We have provided evidence that Guided Visibility Sampling closes an important gap in visibility research. It combines the speed and ease of implementation of sampling-based and special-purpose conservative algorithms with most of the accuracy of exact solutions. Thus, GVS can be used as a general purpose visibility tool.

Acknowledgements

We thank Jiri Bittner for fruitful discussions. This research was also supported by the EU in the scope of the GameTools project (IST-2-004363), and by the NGA, grant no. HM1582-05-1-2004.

Chapter 6

Instant Points

Published as:

- Michael Wimmer and Claus Scheiblauer:
  Instant Points
  In *Proceedings Symposium on Point-Based Graphics 2006*, pages 129–136.
  Eurographics, Eurographics Association, July 2006. ISBN 3-90567-332-0.

Instant Points

Michael Wimmer and Claus Scheiblauer

Abstract

We present an algorithm to display enormous unprocessed point clouds at interactive rates without requiring long postprocessing. The novelty here is that we do not make any assumptions about sampling density or availability of normal vectors for the points. This is very important because such information is available only after lengthy postprocessing of scanned datasets, whereas users want to interact with the dataset immediately. Instant Points is an out-of-core algorithm that makes use of nested octrees and an enhanced version of sequential point trees.

6.1 Introduction

Point-based rendering has gained a lot of popularity due to the availability of 3D range scanners. Recent long-range laser scanners capture data up to 1000m, giving wildly varying sampling densities (Fig. 6.1). Practically all current point rendering approaches rely on assumptions about sampling densities, or the availability of normal vectors for each point. In fact, most papers present models that have been gained by point sampling a geometric mesh.

Unfortunately, post-processing a scanned point cloud to obtain a mesh can take several person months. This is because especially data acquired in the outdoors is not amenable to automatic postprocessing methods. Such data is characterized by holes in almost any surface, varying sampling densities, and a mixture of surface and non-surface structures (such as leaves), which is a problem for meshing approaches. There are several efficient methods to estimate the normal vectors required for point rendering techniques directly from the point cloud [Dey05]. However, normal estimation assumes that the points represent sufficiently dense samples of an underlying surface, which cannot be assumed for long range scans, where the data is simply too sparse in many regions. Thus, lengthy manual post-processing is inevitable.

On the other hand, there is a real need for users to explore and interact with the scanned data instantly. This is a factor of both costs and opportunity: paying several months for a qualified engineer to postprocess a scanned model can be prohibitive for most potential users (e.g., archaeologists, architects et.) of 3D scanners. Furthermore, at the time the post-processed model is available, the original research question might have already been solved in another way, rendering the model unnecessary.

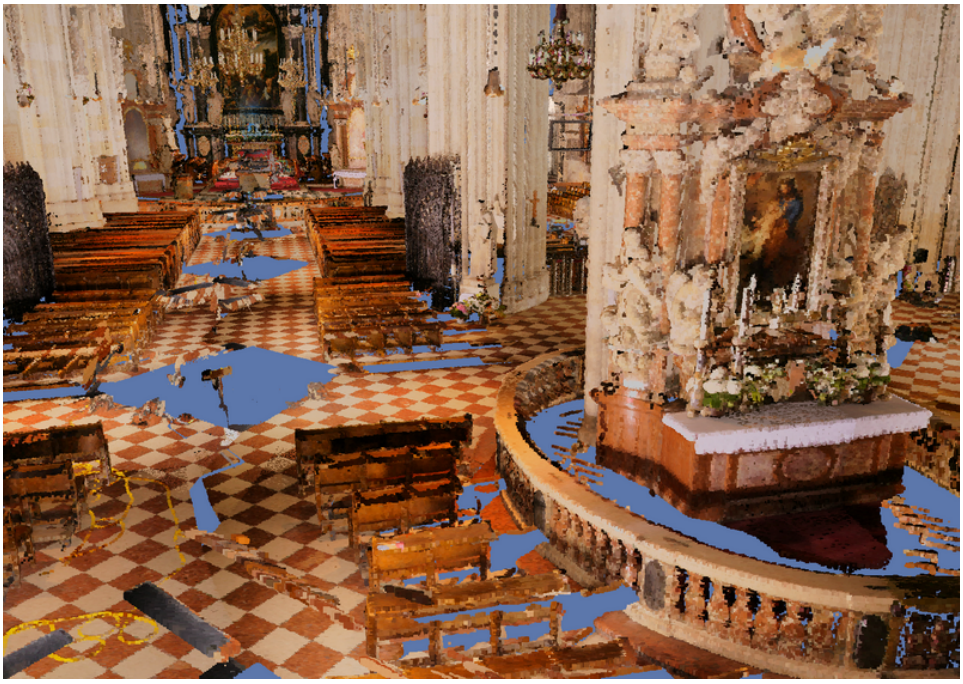


Figure 6.1: Example screenshot of an unprocessed point cloud consisting of 262M points.

This is where *Instant Points* come in: We present the first point-rendering algorithm that does not require postprocessing of point clouds and which at the same times renders enormous amounts of *unprocessed points* at interactive rates with negligible preprocessing. With unprocessed point clouds we mean those which have not been *interpreted* in any way (meshing, normal estimation etc.), i.e., each point is defined only by a 3D position and—if available from the scanning process—an RGB-color. The point cloud is converted into an efficient out-of-core structure which is based on a combination of nested octrees and memory optimized sequential point trees, optimally exploiting current graphics hardware.

Unprocessed point clouds evidently cannot match postprocessed models in image quality. The main contribution of this paper is to show how to trade this reduced image quality against significantly increased visualization speed and improved memory requirements. There is a definite need for unprocessed point rendering, starting from quick onsite visualizations during the scan campaign (which can help in scan planning, scan verification etc.); visualization systems where the general “feel” of a location is more important than the exactness of every minute detail; experimental systems for archaeologists, architects and regional planners, for which such unprocessed point rendering would make range scanning viable at all; visualization systems where access to the original point cloud is needed; and many others.

6.2 Previous Work

Rendering point clouds has recently become a popular topic. However, the huge majority of research has concentrated on rendering datasets that have been post-processed, if not even sampled from an original geometric model. These rendering approaches build on the pioneering Surfel approach [Pfis00], which basically defines a point as a small ellipsoidal disk with normal vector. The trend is towards ever higher-quality rendering methods for Surfels, as presented by Botsch et al. [Bots05], who also give a very good overview of previous high-quality point splatting methods.

However, high-quality input models are not often available, and methods dealing with unprocessed point clouds are rare. Xu and Chen were the first to realize the extreme difficulty of displaying models acquired from long range outdoor scanners. Instead of trying to postprocess the data, they propose to use non-photorealistic rendering techniques in order to change the viewers' expectations on the realism of the viewed model [Xu04]. While their system produces stunning results in many cases, it is limited to smaller datasets due to its computational complexity.

Sequential point trees (SPT) by Dachsacher et al. [Dach03] is one of the fasted algorithm for rendering small to medium point clouds because it makes good use of graphics hardware. They realize that maximum throughput can only be achieved if large batches of primitives are stored in graphics hardware buffer objects. For unprocessed point clouds, however, the representation incurs significant memory and performance overhead as will be shown in Section 6.4.

The first system that was able to render large point sampled models that do not fit in main memory was QSplat [Rusi00], which builds a hierarchy that allows per-node level-of-detail (LOD) selection. However, the selection has to be done on the CPU. A similar approach is taken by Duguet and Drettakis [Dugu04], who also use an LOD representation of a model that is processed per node. They target the special hardware of PDAs, which do not have dedicated graphics processors. The layered point cloud (LPC) [Gobb04] system uses block LODs for point sampled models and achieves rendering rates that are an order of magnitude higher compared to QSplat by making efficient use of graphics hardware. The system assumes a uniform sampling density of the input data. Our nested octrees are similar in spirit to layered point clouds, but our algorithm works on arbitrary data which is not necessarily sampled uniformly. XSplat [Paja05] is another system for out-of-core rendering of huge point clouds. Similar to our Instant Points system, XSplat is based on SPTs and uses a two-level hierarchy. The main difference is that XSplat is aimed at rendering high-quality models, whereas Instant Points offers significant optimizations for unprocessed point clouds. Out-of-core methods have also been extensively studied for triangle rendering, where two-level hierarchies are used as well [Yoon05], but in addition, connectivity has to be taken into account.

Some systems approximate (parts of) the point cloud with textured [Wahl05] or normal-mapped [BOUB05] polygons without extracting the topology. While these approaches can provide extremely fast frame rates for large point clouds, we opted for a system where the original scanned points can be shown at the finest level. A promising alternative to out-of-core rendering is to compress the point cloud so that it fits into graphics card memory and can be decoded directly on the GPU [Krüg05].

6.3 The Instant Points Rendering System

Unprocessed point clouds do not contain any connectivity or density information, making it difficult to devise a suitable rendering representation. The main idea of the Instant Points system is to circumvent this fact by interpreting point samples through the effect they have on rendering in a frame buffer. This means that if more than one point projects to one pixel, only one actual representative point is needed to fill that pixel. More concretely, we deal with the problem of viewing a dense point cloud, which can be solved through subsampling the point cloud. Without additional information, we can not solve the interpolation problem that arises when point clouds are viewed from too near a distance. In this case, we give the user a choice of using a fixed world-space extent for point samples, or a multiple of the sample distance which is derived from the depth of the hierarchy.

The Instant Points system consists of two main elements:

- Memory optimized sequential point trees (MOSPT), a version of SPTs improved for unprocessed point clouds.
- Nested octrees, a structure that allows out-of-core rendering, and contains MOSPTs as elements.

MOSPTs can be used alone for smaller point clouds to remove the 125% of memory overhead caused on average by SPTs for unprocessed point clouds, and increase the rendering speed by not having to render the 50% of additional interior nodes in the hierarchy, and to simplify the rendering process. However, we will mainly use them as parts of the nested octrees described later.

6.4 Memory Optimized Sequential Point Trees

6.4.1 Sequential Point Trees Revisited

Sequential Point Trees (SPTs) [Dach03] are a hierarchical point representation that allows rendering through a *sequential* processing by the GPU, while the CPU is available for other tasks. Each node of the hierarchy is associated with an error e. The recursive traversal checks whether the projected error $e/r < \varepsilon$, where r is the view distance. This can be simplified by storing a minimum distance $r_{min} = e/\varepsilon$ with the node, so that the test becomes $r > r_{min}$. In addition, each node also stores a distance r_{max}, in the simplest case the r_{min} of the node's ancestor. This allows a non-recursive test for each node by checking whether $r \in [r_{min}, r_{max}]$. This test can easily be carried out by a vertex program on the GPU. Additionally, by sorting the vertex list by r_{max} and calculating a lower bound $min(r)$ via the bounding sphere of the object, the GPU need only process a prefix of the vertex list with $r_{max} < min(r)$. Processing only this prefix is the main reason why SPTs are so efficient.

The error e in the original SPT algorithm assumes that the points are actually splats (or surfels) with a splat size d and a normal vector n. Inner nodes have splat sizes that encompass the child nodes. This allows levels of detail where larger splats approximate flat surface areas, and smaller splats are used in curved areas.

6.4.2 Screen Splat Error Metric

A point in an unprocessed point cloud does not represent a surfel (i.e., a splat with normal and radius), but a point sample that is rasterized by graphics hardware as a screen-space splat. For a given point hierarchy, a node should be rendered when further recursive traversal would not change the points that are rasterized. This is the case if the projected size of the node is smaller than a pixel.

In order to achieve this semantics for SPTs, we define the error e of a node as the diameter d of the node geometry. Therefore, $r_{min} = d/\varepsilon$, where ε needs to be adjusted depending on the camera parameters and the desired splat size. This allows unprocessed point clouds to be rendered using SPTs.

6.4.3 SPT overhead

Each hierarchical data structure has a certain overhead depending on the average branching factor α. The *memory overhead* M, i.e., the number of interior nodes relative to the leaf nodes, of a hierarchical structure can be calculated as [Dugu04]:

$$M \sim \frac{1}{\alpha - 1}.$$

For an octree storing 2D surface data, $\alpha = 4$, resulting in an overhead of 33%. However, our experiments have shown that data coming from range scanners have lower branching factors. For several different datasets, the observed branching factor was $\alpha = 3$, resulting in an overhead of $M = 50\%$.

Note that this causes not only an increase in the memory required to store an SPT in graphics hardware, but it also directly reduces rendering performance, since the interior nodes are additional points that have to be processed by the vertex processor. Especially for viewpoints near the model, where $r_{max} > min(r)$ for all

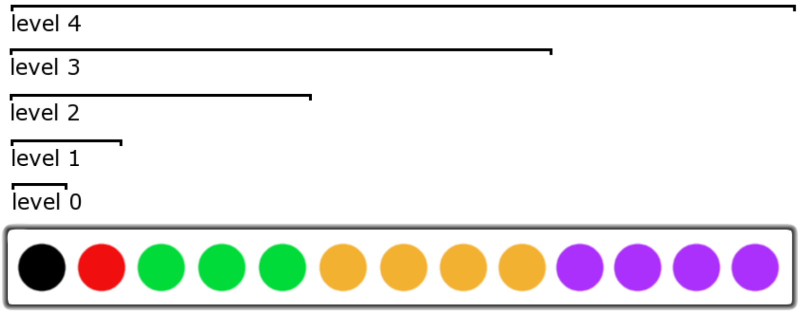


Figure 6.2: Linearized MOSPT hierarchy. Higher-level nodes form part of lower levels of the hierarchy.

nodes and the whole vertex list has to be processed by the GPU, rendering time is increased by $M = 50\%$ on average in comparison to rendering the original points only.

Furthermore, nodes need to store r_{min} and r_{max}, taking 4 bytes (1 float) each. Assuming that for scanned datasets, only a point position (12 bytes) and a color (4 bytes) is stored normally, this would result in a *structural overhead* $S = 50\%$. In total, for unprocessed point clouds, the combined overhead M and S caused by SPTs is 125% on average, depending on the branching factor of the specific model.

6.4.4 Memory optimization

In order to overcome the significant memory and rendering overhead caused by SPTs for unprocessed point clouds, we make use of the following observation, which follows directly from the definition of the screen splat error metric:

Any child node of an interior node that is selected *for rendering will lead to the same pixel on screen being rasterized as the interior node.*

Therefore, instead of creating a new point to represent an interior node, we use an *existing child node* as a representative point for the ancestor. This is similar in spirit to vertex clustering algorithms [Lueb97, Ross93], where all vertices in a hierarchy node are replaced by one representative vertex taken from the input vertices. For an uncolored unprocessed point cloud, the choice of representative is completely arbitrary. However, most interesting point clouds have color information. In this case, we select the color that has the smallest color distance to the average color of the child nodes.

The resulting hierarchy can be stored in an extremely efficient way as an SPT. The nodes are still sorted by r_{max}, but instead of storing all nodes for each level in the vertex list, we omit for each level those nodes that were chosen as representative point in the previous (upper) level, and are therefore already stored in the vertex

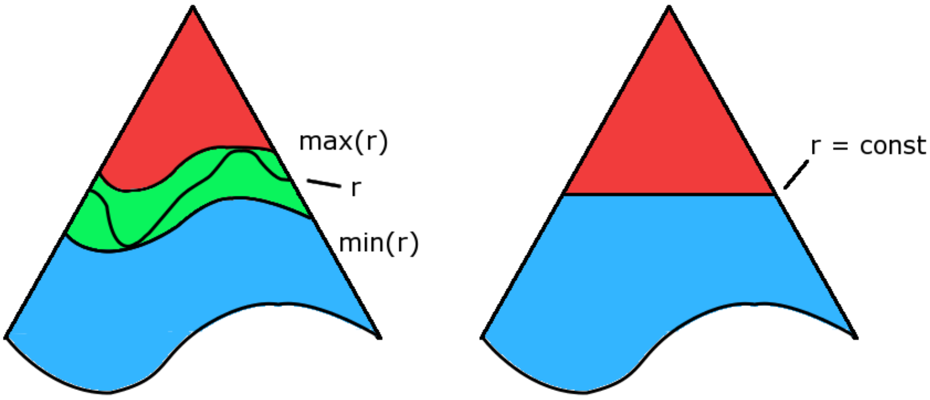


Figure 6.3: Left: In an SPT, $r = const$ selects different levels of the hierarchy. Furthermore, nodes above and below r need to be culled. Right: In an MOSPT, $r = const$ selects exactly one level in a hierarchy from an MOSPT due to the screen splat error metric. All nodes above $r = min(r)$ are rendered.

list. This new, *memory optimized SPT* (MOSPT), does not require more memory than the original point cloud. In fact, the hierarchical SPT structure is achieved by cleverly reordering the original point cloud (see Figure 6.2).

6.4.5 Rendering MOSPTs

The screen splat error metric leads to several simplifications in the rendering of MOSPTs in comparison to SPTs.

- The screen splat error metric d is constant for each level of the MOSPT hierarchy, since all nodes at the same hierarchy level have the same diameter. Therefore, a cut of the hierarchy with $r = const$ gives exactly one level of the hierarchy, instead of multiple levels as in SPTs (see Figure 6.3).
- SPTs evaluate the view distance r for each node in a vertex program to allow culling nodes depending on their actual distance. For example, nodes further away could be rendered with a coarser level of detail. While this would also be possible for MOSPTs, there is no benefit in doing so, since culling a vertex in the vertex shader does not reduce its rendering time. Therefore, we just let the GPU process all nodes with $r_{max} < min(r)$ (see Figure 6.3).
- There is no need to cull any node with $r_{min} > r$ (i.e., interior nodes), since these nodes also form part of the more refined nodes and should therefore be rendered in any case.
- Again due to the constant d for each MOSPT level, it is not necessary to save r_{max} for each node in the hierarchy on the CPU. Instead, it is sufficient

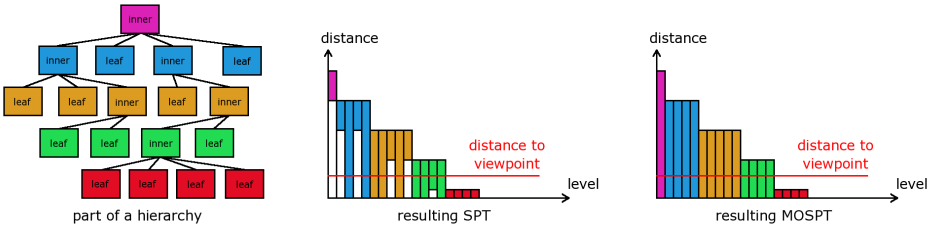


Figure 6.4: A hierarchy (left) created for an SPT (middle) consists of original and additionally created points. The SPT contains inner and leaf nodes mixed in one level of the hierarchy. A hierarchy for an MOSPT (right) consists only of original points.

> to store in an *index array* for each hierarchy level the number of points contained in this and all preceding hierarchy levels. The r_{max} values for each hierarchy level can easily be recomputed on the fly.

These consequences of using MOSPTs lead to a very simple rendering algorithm:

1. For each frame, the CPU calculates $min(r)$ from the bounding sphere of the model as in SPTs.
2. This is successively compared to $r_{max_i} = d_i/\varepsilon$ for each hierarchy level i until $r_{max_i} > min(r)$.
3. The number p of points to render is looked up in the index array at position i.
4. The GPU is instructed to render the first p points of the MOSPT.

Note that no vertex program is necessary to render an MOSPT. The difference between a linearized hierarchy used for an SPT and a linearized hierarchy created for an MOSPT can be seen in Figure 6.4. The SPT hierarchy contains additionally created points for the inner nodes. For the SPT, all points from the root node down to nodes that are just visible from the current viewpoint are sent to the GPU, and the inner points of the three upper levels will be culled. For the MOSPT, the same points are sent to the GPU, but all points will also be rendered, because they are part of the model at any level of detail.

6.4.6 MOSPT Creation

The hierarchy chosen for MOSPTs is an octree with a user-specified maximum recursion level. In a first step, an octree with empty interior nodes is created.

Points are inserted into the octree consecutively and filtered down the hierarchy. There are three possibilities when a point reaches a leaf cell:

1. The leaf cell is empty, and the point is stored there.
2. The leaf already contains a point. Then the leaf is split, and both points are filtered further down the hierarchy.
3. The leaf is at the maximum recursion level and already contains a point. There are two leaf node strategies:
    (a) Simply add the point to the node.
    (b) Reject the new point, as it does not add any new information to the hierarchy.

In relation to nested octrees (see Section 6.5), the first leaf node strategy will be used for inner nodes where the number of input points is below a certain threshold. This guarantees that all input points are stored in the MOSPT. The second leaf node strategy will be used for other inner nodes, with the result that each leaf node stores exactly one point.

In a second step, representatives for the inner nodes are chosen in a bottom-up fashion. For each inner node, we choose the node whose color has the least distance to the average color of the child nodes. The chosen child node is then deleted. Finally, the octree nodes are sorted by r_{max} and stored in a sequential array.

6.5 Nested Octrees

6.5.1 Motivation and Definition

MOSPTs are an efficient data structure for rendering a large number of points. However, they face three major problems:

1. There is no way to do view-frustum culling, e.g. for inside views of the model.
2. Only one level of detail can be selected for the whole model.
3. It is not possible to render models that do not fit into the available memory.

It is well known that the first two problems can be solved by a straight-forward combination of octrees and SPTs, where only the lower levels of the octree are sequentialized, and the upper levels are used for culling and indexing. This would,

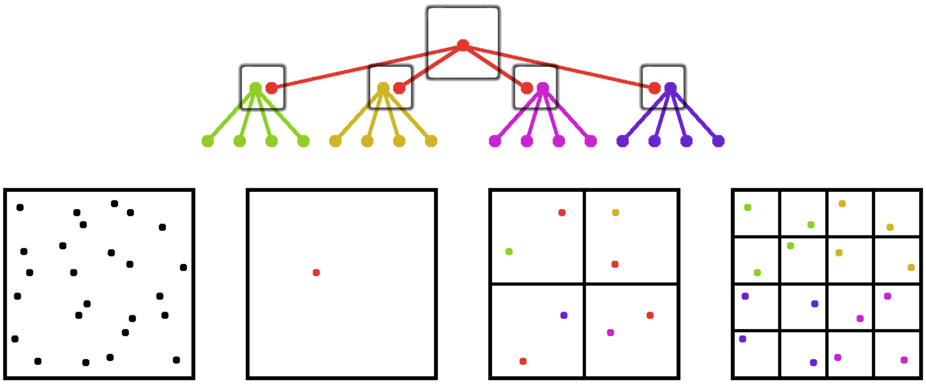


Figure 6.5: 2D example: a nested quadtree of three levels. Inner quadtrees are in color and limited to a depth of two.

however, require most of the SPTs to reside in graphics card memory, since it is unlikely that a whole SPT projects to less than a pixel. A non-sequentialized, classical octree on the other hand would be too slow because of the inadequate use of the graphics API. Therefore, we need a data structure that organizes chunks of points that are large enough so that they can be efficiently processed by the graphics hardware and streamed into memory by a single disk access, and small enough that they allow fine-grained view-frustum culling and memory control.

For this, we introduce *nested octrees*, a data structure that consists of an outer octree and nested inner octrees. The outer octree allows view-frustum culling and out-of-core rendering with incremental refinement, while MOSPTs are used as inner octrees for highly efficient rendering and API use. The main novelty in our nested octree approach is that the inner octrees overlap in the space they occupy. This overlap provides an efficient LOD representation at each level of the outer octree, so that more detailed MOSPTs are only loaded from external memory when required: refining a level in the outer octree only adds one additional level of points to the representation. Each node of the outer octree holds exactly one inner octree, which in turn holds the actual points. Each inner octree has a maximum user-specified depth, as does the outer octree. The result is a *layered* structure similar to layered point clouds [Gobb04], but which does not depend on a uniform sampling density due to the different construction process.

Figure 6.5 shows the 2D case, a nested quadtree. The outer quadtree is represented by the boxes, and is used as traversal hierarchy to reach the inner quadtrees. The overlapping inner quadtrees are limited to depth 2. Figure 6.6 shows the 1D case, a nested binary tree, with inner binary trees with a depth of 3 (the complete outer tree is not shown).

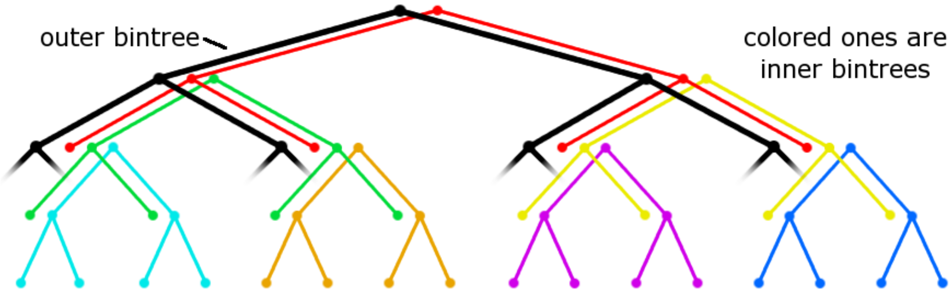


Figure 6.6: 1D example: a nested bintree of 5 levels. The inner bintrees (in color) are limited to depth 3.

6.5.2 Hierarchy Creation

As input data for creating nested octrees we take an arbitrary point cloud possibly with color information. The depth for the inner octrees should be set neither too small (to avoid a too large number of inner octrees) nor too large (as view-frustum culling would becomes less efficient). We perform several sequential passes over the input, keeping a good working set in main memory during each pass. In a preliminary pass, each input file is scanned to build the bounding box of the whole point cloud from the point coordinates. This bounding box is then inflated to a cube and forms the root node *R* of the nested octree. *R* is set to be the current node, and the original point cloud set to be the current input file.

Each subsequent pass performs the following steps until all points are filled into the nested octree:

1. Create a new empty MOSPT for the current node.
2. Set its leaf node strategy to “reject” (see Section 6.4.6).
3. Create a “rejection file” for each child node of the outer octree, named using a unique node identifier (e.g., “R057”).
4. Scan the current input file and insert all points into the new MOSPT. Write each rejected point into one of the 8 rejection files, depending on its position.
5. If the number of points in any of the rejection files is smaller than a threshold:
    (a) Set the leaf node strategy to “add,” so that points are added to instead of rejected from the MOSPT.
    (b) Add the points from the small rejection file to the current MOSPT.
6. Write the current MOSPT to disk.
7. Select a non-processed rejection file, set the current node from the filename and start another pass with this rejection file as input.

Finally, the outer octree, which stores for each node just the filename of the associated MOSPT, is written to disk. As an optimization, the above algorithm can be extended to fill several levels of the outer octree simultaneously. In this case, instead of writing rejected points to a rejection file, they are inserted immediately into an appropriate MOSPT. Although the hierarchy creation theoretically requires $O(\log n)$ passes, this number is very low in practice. For the 262M points cathedral model shown in Section 6.6, we used 18 total octree levels with a MOSPT depth of 7, resulting in 11 outer octree levels. On our 1GB machine, we were able to compute 3 levels simultaneously per pass, so that only 4 passes were needed.

6.5.3 Rendering

For rendering a nested octree, the user sets the maximum number of points (the *budget*) to render for each frame to guarantee interactive navigation. For example, a budget of 12M points will keep frame rates above 10fps on a graphics hardware that processes 120M points/s. The goal of the rendering process is to render the most important points as efficiently as possible. These points are those that are: not view-frustum culled; not contained in a node that falls below a 1-pixel threshold; and are currently loaded into graphics memory.

The outer octree is loaded completely into memory and traversed with the help of a *priority queue*, with the size of the projected node bounding box as priority. There is a second *render queue* that collects MOSPTs that are to be rendered. The MOSPTs are not rendered immediately from the priority queue because the processing of child nodes can change the rendered splat size of the parent node.

For each node that is popped from the priority queue, the following steps happen:

1. Check whether rendering the current MOSPT would exceed the budget. If yes, stop traversal.
2. View-frustum cull the node. If not visible, skip node.
3. Check whether the projected bounding sphere of the *lowest level node* of the associated MOSPT is below a threshold (typically 1 pixel). If this is the case, skip the node.
4. Check whether the associated MOSPT is loaded into graphics card memory.
    - If yes, put the node on the render queue, and put its 8 children on the priority queue.
    - If not, request the MOSPT from external memory and skip it for this frame.

The splat size used to render MOSPTs is determined in each outer octree node by the lowest level descendant (more specifically, the smallest projected bounding box of its associated MOSPT) that has all requested children loaded. This prevents gaps caused by missing nodes. The splat size of leaf nodes will coarsely depend on the sampling density using this construction. Alternatively, a fixed node size can be used for rendering leaf nodes.

When the priority queue has been fully traversed, the rendering queue is traversed and all contained nodes are rendered using graphics hardware.

View-frustum culling is done in clip-coordinates. The bounding boxes of the cells can be calculated during rendering in a way that only requires additions, as described in [Dugu04].

Handling out-of-core requests for MOSPTs that are not in graphics card memory happens in a separate thread so that rendering can continue undisturbed. Each MOSPT is stored in a file that can be loaded directly into a graphics card buffer object without preprocessing. Even though there may be a large number of MOSPT files (about 30K files for a model with 262M points), the operating system provides fast access to the individual files (e.g., the NTFS file system uses B-Trees for large directories). The MOSPTs are managed in two LRU caches, one in graphics card memory and one in main memory.

6.6 Results

We demonstrate Instant Points on a point cloud of a large cathedral, which consists of more than 262M points from 77 scan positions. The accompanying video shows an interactive session where we set a minimum target frame rate of 10 frames/s, which is met or exceeded during the whole interaction. The video also shows that some areas are successively refined as they are streamed in from hard disk. The whole dataset for this animation requires 4GB on harddisk and was created automatically in about 2 hours. Instant Points makes efficient use of current graphics hardware: we achieve a throughput of 105-115M points/s during the whole animation, which is near the maximum point throughput (116M points/s) of our graphics card. Only when the viewpoint is moving fast through the model, the throughput drops to roughly 80M vertices/s. During rendering the cathedral, a maximum of some 400 rendered cells is never exceeded when using 7 levels for the inner octrees, which makes setup times for the graphics card buffer objects during rendering negligible.

These results were generated on an Intel Pentium4 3.2GHz computer with hyper-threading enabled, with two 10,000RPM harddisks (set up as a RAID 0), and 1GB of RAM. The graphics card was an NVIDIA GeForce 6800GTO with a maximum point primitive throughput of 116M points/s according to NVIDIA (we were able to confirm this maximum throughput in our own tests). All scans used

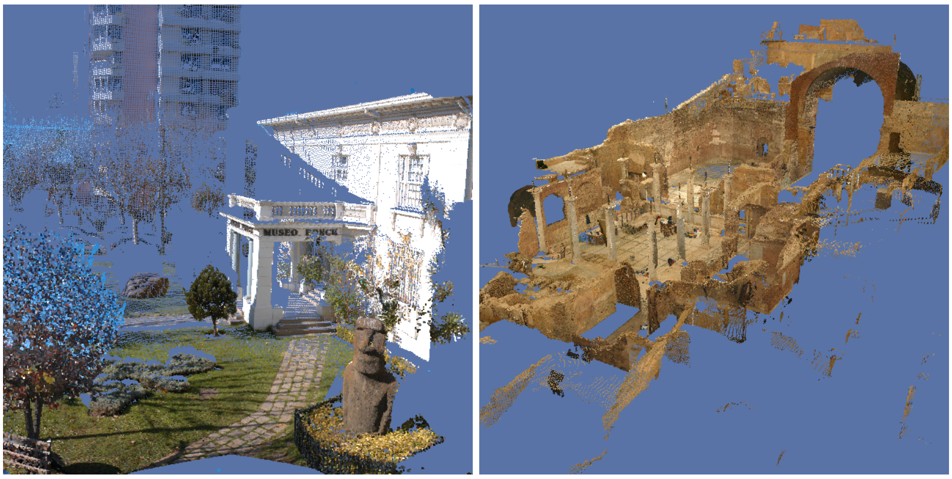


Figure 6.7: Single scans of and island location ($M = 59\%$) and of an archaeological excavation site ($M = 53\%$).

in our tests were obtained using a Riegl LMS Z420i laser scanner with a range of up to 800m at an accuracy of the depth measurement of 1cm.

To compare the efficiency of the original SPTs and MOSPTs on unprocessed point data, we used one single scan of the cathedral. The original SPT contained 10,021,473 points, requiring 243MB memory, whereas MOSPTs required only 6,609,305 points (i.e., the original ones), at 107MB, which corresponds exactly to our projected total overhead of 125%. Figure 6.7 shows two more scenes with very similar overheads, confirming the empirical branching factor of $\alpha = 3$ for scanned datasets.

Hierarchy creation times ranged from 14 minutes for a model of 26M points (8 scan positions, with 48,793 MOSPTs of depth 5), up to slightly over 2 hours for the whole cathedral model (77 scan positions, with 33,887 MOSPTs of depth 7). These creation times are still reasonable even for usage in scanning campaigns, where day-to-day planning has to take into account the already scanned positions, and interaction with those positions is crucial.

The image quality of the rendering can naturally not compete with fully post-processed point cloud models. The goal of this work is not to provide the highest quality rendering—which is simply not possible with unprocessed point clouds—but to provide quick interaction with huge point clouds by exploiting the fact that unprocessed point clouds are simpler in structure. In the case of a high point density, the sub-sampling approach implemented by MOSPTs provides a reasonable approximation that has the advantage that no assumptions about the point cloud have to be made. Figure 6.8 compares a full SPT calculated using averaging with an MOSPT using subsampling. Another property of the algorithm is that it does not take the direction from where the sample was taken into account. It is therefore not possible to distinguish between front- and backfacing geometry, as is evident in the cathedral video. This is a topic for future work.

The out-of-core process inherently also leads to popping artifacts when new information is streamed into memory to refine the model. However, these popping artifacts are only temporary, and when the viewer remains stationary for a short time, he will be able to observe the full quality model when all the points are loaded. Note that not all available points will be loaded by the system, but only the amount of points that result in the desired target frame rate. Figure 6.9 shows screenshots from the walkthrough where the image quality of the Instant Points algorithm can be observed.

6.7 Conclusions and Future Work

We have presented Instant Points, a system to render huge unprocessed point clouds with only little preprocessing. The algorithm does not rely on normal vector or splat size estimation and can therefore render models with strongly varying densities and many undersampled areas, which occur often in 3D range scanner data. This is becoming a more and more important topic, since interaction with such models is often necessary already before lengthy postprocessing to fix the model or even manual mesh creation can take place. The system consists of an out-of-core data structure called nested octrees, and utilizes an improved version of sequential point trees called MOSPT, which take advantage of the restrictions of unprocessed point clouds and require less memory and render faster than SPTs.

In terms of future work, we want to investigate more advanced methods to adapt the splat size when zooming into the model, which is difficult in the absence of neighborhood information. In the realm of triangle rendering, the randomized z-buffer approach [Wand01] handles huge polygonal scenes, and it would be interesting to compare their randomization techniques to ours. Another acceleration technique commonly used in triangle rendering is occlusion culling. While the scene structure even of huge scanned datasets like the cathedral is not necessarily amenable to culling, integrating the coherent hierarchical culling algorithm proposed by Bittner et al. [Bitt04] into the render queue of nested octrees seems straightforward. Finally, the memory savings obtained by MOSPTs need to be contrasted with dedicated compression techniques, which can achieve much higher compression rates at some additional cost (e.g., using quantization and delta-coding [Krüg05]). We will investigate how these two complimentary approaches can be combined while maintaining a high rendering speed.

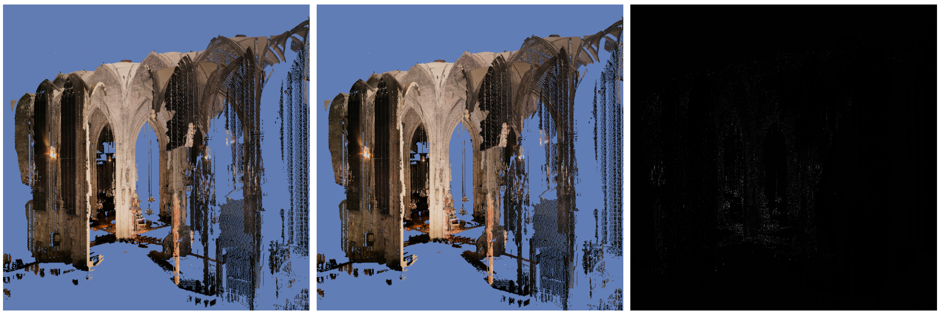


Figure 6.8: Quality comparison of SPT (left) and MOSPT (middle) rendering. The colors for the SPT were obtained by averaging, those in MOSPT come from selecting representative colors from child nodes. While the difference image (right) does show some discrepancies, the order of magnitude of these differences is not higher than that of the noise contained in the original data.

Acknowledgements

This work was supported by the TUW-Ilscan Centre of Competence at the Vienna University of Technology.

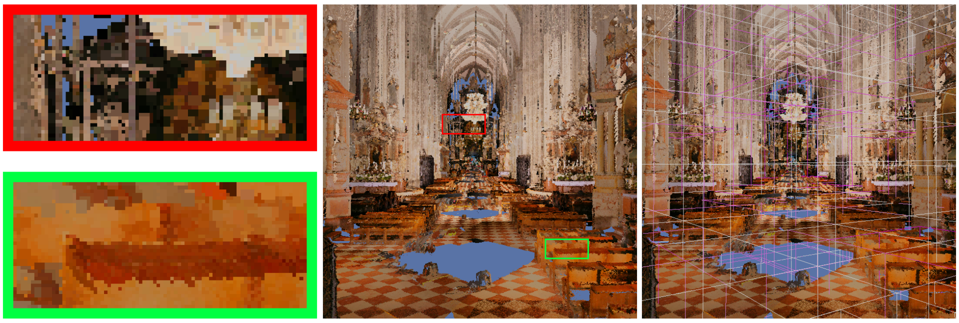


Figure 6.9: Screenshots of the walkthrough in the cathedral model. The zoomed parts of the middle image show that the point sizes are similar for the far away and for the nearby regions, proving that the LOD selection works well. The middle image was rendered with 9,890,458 points in 308 cells, and loading was completed. In the right image, the magenta bounding boxes indicate that some children of these cells were not rendered because the projected sizes of their bounding boxes were too small.

Bibliography

[Aila04a] T. Aila and S. Laine. Alias-Free Shadow Maps. In *Proceedings of the Eurographics Symposium on Rendering 2004*, pages 161–166. Eurographics, Eurographics Association, June 2004. Cited on page 44.

[Aila04b] Timo Aila and Ville Miettinen. dPVS: An Occlusion Culling System for Massive Dynamic Environments. *IEEE Computer Graphics & Applications*, 24(2):86–97, 2004. Cited on page 95.

[Aire90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 41–50, March 1990. Cited on pages 94 and 95.

[Alia99] Daniel G. Aliaga and Anselmo Lastra. Automatic Image Placement to Provide a Guaranteed Frame Rate. In Alyn Rockwood, editor, *SIGGRAPH 99 Conference Proceedings*, Annual Conference Series, pages 307–316. ACM SIGGRAPH, Addison Wesley, August 1999. ISBN 0-201-48560-5. Cited on pages 15, 18, and 29.

[Andu00] C. Andujar, C. Saona, and I. Navazo. LOD Visibility Culling and Occluder Synthesis. *Computer Aided Design*, 32(13):773–783, 2000. Cited on page 94.

[Artu03] Alessandro Artusi, Jiří Bittner, Michael Wimmer, and Alexander Wilkie. Delivering Interactivity to Complex Tone Mapping Operators. In Per Christensen and Daniel Cohen-Or, editors, *Rendering Techniques 2003 (Proceedings Eurographics Symposium on Rendering)*, pages 38–44. Eurographics, Eurographics Association, June 2003. ISBN 3-905673-03-7. Cited on page 5.

[ATI 01] ATI Technologies Inc. TRUFORM – White Paper. `http://www.ati.com/technology/`, 2001. Cited on page 24.

[ATI 03] ATI Technologies Inc. ATI OpenGL extensions, Vertex Array Object extension. `http://www.ati.com/`, 2003. Cited on page 23.

[Bitt01] Jiří Bittner, Peter Wonka, and Michael Wimmer. Visibility Preprocessing for Urban Scenes using Line Space Subdivision. In Bob Werner, editor, *Proceedings of Pacific Graphics 2001 (Ninth Pacific Conference on Computer Graphics and Applications)*, pages 276–284, Los Alamitos, CA, October 2001. IEEE Computer Society Press. ISBN 0-7695-1227-5. Cited on pages 4, 77, and 94.

[Bitt02] Jiří Bittner. Efficient construction of visibility maps using approximate occlusion sweep. In *SCCG '02: Proceedings of the 18th spring conference on Computer graphics*, pages 167–175, 2002. ISBN 1-58113-608-0. Cited on pages 84 and 96.

[Bitt03] J. Bittner. *Hierarchical Techniques for Visibility Computations.* PhD Thesis, Czech Technical University in Prague, 2003. Cited on pages 88 and 94.

[Bitt04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum*, 23(3):615–624, September 2004. Proceedings EUROGRAPHICS 2004, ISSN 0167-7055. Cited on pages 4, 10, 59, 61, and 117.

[Bitt05] Jiří Bittner, Peter Wonka, and Michael Wimmer. Fast Exact From-Region Visibility in Urban Scenes. In Kavita Bala and Philip DutrÃľ, editors, *Rendering Techniques 2005 (Proceedings Eurographics Symposium on Rendering)*, pages 223–230. Eurographics, Eurographics Association, June 2005. ISBN 3-905673-23-1. Cited on pages 4 and 77.

[Bots05] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-Quality Surface Splatting on Today's GPUs. In *Eurographics Symposium on Point-Based Graphics*, pages 17–24, 2005. Cited on pages 7 and 105.

[BOUB05] T. BOUBEKEUR, F. DUGUET, and C. SCHLICK. Rapid Visualization of Large Point-Based Surfaces. In *Proceedings of VAST 2005*, pages 75–82, 2005. Cited on page 106.

[Brab02] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Practical shadow mapping. *Journal of Graphics Tools: JGT*, 7(4):9–18, 2002. ISSN 1086-7651. Cited on page 43.

[Čadí06] Martin Čadík, Michael Wimmer, Laszlo Neumann, and Alessandro Artusi. Image Attributes and Quality for Evaluation of Tone Mapping Operators. In *Proceedings of Pacific Graphics 2006 (14th Pacific Conference on Computer Graphics and Applications)*, pages 35–44. National Taiwan University Press, October 2006. Cited on page 5.

[Carm00] John Carmack. Plan update 03/07/00, 2000. available at `http://finger.planetquake.com/plan.asp?userid=johnc&id=14310`. Cited on page 22.

[Chhu05] Jatin Chhugani, Budirijanto Purnomo, Shankar Krishnan, Jonathan Cohen, Suresh Venkatasubramanian, and David S. Johnson. vLOD: High-Fidelity Walkthrough of Large Virtual Environments. *IEEE Trans. on Visualization and Computer Graphics*, 11(1):35–47, 2005. ISSN 1077-2626. Cited on page 97.

[Chon04] H. Chong and S. Gortler. A Lixel for every Pixel. In *Proceedings of the Eurographics Symposium on Rendering 2004*, pages 167Ű–172. Eurographics, Eurographics Association, June 2004. Cited on pages 43 and 44.

[Cohe03] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A Survey of Visibility for Walkthrough Applications. *IEEE Trans. on Visualization and Computer Graphics*, 9(3):412–431, 2003. Cited on pages 61, 62, 63, 73, and 94.

[Crow77] Franklin C. Crow. Shadow Algorithms for Computer Graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11(2):242–248, July 1977. Cited on pages 6 and 43.

[Dach03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Trans. on Graphics*, 22(3):657–662, 2003. Cited on pages 105 and 106.

[Dey05] T. K. Dey, G. Li, and J. Sun. Normal Estimation for Point Clouds : A Comparison Study for a Voronoi Based Method. In *Eurographics Symposium on Point-Based Graphics*, pages 39–46, 2005. Cited on page 103.

[Dugu02] Florent Duguet and George Drettakis. Robust Epsilon Visibility. In *Proc. ACM SIGGRAPH 2002*, pages 567–575, July 2002. Cited on page 94.

[Dugu04] Florent Duguet and George Drettakis. Flexible point-based rendering on mobile devices. *Computer Graphics and Applications*, 24(4):57–63, 2004. Cited on pages 105, 107, and 115.

[Dura99] Fredo Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Universite Joseph Fourier, Grenoble, France, July 1999. Cited on pages 94 and 96.

[Dura00] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. Conservative Visibility Preprocessing Using Extended Projections. In

Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 239–248. ACM SIGGRAPH, Addison Wesley, July 2000. Cited on page 94.

[Fern01] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive Shadow Maps. In *SIGGRAPH 2001 Conference Proceedings*, pages 387–390, 2001. Cited on page 43.

[Funk93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. In James T. Kajiya, editor, *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 247–254. ACM SIGGRAPH, Addison Wesley, August 1993. ISBN 0-201-51585-7. Cited on pages 3, 15, 17, 25, 30, and 32.

[Gart93] N. Gartner, C. Messer, and A. Rathi. Traffic Flow Theory. Technical Report, Turner-Fairbank Highway Research Center, 1993. Cited on page 21.

[Gieg06] Markus Giegl and Michael Wimmer. Queried Virtual Shadow Maps. In Wolfgang Engel, editor, *ShaderX 5 – Advanced Rendering Techniques*, volume 5 of *ShaderX*. Charles River Media, December 2006. ISBN 1-58450-499-4. Cited on pages 6 and 39.

[Gieg07a] Markus Giegl and Michael Wimmer. Fitted Virtual Shadow Maps. In *Proceedings of Graphics Interface 2007*, May 2007. Cited on pages 6 and 39.

[Gieg07b] Markus Giegl and Michael Wimmer. Queried Virtual Shadow Maps. In *Proceedings of ACM SIGGRAPH 2007 Symposium on Interactive 3D Graphics and Games*, pages 65–72, New York, NY, USA, April 2007. ACM Press. ISBN 978-1-59593-628-8. Cited on pages 6 and 39.

[Gieg07c] Markus Giegl and Michael Wimmer. Unpopping: Solving the Image-Space Blend Problem for Smooth Discrete LOD Transitions. *Computer Graphics Forum*, 26(1):46–49, March 2007. ISSN 0167-7055. Cited on page 4.

[Gobb04] Enrico Gobbetti and Fabio Marton. Layered Point Clouds. In *Eurographics Symposium on Point-Based Graphics*, pages 113–120, 2004. ISBN 3-905673-09-6. Cited on pages 105 and 112.

[Gots99] C. Gotsman, O. Sudarsky, and J. Fayman. Optimized Occlusion Culling Using Five-Dimensional Subdivision. *Computers and Graphics*, 5(23):645–654, 1999. Cited on page 95.

[Govi03] Naga K. Govindaraju, Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha. Interactive shadow generation in complex environments. *ACM Transactions on Graphics*, 22(3):501–510, July 2003. Cited on page 43.

[Guen06] Gael Guennebaud, Loïc Barthe, and Mathias Paulin. Real-time soft shadow mapping by backprojection. In *Eurographics Symposium on Rendering, Nicosia, Cyprus, 26/06/06-28/06/06*, pages 227–234, http://www.eg.org/, 2006. Eurographics. Cited on page 6.

[Habe07] Ralf Habel, Alexander Kusternig, and Michael Wimmer. Physically Based Real-Time Translucency for Leaves. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)*. Eurographics, June 2007. Cited on page 5.

[Haum05] Denis Haumont, Otso Mäkinen, and Shaun Nirenstein. A Low Dimensional Framework for Exact Polygon-to-Polygon Occlusion Queries. In *Proc. Eurographics Symposium on Rendering*, pages 211–222, June 2005. Cited on page 94.

[Helm94] James L. Helman. Architecture and Performance of Entertainment Systems, Appendix A. *ACM SIGGRAPH 94 Course Notes - Designing Real-Time Graphics for Entertainment*, 23:1.19–1.32, July 1994. Cited on pages 3, 15, and 33.

[Hopp99] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In Alyn Rockwood, editor, *SIGGRAPH 99 Conference Proceedings*, Annual Conference Series, pages 269–276. ACM SIGGRAPH, Addison Wesley, August 1999. Cited on pages 17 and 30.

[Jesc02a] Stefan Jeschke and Michael Wimmer. Textured Depth Meshes for Real-Time Rendering of Arbitrary Scenes. In Paul Debevec and Simon Gibson, editors, *Rendering Techniques 2002 (Proceedings Eurographics Workshop on Rendering)*, pages 181–190. Eurographics, Eurographics Association, June 2002. ISBN 1-58133-534-3. Cited on page 5.

[Jesc02b] Stefan Jeschke, Michael Wimmer, and Heidrun Schumann. Layered Environment-Map Impostors for Arbitrary Scenes. In Wolfgang Stürzlinger and Michael McCool, editors, *Proceedings of Graphics Interface 2002*, pages 1–8. AK Peters Ltd., May 2002. ISBN 1-56881-183-7. Cited on page 5.

[Jesc05] Stefan Jeschke, Michael Wimmer, Heidrun Schumann, and Werner Purgathofer. Automatic Impostor Placement for Guaranteed Frame

Rates and Low Memory Requirements. In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*, pages 103–110. ACM, ACM Press, April 2005. ISBN 1-59593-013-2. Cited on pages 5 and 98.

[Jesc07] Stefan Jeschke, Stephan Mantler, and Michael Wimmer. Interactive Smooth and Curved Shell Mapping. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)*, page 10. Eurographics, 6 2007. Cited on page 5.

[Kolt01] Vladlen Koltun, Yiorgos Chrysanthou, and Cohen-Or Cohen-Or. Hardware-Accelerated from-Region Visibility Using a Dual Ray Space. In *Rendering Techniques 2001*, pages 205–216, 2001. Cited on page 94.

[Kozl04] Simon Kozlov. Perspective Shadow Maps - Care and Feeding. In *GPU Gems*, pages 214–244. Addison-Wesley, 2004. Cited on page 44.

[Krüg05] Jens Krüger, Jens Schneider, and Rüdiger Westermann. DuoDecim - A Structure for Point Scan Compression and Rendering. In *Eurographics Symposium on Point-Based Graphics*, pages 99–107, 2005. Cited on pages 106 and 117.

[Leht00] Lasse Lehtinen. 3Dfx Voodoo and Voodoo 2 FAQ, 2000. available at `http://user.sgic.fi/~blob/Voodoo-FAQ/`. Cited on page 2.

[Levo96] Marc Levoy and Pat Hanrahan. Light Field Rendering. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. Cited on page 95.

[Leyv03] Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. Ray space factorization for from-region visibility. *ACM Transactions on Graphics*, 22(3):595–604, 2003. ISSN 0730-0301. Cited on page 94.

[Low03] Kok-Lim Low and Adrian Ilie. Computing a View Frustum to Maximize an Object's Image Area. *Journal of Graphics Tools: JGT*, 8(1):3–15, 2003. ISSN 1086-7651. Cited on page 43.

[Lueb97] David Luebke and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 199–208. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7. Cited on page 108.

[Mart04] T. Martin and T.-S. Tan. Anti-aliasing and Continuity with Trapezoidal Shadow Maps. In *Proceedings of the Eurographics Symposium on Rendering 2004*, pages 153–160. Eurographics, Eurographics Association, June 2004. Cited on page 44.

[Matt06] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. Adaptive Visibility-Driven View Cell Construction. In Wolfgang Heidrich and Tomas Akenine-Moller, editors, *Rendering Techniques 2006 (Proceedings Eurographics Symposium on Rendering)*, pages 195–206. Eurographics, Eurographics Association, June 2006. ISBN 3-90567-335-5. Cited on pages 4 and 77.

[Matt07] Oliver Mattausch, Jiří Bittner, Peter Wonka, and Michael Wimmer. Optimized Subdivisions for Preprocessed Visibility. In *Proceedings of Graphics Interface 2007*, May 2007. Cited on pages 4 and 77.

[McDe87] D. McDermott and A. Gelsey. Terrain Analysis For Tactical Situation Assessment. In *Proceedings Spatial Reasoning and Multi-Sensor Fusion*, pages 420–429, 1987. Cited on page 98.

[Möll02] Tomas Möller and Eric Haines. *Real-Time Rendering, Second Edition*. A. K. Peters Limited, 2002. ISBN 1568811829. Cited on page 44.

[Mora05] Frédéric Mora, Lilian Aveneau, and Michel Mériaux. Coherent and Exact Polygon-to-Polygon Visibility. In *Proceedings of Winter School on Computer Graphics 2005*, pages 87–94, 2005. Cited on page 94.

[Müll06] Pascal Müller, Peter Wonka, Simon Hägler, Andreas Ulmer, and Luc Van Gool. Procedural Modeling of Buildings. *ACM Transactions on Graphics*, 25(3):614–623, 2006. Cited on page 88.

[Nied92] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM Philadelphia, 1992. Cited on page 84.

[Nire02] S. Nirenstein, E. Blake, and J. Gain. Exact from-region visibility culling. In *Rendering Techniques 2002*, pages 191–202, 2002. Cited on pages 82 and 94.

[Nire04] S. Nirenstein and E. Blake. Hardware Accelerated Visibility Preprocessing using Adaptive Sampling. In *Rendering Techniques 2004*, pages 207–216, 2004. Cited on pages 88, 89, 92, 93, and 95.

[NVID03a] NVIDIA Corporation. NVIDIA Developer Website, 2003. `http://developer.nvidia.com/`. Cited on page 24.

[NVID03b] NVIDIA Corporation. NVIDIA OpenGL Specifications, Vertex Array Range extension, 2003. available at `http://developer.nvidia.com/`. Cited on page 23.

[NVID04] NVIDIA Corporation. NVIDIA GPU Programming Guide, 2004. `http://developer.nvidia.com/object/gpu\_programming\_guide.html`. Cited on page 63.

[Paja05] Renato Pajarola, Miguel Sainz, and Roberto Lario. XSplat: External Memory Multiresolution Point Visualization. In *Proceedings IASTED International Conference on Visualization, Imaging and Image Processing*, pages 628–633, 2005. Cited on page 105.

[Pfis00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface Elements as Rendering Primitives. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 335–342. ACM SIGGRAPH, Addison Wesley, 2000. Cited on pages 7 and 105.

[Phon75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975. ISSN 0001-0782. Cited on page 5.

[Pito99] Richard Pito. A Solution to the Next Best View Problem for Automated Surface Acquisition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(10):1016–1030, 1999. ISSN 0162-8828. Cited on pages 81 and 95.

[Prou01] Kekoa Proudfoot, William R. Mark, Pat Hanrahan, and Svetoslav Tzvetkov. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In Stephen Spencer, editor, *SIGGRAPH 2001 Conference Proceedings*, Annual Conference Series, pages 159–170, New York, August 12–17 2001. ACM Press. Cited on page 24.

[Reev87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering Antialiased Shadows with Depth Maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):283–291, July 1987. Cited on pages 43 and 49.

[Rega94] Matthew Regan and Ronald Pose. Priority Rendering with a Virtual Reality Address Recalculation Pipeline. In *SIGGRAPH 94 Conference Proceedings*, pages 155–162, 1994. Cited on page 18.

[Resh05] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. on Graphics*, 24(3):1176–1185, 2005. Cited on pages 11, 83, and 88.

[Rohl94] John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real–Time 3D Graphics. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0. Cited on page 18.

[Ross93] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications (Proc. Second Conference on Geometric Modelling in Computer Graphics*, pages 455–465, Berlin, June 1993. Springer-Verlag. Cited on page 108.

[Rusi00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 343–352. ACM SIGGRAPH, Addison Wesley, July 2000. Cited on pages 7 and 105.

[Sber93] M. Sbert. An Integral Geometry Method for Fast Form Factor Computation. *Computer Graphics Forum*, 12(3):C409–C420, 1993. Cited on page 84.

[Scha00] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. Conservative Volumetric Visibility with Occluder Fusion. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 229–238. ACM SIGGRAPH, Addison Wesley, July 2000. Cited on pages 94 and 95.

[Sche07] Daniel Scherzer, Stefan Jeschke, and Michael Wimmer. Pixel-Correct Shadow Maps with Temporal Reprojection and Shadow Test Confidence. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering).* Eurographics, June 2007. Cited on pages 6 and 39.

[Seku04] Dean Sekulic. Efficient Occlusion Culling. In Randima Fernando, editor, *GPU Gems*, pages 487–503. Addison-Wesley, 2004. Cited on pages 63 and 73.

[Sen03] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Shadow silhouette maps. *ACM Trans. Graph.*, 22(3):521–526, 2003. ISSN 0730-0301. Cited on page 43.

[Shad98] Jonathan W. Shade, Steven J. Gortler, Li-wei He, and Richard Szeliski. Layered Depth Images. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 231–242. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8. Cited on page 95.

[Sill94] François Sillion and Claude Puech. *Radiosity and Global Illumination.* Morgan Kaufmann Publishers, San Francisco, 1994. ISBN 1-558. Cited on page 6.

[Sloa02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. In Eugene Fiume, editor, *Proceedings of ACM SIGGRAPH 2002 (SIGGRAPH-02)*, volume 21 of *acm Transactions on Graphics*, pages 527–536, San Antonio, USA, July 2002. Association of Computing Machinery (ACM), ACM. ISBN 1-58113-521-1. Cited on page 6.

[Stam02] Marc Stamminger and George Drettakis. Perspective Shadow Maps. In *Siggraph 2002 Conference Proceedings*, volume 21, 3, pages 557–562, July 2002. Cited on pages 41, 43, and 45.

[Stue99] Wolfgang Stuerzlinger. Imaging all Visible Surfaces. In *Proc. Graphics Interface 1999*, pages 115–122, June 1999. Cited on page 95.

[Tell91] Seth J. Teller and Carlo H. Séquin. Visibility Preprocessing for Interactive Walkthroughs. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH 91 Proceedings)*, volume 25, pages 61–69. ACM SIGGRAPH, ACM Press, July 1991. Cited on page 94.

[Wahl05] R. Wahl, M. Guthe, and R. Klein. Identifying Planes in Point-Clouds for Efficient Hybrid Rendering. In *13th Pacific Conference on Computer Graphics and Applications*, 2005. Cited on page 105.

[Wald03] Ingo Wald, Timothy J. Purcell, Joerg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003. Cited on page 83.

[Wald04] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Rendering Techniques 2004*, pages 81–92, June 2004. Cited on page 92.

[Wand01] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *Proc. ACM SIGGRAPH 2001*, pages 361–370, 2001. ISBN 1-58113-374-X. Cited on page 117.

[Wang94] Yulan Wang and Steven Molnar. Second-Depth Shadow Mapping. Technical Report, University of North Carolina at Chapel Hill, 1994. Cited on page 43.

[Will78] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12(3):270–274, August 1978. Cited on pages 6, 41, and 43.

[Wils03] Andrew Wilson and Dinesh Manocha. Simplifying complex environments using incremental textured depth meshes. *ACM Transactions on Graphics*, 22(3):678–688, 2003. ISSN 0730-0301. Cited on page 95.

[Wimm98] Michael Wimmer and Dieter Schmalstieg. Load Balancing for Smooth LODs. Technical Report TR-186-2-98-31, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, December 1998. human contact: technical-report@cg.tuwien.ac.at. Cited on page 4.

[Wimm99a] Michael Wimmer, Markus Giegl, and Dieter Schmalstieg. Fast Walkthroughs with Image Caches and Ray Casting. *Computers and Graphics*, 23(6):831–838, December 1999. ISSN 0097-8493. Cited on page 5.

[Wimm99b] Michael Wimmer, Markus Giegl, and Dieter Schmalstieg. Fast Walkthroughs with Image Caches and Ray Casting. In Michael Gervautz, Dieter Schmalstieg, and Axel Hildebrand, editors, *Virtual Environments '99. Proceedings of the 5th Eurographics Workshop on Virtual Environments*, pages 73–84. Eurographics, Springer-Verlag Wien, June 1999. ISBN 3-211-83347-1. Cited on page 5.

[Wimm01] Michael Wimmer, Peter Wonka, and François Sillion. Point-Based Impostors for Real-Time Visualization. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques 2001 (Proceedings Eurographics Workshop on Rendering)*, pages 163–176. Eurographics, Springer-Verlag, June 2001. ISBN 3-211-83709-4. Cited on page 5.

[Wimm03] Michael Wimmer and Peter Wonka. Rendering Time Estimation for Real-Time Rendering. In Per Christensen and Daniel Cohen-Or, editors, *Rendering Techniques 2003 (Proceedings Eurographics Symposium on Rendering)*, pages 118–129. Eurographics, Eurographics Association, June 2003. ISBN 3-905673-03-7. Cited on page 8.

[Wimm04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light Space Perspective Shadow Maps. In Alexander Keller and Henrik W. Jensen, editors, *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, pages 143–151. Eurographics, Eurographics Association, June 2004. ISBN 3-905673-12-6. Cited on page 8.

[Wimm05] Michael Wimmer and Jiří Bittner. Hardware Occlusion Queries Made Useful. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, March 2005. ISBN 0-32133-559-7. Cited on page 8.

[Wimm06a] Michael Wimmer and Claus Scheiblauer. Instant Points. In *Proceedings Symposium on Point-Based Graphics 2006*, pages 129–136. Eurographics, Eurographics Association, July 2006. ISBN 3-90567-332-0. Cited on page 8.

[Wimm06b] Michael Wimmer and Daniel Scherzer. Robust Shadow Mapping with Light Space Perspective Shadow Maps. In Wolfgang Engel, editor, *ShaderX 4 – Advanced Rendering Techniques*, volume 4 of *ShaderX*. Charles River Media, March 2006. ISBN 1-58450-425-0. Cited on pages 6 and 39.

[Wlok03] Matthias Wloka. Batch, Batch, Batch: What does It Really Mean? Presentation at Game Developers Conference 2003, 2003. `http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf`. Cited on page 65.

[Wonk00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. In Bernard Péroche and Holly Rushmeier, editors, *Rendering Techniques 2000 (Proceedings Eurographics Workshop on Rendering)*, pages 71–82. Eurographics, Springer-Verlag Wien New York, June 2000. ISBN 3-211-83535-0. Cited on pages 4, 34, and 94.

[Wonk01] Peter Wonka, Michael Wimmer, and François Sillion. Instant Visibility. *Computer Graphics Forum*, 20(3):411–421, September 2001. Günther Enderle [Best Paper] Award, Best Student Paper Award. A. Chalmers and T.-M. Rhyne (eds.), Proceedings EUROGRAPHICS 2001, ISSN 0167-7055. Cited on pages 4 and 98.

[Wonk06] Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov. Guided Visibility Sampling. *ACM Transactions on Graphics*, 25(3):494–502, July 2006. Proceedings ACM SIGGRAPH 2006, ISSN 0730-0301. Cited on page 8.

[Woo99] M. Woo, J. Neider, T. Davis, D. Shreiner, and OpenGL Architecture Review Board. *OpenGL Programming Guide*. Addison Wesley, 1999. Cited on page 34.

[Woop05] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics*, 24(3):434–444, 2005. ISSN 0730-0301. Cited on page 92.

[Xu04] Hui Xu and Baoquan Chen. Stylized Rendering of 3D Scanned Real-world Environments. In *Proc. Symposium on Non-Photorealistic Animation and Rendering 2004*, pages 25–34, 2004. Cited on page 105.

[Yoon05] Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha. Quick VDR: Out-of-Core View-Dependent Rendering of Gigantic Models. *IEEE Trans. on Visualization and Computer Graphics*, 11(4):369–382, 2005. Cited on page 105.

Acknowledgements

This thesis would not have been possible without the help of many people. First and foremost I would like to thank Werner Purgathofer, head of the Institute of Computer Graphics and Algorithms. He provided me with ideal working conditions, supported me in my research endeavors, protected me from taking on too many tasks when I was already overworked, and most of all, allowed me the freedom to pursue the research directions I chose and to coordinate the different projects I acquired over time under my responsibility.

I would like to thank the PhD students I supervised or co-supervised, or am still supervising, for their patience when trying to get hold of me, and the diploma students who helped implement the ideas that lead to the papers in this thesis, especially Daniel Scherzer and Claus Scheiblauer.

My colleagues Peter Wonka, Jiří Bittner and Stefan Jeschke were always an inspiration. It is a pleasure to work with them, and I am grateful for their contributions to the work in this thesis.

A special mention goes to our administrative angels, especially Anita Maierhofer in all financial affairs, and Andrea Fübi and Viktoria Daubner in all other aspects. They made my life at the Institute of Computer Graphics and Algorithms so much easier. The same holds true for our technicians Stephan Plepelits, Andreas Weiner and Alexander Piskernik, who recovered many a `rm -rf`. In general, I greatly enjoyed the working climate at the Institute, in which our beloved Master Eduard Gröller played no small part.

I am grateful to the funding agencies who so generously supported parts of this work or follow-up research to this work, especially the EU under the FP6 project GameTools (IST-2-004363), the FFG in the project Scanopy, and the TUWIlscan center of competence at the Vienna University of Technology.

Finally, my deepest gratitude goes to my wife Romana Wimmer and my daughter Sarah Wimmer, who showed me that there is life and reason beyond work, and who always supported me in times of greatest need.