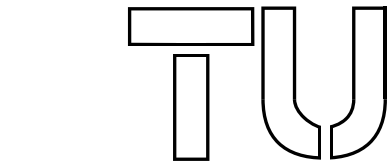


TECHNISCHE UNIVERSITÄT WIEN

Dissertation

Representing and Rendering Distant Objects for Real-Time Visualization

ausgeführt
zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

unter der Leitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Michael Gervautz,
Institut 186 für Computergraphik und Algorithmen,
und unter Mitwirkung von
Univ.-Ass. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

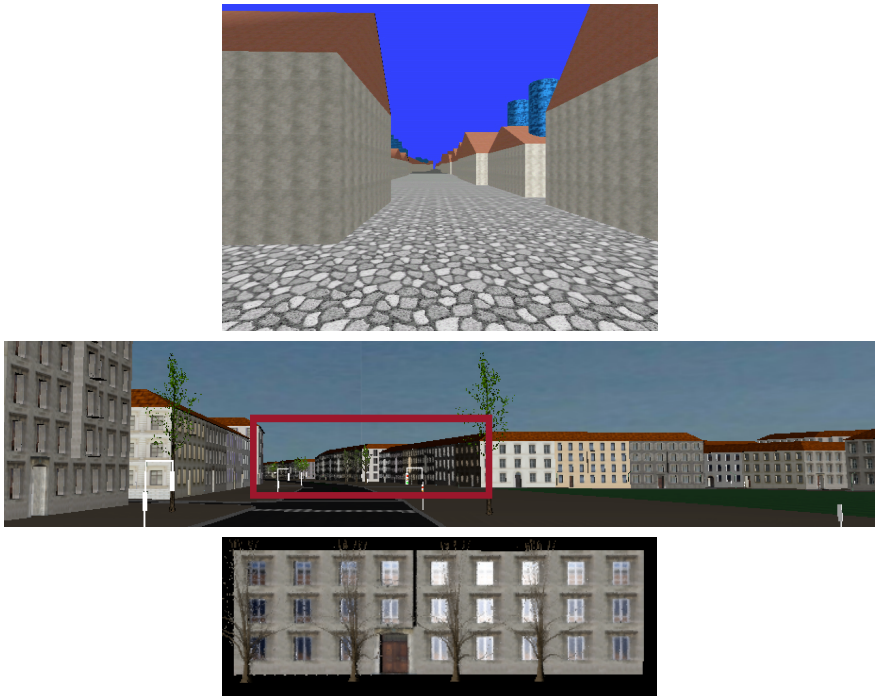
eingereicht
an der Technischen Universität Wien,
Fakultät für Technische Naturwissenschaften und Informatik,

von
Dipl.-Ing. Michael Wimmer,
Matrikelnummer 9225135,
Veronikagasse 42/14,
A-1170 Wien, Österreich,
geboren am 23. Dezember 1973 in Wien.

Wien, im Juni 2001.

Michael Wimmer (PhD Thesis)

Representing and Rendering Distant Objects for Real-Time Visualization



http://www.cg.tuwien.ac.at/research/vr/urbanviz/
mailto:wimmer@cg.tuwien.ac.at

reviewer:
Michael Gervautz
François X. Sillion
Dieter Schmalstieg

Abstract

Computer graphics is the art of creating believable images. The difficulty in many applications lies in doing so quickly. Architectural walkthroughs, urban simulation, computer games and many others require high-quality representation of very large models at interactive update rates. This usually means creating a new image at least 60 times a second. This is what real-time visualization is about.

This thesis presents two methods to accelerate the rendering of very large virtual environments. Both algorithms exploit a common property of many such environments: distant objects usually take up a significant amount of computation time during rendering, but contribute only little to the final image. This thesis shows how to represent and render distant objects with a complexity proportional to the image area they cover, and not to their actual complexity. The algorithms are destined for different scenarios: the first is an online algorithm that carries out all computation during runtime and does not require precomputation. The second algorithm makes use of preprocessing to speed up online rendering and to improve rendering quality.

The first part of the thesis shows an output-sensitive rendering algorithm for accelerating walkthroughs of large, densely occluded virtual environments using a multi-stage image-based rendering pipeline. In the first stage of the pipeline, objects within a certain distance (the near field) are rendered using the traditional graphics pipeline. In the following stages, the remainder of the scene (the far field), which consists of all pixels not yet covered by near-field geometry, is rendered by a pixel-based approach using a panoramic image cache, horizon estimation to avoid calculating sky pixels, and finally, ray casting. The time complexity of the approach does not depend on the total number of primitives in the scene. We have measured speedups of up to one order of magnitude compared to standard rendering with view-frustum culling.

In the second part of the thesis, a new data structure for encoding the appearance of a geometric model as seen from a viewing region (view cell) is presented. This representation can be used in interactive or real-time visualization applications to replace complex models—especially distant geometry—by an impostor, maintaining high-quality rendering while cutting down on rendering time. The approach relies on an object-space sampled representation similar to a point cloud or a layered depth image, but introduces two fundamental additions to previous techniques. First, the sampling rate is controlled to provide sufficient density across all possible viewing conditions from the specified view cell. Second, a correct, antialiased representation of the plenoptic function is computed using Monte Carlo integration. The system therefore achieves high-quality rendering using a simple representation with bounded complexity.

This thesis also contains a comprehensive overview of related work in the field of real-time visualization, and an in-depth discussion of the advantages and disadvantages of image-based and point-based representations for distant objects.

Kurzfassung

Computergraphik ist die Wissenschaft, die sich mit der Generierung glaubwürdiger Bilder beschäftigt. Eine der größten Herausforderungen dabei ist, diese Bilder in ausreichender Geschwindigkeit zu erzeugen. Speziell bei der Simulation von Fahrzeugen in Stadtgebieten, bei der virtuellen Erforschung von Gebäuden (ob noch nicht gebaute, existierende oder schon lange zerstörte), bei Computerspielen und vielen anderen Anwendungen ist es wichtig, daß die Bilder in flüssiger Abfolge erscheinen. Üblicherweise versteht man darunter eine Bildrate von mindestens 60 Bildern pro Sekunde. Das ist das Thema der Echtzeitvisualisierung.

In dieser Dissertation werden zwei Algorithmen zur beschleunigten Darstellung von großen virtuellen Szenen vorgestellt. Dabei wird bei beiden Algorithmen eine interessante Eigenschaft von vielen solchen Szenen ausgenützt: Objekte, die sich weiter weg vom Betrachter befinden, machen nur einen kleinen Teil des endgültigen Bildes aus, benötigen aber relativ viel Rechenzeit. In dieser Dissertation wird gezeigt, wie man entfernte Objekte mit einer Komplexität, die der überdeckten Bildfläche – und nicht ihrer eigentlichen geometrischen Komplexität – entspricht, repräsentieren und darstellen kann. Die beiden Algorithmen sind für unterschiedliche Szenarien gedacht. Die erste Methode funktioniert zur Laufzeit, braucht also keine Vorberechnung. Die zweite Methode hingegen hat einen wichtigen Vorberechnungsschritt, der bei der Darstellung sowohl die Geschwindigkeit als auch die Qualität signifikant erhöht.

Der erste Teil der Dissertation beschäftigt sich mit einem Algorithmus zur Darstellung von Szenen mit starker gegenseitiger Verdeckung von Objekten. Dabei kommen in mehreren Schritten bildbasierte Renderingmethoden zum Einsatz. Objekte bis zu einer bestimmten Entfernung vom Betrachter werden mit gewöhnlichen polygonbasierten Methoden gezeichnet. In einem weiteren pixelbasierten Schritt werden dann alle noch nicht bedeckten Pixel des Bildes identifiziert und in einem zylindrischen Zwischenspeicher für Farbwerte nachgesehen. Sollte dort kein sinnvoller Wert vorhanden sein, wird die Farbe des Pixels mittels eines Blickstrahls ermittelt, sofern sich das Pixel nicht über dem Horizont befindet. Die Methode funktioniert praktisch unabhängig von der Anzahl der verwendeten Objekte in der Szene und erreicht eine bis zu zehnfache Beschleunigung im Vergleich zu üblichen Darstellungsmethoden.

Im zweiten Teil der Dissertation wird eine Datenstruktur zur getrennten Speicherung von Geometrie- und Farbinformationen für ein Objekt präsentiert, geeignet für die Betrachtung aus einem bestimmten räumlich abgegrenzten Bereich. Damit sollen komplexe Objekte in virtuellen Szenen – insbesonders weit entfernte Objekte – ersetzt werden, um eine schnellere und qualitativ bessere Darstellung dieser Objekte zu erreichen. Dabei wird das Objekt quasi mit einer Punktwolke

dargestellt, deren Dichte sich nach den möglichen Betrachterpositionen richtet. Das Aussehen der Punktwolke wird mittels eines Monte Carlo Verfahrens bestimmt, das eine artefaktfreie Darstellung von allen erlaubten Blickpunkten aus gestattet.

Außerdem gibt diese Dissertation einen ausführlichen Überblick über schon publizierte Methoden im Bereich der Echtzeitvisualisierung, und enthält eine Analyse über Vor- und Nachteile von bild- und punktbasierten Renderingmethoden für die Darstellung von entfernten Objekten.

Acknowledgements

This thesis would not have been possible without the help of many people: I would like to thank my advisor, Michael Gervautz, for bringing me into the wonderful research group at Vienna. I owe thanks to Dieter Schmalstieg, who introduced me during many discussions to the peculiarities of scientific research, and who supervised the first part of this thesis. The second part of the thesis was supervised by François Sillion, who never failed to bring the work on track in critical situations.

Many of the ideas leading to the first part of the thesis are directly attributable to Markus Giegl, who also kindly let me use his ArsCreat programming framework to implement the method. I equally want to thank his wife, Gabi, for designing a test model for this part of the work.

The second part of this thesis is the direct result of endless discussions, throwing around ideas, filling one drawing board after the other, and generally working together, with my colleague Peter Wonka. He also provided the test model for the walkthroughs shown in the second part of the thesis. This model also profited greatly from the dedicated work of our students: thanks to Gregor Lehninger and Christian Petzer for tree models, Paul Schroffenegger for an automatic roof generator, and Gerald Hummel for providing a street generation tool.

Finally, I wish to thank all people of the research group in iMAGIS, Grenoble, where I spent six months working on the thesis, and the research group in Vienna, where I spent the rest of my time.

Part of this research was supported by the Austrian Science Fund (FWF) contract no. P13867-INF and P11392-MAT, and by the EU Training and Mobility of Researchers network (TMR FMRX-CT96-0036) “Platform for Animation and Virtual Reality”.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The evolution of 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 effects and whole image sequences in the movie industry.

Yet, many users did not content themselves with single images which took in the range of minutes or even hours to compute. Many application areas called for interactive image generation. Early systems were very plain, presenting the user with wireframe views of a scene, barely sufficient to form a notion of the spatial relationship of objects. The potential of interactive techniques was soon discovered, however, and a huge effort was dedicated to creating systems of both higher display speed and higher image quality. A new research direction was born: “real-time rendering”. Flight simulation, driving simulation, architectural walkthroughs, interactive modeling packages and virtual reality were among the applications that benefited from or even were inspired by the advances made in real-time rendering.

Yet as interactive graphics systems got faster and faster, the expectations and requirements of such systems grew at an even faster pace. While early applications offered manipulation of some simple objects, it was soon expected to see terrain flythroughs and car simulations with considerably increased complexity, while today’s applications try to display full-fledged city models with realistic lighting and full real-time, interactive car simulations.

In recent years, the popularity of three-dimensional computer games has given an unexpected boost to consumer graphics hardware, starting with the introduction of the 3DFX Voodoo graphics card in 1997 [Leht00]. Relatively cheap graphic boards are now threatening to overtake expensive workstation hardware in speed as well as in quality. Practically all newly sold personal computers come with considerable graphics capabilities.

While recently introduced graphics boards promise to achieve near photorealistic image generation in real-time, the sheer complexity and size of today's models still tends to be overwhelming for contemporary hardware, and a lot of research has been invested into finding ways to deal with that complexity on a higher level, making use of a priori knowledge about the given models.

This thesis is about one such approach, which is apt to reduce the rendering time for very complex models, while at the same time improving their visual quality.

1.2 Problem statement

Interactive computer graphics applications are used in many fields, but most of them can be summarized in three categories:

- Indoor exploration: the user navigates inside a building or similar closed environment, like a cave system.
- Outdoor exploration: the user walks, drives or flies over an outdoor environment like a city or a terrain.
- Object manipulation: the user interactively manipulates a virtual object like a prototype of a car or a manufacturing part.

Of these three categories, the first two offer an interesting challenge: while the requirements on display speed and image quality are very stringent, indoor and outdoor scenes feature practically unbound complexity, making it very difficult to meet such requirements. Even a very modest city model, for example, where individual buildings consist of not more than a few hundred triangles, total several million triangles. Walkthrough and simulation applications, however, usually demand frame rates high enough to match the display update rate (between 60 and 85 Hz). Modern graphics accelerators may display several million triangles per second, but this does not suffice to display large models at an update rate of

60 times per second, and models have not reached their peak of complexity for long.

Besides performance, large environments pose another problem: distant objects, although of the same complexity as foreground objects, are perceived on a much smaller area on the screen as foreground objects. Therefore, the screen resolution is not sufficient to display distant objects in their full detail. Current rendering algorithms have to subsample distant objects, leading to spatial and temporal aliasing. This creates a noticeable image quality problem.

Fortunately, models in one of the mentioned categories are structured and share some interesting aspects: due to the nature of perspective (which makes near objects appear large and far objects appear small) and occlusion (which causes many objects to be hidden by other objects in any given view) only a very small number of objects account for covering most of the pixels on the screen.

The problem treated in this thesis is how to exploit this particular property of virtual environments to improve both frame rates and image quality.

1.3 Thesis statement

This research presents a set of algorithms to exploit the specific structure of many virtual environments in order to obtain higher frame rates and better image quality.

The central thesis of this research is that

> *It is worthwhile to treat distant objects using different representations and rendering algorithms than foreground objects. In this way, distant objects can be rendered faster and in higher quality than with traditional rendering.*

1.4 Proposed solutions and chapter layout

This dissertation presents two conceptually different approaches to the problem. The first approach makes direct use of the observation that near objects cover a large part of the screen. It directly computes color values for screen pixels that have not yet been covered by near objects. The second approach computes an alternative representation for distant objects based on points, which allows for fast, high-quality rendering.

As a prerequisite for the work presented in this thesis, chapter 2 discusses the state of the art in the field of real-time rendering. This chapter also contains

an overview of other related work in as much as it is used in the thesis, mainly sampling theory and ray tracing. Chapter 3 introduces some terms and notation used throughout the rest of the thesis. The first approach is treated in chapter 4, thoroughly describing the different algorithms used and showing results on a city walkthrough. Chapter 5 is dedicated to the second approach, containing among others a mathematical treatment of points for common point rendering algorithms. Conclusions are drawn in chapter 6, along with a discussion of future work that could be spawned by this thesis.

1.5 Contributions

This dissertation contributes to the field of computer graphics a family of algorithms useful for rendering large virtual environments.

Most notably, chapter 4 is based upon

- an algorithm to partition a scene from a specific viewpoint into an area near the viewpoint and an area far from the viewpoint, and a method to assign individual pixels to the one or the other area,
- a panoramic image cache to preserve pixel color values already calculated in a previous frame,
- an algorithm to find the horizon of a scene from a specific viewpoint, significantly reducing the number of pixels which have to be calculated, and
- a working system which combines the three algorithms above with a fast ray caster to quickly calculate contributions from pixels identified as being in the area far from the viewpoint.

In chapter 5, the significant contributions include

- an object-space resampling method that distributes sampling of parts of a scene with respect to prospective viewing locations,
- a mathematical analysis of the meaning of a point in the light of current point rendering algorithms,
- a fast rendering algorithm for points that takes into account the mathematical properties of a point mentioned before, and which runs fully accelerated on current graphics hardware, and

- a system which demonstrates the feasibility of the approach and a discussion of the practical implementation of such a system.

Furthermore, this work contains a comprehensive overview of research in real-time rendering and a discussion of the most important methods and how they relate to the algorithms described in this thesis.

Substantial evidence will be provided supporting that the techniques presented in this thesis are feasible, offer significant performance gains and improvements in image quality.

1.6 Motivations

While the development of graphics systems shows dramatic increases in performance and image quality over the last years, their power is still far from being able to render large virtual environments without the help of specialized algorithms.

The idea for the method presented in chapter 4 arose from the need to render large outdoor environments on low- and medium-end computer systems, as for example available for computer games. One requirement was not having to rely on hardware features unavailable in mainstream systems, like reading back the frame buffer quickly. The method should be simple, and should not require long preprocessing or large additional storage space. Ray casting fit very well into these requirements, and the other algorithms used in addition were found to provide the necessary operating speed.

The method in chapter 5 was inspired by the author's previous work in visibility, where sets of objects potentially visible from specific regions of space were precalculated to reduce the number of objects to be rendered from that region. In many cases, the reduction obtained was not sufficient to allow for real-time rendering, leading to the necessity of representing distant objects visible from this region in a more efficient way. Investigations into the mathematical properties of several different representations lead to the conviction that image quality was just as important as speed when designing such representations.

1.7 Individual publications about this work

Results of this work have been previously published by the author. The following papers describe the preliminary outcome of the work [Wimm99a, Wimm99b, Wimm01]:

- 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. Springer Verlag-Wien, June 1999.

- Republished as an extended version in *Computers & Graphics*, 23(6):831–838, December 1999. ISSN 0097-8493.

- Michael Wimmer, Peter Wonka, and François Sillion. Point-Based Impostors for Real-Time Visualization. In Karl Myszkowski and Steven J. Gortler, editors, *Rendering Techniques 2001 (Proceedings of the EUROGRAPHICS Workshop on Rendering 2001)*. Eurographics, Springer-Verlag Wien New York, June 2001.

The author has also participated in projects closely related to this thesis, and contributed to the following publications [Wonk00, Wonk01]:

- 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 of the EUROGRAPHICS Workshop on Rendering 2000)*, pages 71-82. Eurographics, Springer-Verlag Wien New York, June 2000. ISBN 3-211-83535-0.

- Peter Wonka, Michael Wimmer, and François Sillion. Instant Visibility. *Computer Graphics Forum (Proc. EUROGRAPHICS 2001)*, 20(3), September 2001.

Chapter 2

Related Work

This chapter contains a comprehensive overview of research related to this thesis. The field of real-time rendering will be given special attention, including traditional acceleration techniques like state sorting, levels of detail and visibility culling. As an alternative to traditional techniques, image-based rendering has received a lot of attention in recent years, and will be discussed with a view to chapter 5. The same chapter will profit from an analysis of prior point-based rendering algorithms, and a discussion of some mathematical tools needed later on. Chapter 4 will make use of ray tracing, especially acceleration structures for finding the first intersection.

2.1 Definition and goals of real-time rendering

Real-time rendering is concerned with the display of computer-generated images at rates which let a human observer believe that he is looking at a smooth animation. Ideally, we would like to produce those images in a way that matches or exceeds the limits of the human visual system in all aspects. Unfortunately, limits in the way current graphics generators are built, both with respect to their physical characteristics as well as to their raw performance, make this goal unattainable [Helm94].

Nevertheless, attempts to reduce the gap between the degree of realism that a computer system can generate and that a human user can perceive have been manifold. Before elaborating on some of these attempts in more detail, it is important to define the parameters of computer-generated scenery that determine the degree of realism perceived by the user.

Image resolution (measured in the number of horizontal and vertical pixels) is

usually referred to as the main factor influencing the quality of an image displayed on a computer screen. At the lower end, we find arcade and console systems designed for standard TV screens, which offer output resolutions of 640x480 pixels (NTSC) or 720x576 pixels (PAL). Personal computers are usually equipped with 17” screens, allowing up to 1280x1024 pixels (which is also the usual upper limit for LCD displays), whereas professional users typically work on 21” screens set to 1600x1200 pixels. At the very high end, we can sporadically find displays capable of showing 2048x1536 pixels, but neither displays nor graphics hardware are usually capable of displaying this resolution with good quality.

Such displays have practically reached the limits of detail the human eye can resolve. For example, one pixel on the center of a 21” screen (which is 406 mm wide) driven at 1600x1200 pixels subtends an angle of only about 1.4 (horizontal) arc minutes at the eye when viewed from a distance of 60 cm. The *visual acuity* of the human eye, i.e., the degree to which visible features can be perceived (usually measured as the angle subtended at the eye), is about 1 arc minute at the center of the fovea, and deteriorates rapidly towards the outer regions [Helm94].

A factor which is frequently overlooked but equally important is image quality in a signal theoretic sense. Images displayed on a computer screen are signals that have gone through a variety of signal transformation, most importantly sampling of a continuous signal to discrete pixel values and reconstruction of a continuous signal from these pixel values. Signal theory states that signals have to be properly band-limited (“smoothed”) before sampling or resampling so as to allow perfect reconstruction [Glas95]. Not properly band-limited signals manifest themselves in artifacts known as spatial and temporal *aliasing*, for example the well-known jaggies, staircase and moiré effects. Spatial aliasing effects are often most noticeable during motion, leading to pixel blinking and flickering and a generally noisy appearance of the image as pixels erratically change their colors. Remedies for this problem include mip-mapped texture filtering and super sampling, which will be discussed along with a more mathematical treatment of aliasing effects later on.

Temporal aspects are equally important for image quality. *Latency*, for instance, is the time measured from the setting of an input until the corresponding output is obtained. In real-time rendering, this is usually the time when the last pixel of the frame containing the result of the user action is displayed (in this case, the term is visual latency). Latency is usually an issue when multiple processors are used to achieve higher frame rates through concurrency [Jone94].

One of the most important figures to judge the quality of a real-time system by is its *frame rate*. A sufficiently high frame rate is necessary to fool the human eye into seeing a smooth and continuous motion. 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].

The goal of any real-time rendering system is therefore to maintain a frame rate which matches the display update rate—a difficult task, given that the frame rate is bound to the scene complexity in traditional rendering systems. After introducing the basic rendering pipeline which has been standard for years now, the rest of this chapter will introduce techniques invented to achieve this goal.

2.2 The traditional rendering pipeline and standard acceleration techniques

Practically all current real-time rendering systems are based on a polygonal rendering pipeline. This pipeline consists of the following stages:

- Traversal: the application traverses its internal data structures and produces the polygons to be sent to the next stage
- Geometry: Vertices of polygons are transformed into screen space and lighting calculations are performed on the vertices. This stage also includes other per-vertex operations such as the calculation of texture coordinates for environment mapping.
- Rasterization and pixel processing: color values are generated for all pixels inside a polygon and stored in the frame buffer. Pixel processing involves operations such as depth buffer testing, antialiasing, texturing, and alpha blending. Most of these operations require access to memory (e.g. for looking up texture values), which can have great impact on the performance.

While traversal always happens on the host CPU, all other stages are carried out by graphics hardware on current systems. Some recently released graphics cards offer fully programmable vertex and pixel processing units, paving the way for near-photorealistic rendering in real-time [Corp01]. The overall speed of the pipeline depends on its slowest stage (the bottleneck of the pipeline). Keeping all pipeline stages equally saturated is the key to a balanced rendering system.

To make optimal use of available resources, the peculiarities of the target graphics architecture have to be known. Most architectures are optimized to process triangle strips in order to minimize the number of vertices that have to be transformed. Recent cards keep a number of transformed and lit vertices in a vertex cache, making it important to order vertices so as to maximize the number of cache hits.

Changing the rendering mode (for example, switching the texture) usually involves a performance penalty, so presorting primitives according to their type and rendering mode is crucial to avoid costly state changes in the graphics hardware.

Many scene attributes do not change over time, and calculating their effects beforehand avoids runtime calculations (for example, static lights on static geometry, or transformations that are important for modeling, but do not change later on).

Other runtime optimizations include precompiling rendering commands into so-called display lists, making good use of the cache hierarchy (for example by storing static geometry on the graphics card memory) and optimizing texture layout. Applications with significant computation load beside database traversal profit from concurrency between the graphics card and the host CPU, if properly utilized.

The optimizations presented here need not always be implemented by hand. Many of them are readily available in commercial rendering toolkits, most notably the SGI OpenGL Performer and Optimizer programming libraries [Rohl94, Ecke00, Ecke98]. While Performer concentrates on high-speed rendering using multiprocessing and parallel culling strategies, Optimizer offers high-level optimization tools for surface simplification and occlusion culling (see the following sections).

Note that although the above-mentioned optimizations result in good utilization of available resources and are an often-overlooked prerequisite for real-time rendering, they can guarantee sufficient frame rates only up to a certain degree of scene complexity. More complex virtual environments require more sophisticated techniques, all of which aim to reduce the number of geometric primitives which have to be passed to the graphics subsystem in the first place. The most prominent techniques in this class are visibility culling (section 2.2.1) and level-of-detail

rendering (section 2.2.2).

2.2.1 Visibility Culling

An important challenge of early computer graphics was hidden surface removal, the elimination of all parts of surfaces not visible from a specific viewpoint. Several solutions have been proposed to this classical problem (later on referred to as the visibility problem), of which nowadays only the z-buffer [Catm75] is used in real-time graphics because of its simple hardware implementation. In order to render scenes larger than the limits imposed by the number and size of geometric primitives that a hardware z-buffer can handle in real-time, the focus of research efforts has changed to resolving visibility on a coarser scale. The goal is to cull surfaces, whole objects or even larger parts of the scene that are not visible from the current viewpoint, with backface culling being a trivial example. Most methods are conservative, meaning that they only cull objects that are guaranteed to be invisible. The result of a visibility culling algorithm is usually called the potentially visible set (PVS) [Aire90], because it is composed of a list of objects which are potentially visible. This list is passed on to the graphics hardware, where final hidden surface removal is done.

There is a natural distinction between algorithms that try to solve visibility during runtime and algorithms that do some preprocessing on the scene and determine visibility beforehand. Runtime algorithms almost invariably operate on a per-frame basis, i.e., each frame they calculate a PVS for the current viewpoint.

A very simple but effective example for online visibility culling is view-frustum culling, which discards all objects not contained in the current viewing frustum. This form of visibility culling, or its more advanced variant, hierarchical view-frustum culling [Clar76], is implemented in practically all current real-time rendering systems. As is valid for all visibility culling algorithms, a thoughtful spatial organization of the scene based on a grouping of nearby objects is beneficial to culling efficiency. Objects may be grouped in a spatial hierarchy such as an octree [Same89] or a binary space partition (BSP) tree [Fuch80], or in a bounding volume hierarchy determined in the modeling phase.

Certain densely occluded scenes can be structured into a set of cells connected by portals. Such a subdivision comes most naturally for indoor scenes like building interiors, where individual rooms and hallways constitute the cells and doors and windows make up the portals. After calculating the cell adjacency graph, Teller and Sequin [Tell91] precompute cell to cell and cell to object visibility using an analytic method and linear programming, which is quite time consuming. Luebke and Georges [Lueb95] proposed a method that works without preprocessing:

during the traversal of the cell adjacency graph, they accumulate the intersection of projected screen-space bounding boxes of the portals until this intersection is empty. Objects in each cell are tested by intersecting their screen-space bounding box with the aggregate intersection of the current portal sequence.

Most *online occlusion culling* algorithms build an occlusion data structure in object space or image space, and test the scene hierarchically against this structure. Building and testing can be interleaved, adding visible objects to the current occlusion data structure after testing them, or separate, building the occlusion structure first and testing all objects against the complete structure. The occlusion data structure is often maintained in image space, which reduces the complexity of visibility computation by reducing the dimensionality of the problem. Another way to handle complexity is by discretization of the domain. This is straightforward for image space methods where the projection plane is discretized into pixels, but can be applied likewise to object-space methods for example by discretizing the scene into an octree.

The hierarchical z-buffer (HZB) [Gree93] is an extension to the classical z-buffer in which the z-buffer is maintained as a quadtree (it is therefore an image space method using discretization). The upper levels of the hierarchy store the maximum of the z-values of its children. The scene is also maintained hierarchically in an octree. If an octree node is found visible during front to back traversal, its children are traversed, or, if it is a leaf node, its stored geometry is rendered into the hierarchical z-buffer. In practice, hardware implementations are not available and only low resolution variants of the z-buffer can be used because of the rasterization overhead for higher resolutions. This drawback is partially overcome by hierarchical occlusion maps (HOMs) [Zhan97]: opacity values are used instead of depth, so occlusion maps can be rendered with graphics hardware. However, creating the hierarchy and reading back the results from the frame buffer is still costly. In the absence of depth information, the method relies on a good selection of occluders, which have to be guaranteed to lie in front of the objects to be tested for visibility. Both algorithms work well for arbitrary environments, including scenes which are not densely occluded.

Although projection into the current viewing frustum (as in the HZB and HOM techniques) is very general, it raises problems because the size of the occlusion image is not independent of screen resolution. In a large class of scenes, occlusion is mainly due to objects connected to the ground, as for example in an urban environment, where buildings are the most important occluders. Wonka and Schmalstieg [Wonk99] have proposed to use an orthographic projection from above, i.e., a birds-eye view, to record the shadows cast by such occluders. The projection is organized in a rectangular array of height values, called cull map. Each occluder, given as a function $z = f(x, y)$, creates a shadow volume when seen from

a particular viewpoint. Only the upward-pointing faces of this shadow volume are relevant for occlusion. They are rasterized into the cull map using z-buffered graphics hardware. After reading back the cull map from the frame buffer (which can be a bottleneck on contemporary graphics hardware), the cull map is traversed and objects lower than the depth value recorded in the cull map are discarded. Advantages of the method include the ability to handle a large number of occluders, occlusion due to the interaction of several occluders (occluder fusion) and the fact that cull map resolution is independent of screen resolution, which allows tuning the algorithm to the properties of the available graphics hardware.

Several authors have proposed object-space methods for online occlusion culling which rely on geometric computations. As the computational complexity precludes a large number of occluders, such methods typically rely on selecting a small number of large occluders according to a metric based on the projected screen area. Hudson et al. [Huds97] test a bounding volume hierarchy of the scene against the individual shadow frusta of occluders. Coorg and Teller [Coor97, Coor99] traverse an octree, testing each node for occlusion using supporting and separating planes. They also incorporate temporal coherence in their framework. Finally, Bittner [Bitt98] presented an algorithm based on a shadow volume BSP tree for occluders.

While many of the online occlusion culling algorithms presented here can significantly reduce the total frame rendering time, they all suffer from the problem that the time required to calculate visibility reduces the time available to render geometry. If the result of visibility computations were available beforehand, it would only have to be looked up somewhere at runtime and the entire frame time would be available to render geometry. Naturally, it is impossible to precompute a PVS for every possible viewpoint (there are infinitely many). In recent times, researchers have therefore focused on visibility culling for a region of space, typically called *view cell*. The idea is to calculate a PVS which contains all objects that are potentially visible from any viewpoint in the view cell. The scene is partitioned into view cells so that all possible observer positions are contained in at least one view cell. During runtime, it is only necessary to determine the view cell of the observer, and display the appropriate PVS, which has been calculated offline.

Unfortunately, calculating visibility for a region of space is not straightforward. There are striking similarities to shadow rendering, where it is significantly more difficult to find the shadow due to an area or volumetric light source as opposed to a point light source. Indeed, it turns out that finding the PVS with respect to a view cell is equivalent to finding all objects not totally contained in an umbra region if the view cell is considered as a volumetric light source.

The complexity of the problem was first made apparent in the context of aspect graph computation, a data structure which encodes all qualitatively different views of an object [Gigu91, Plan90]. Later, Teller [Tell92] formulated the problem in line space to compute potentially visible sets for scenes made up of cells and portals, and Durand et al. [Dura97] presented the visibility skeleton, a data structure which compactly encodes all visibility relationships in a scene. All authors emphasize that region visibility computations are complex in part because the object of primary interest, the umbra, is bounded by curved quadratic surfaces. Such surfaces arise when edges of the view cell interact with edges of occluders. Another noteworthy complication is that there is no obvious way to consider occlusion due to multiple occluders. The union of umbrae is often negligible, whereas the umbra of the union of the occluders would be very considerable. The challenge in designing algorithms for region visibility is therefore to best approximate the umbra region due to the view cell and all occluders. While early attempts to calculate region visibility for real-time graphics applications ignored the problem altogether (considering only strong occlusion, i.e., occlusion due to one convex occluder [Cohe98b]), recent publications show several different ways to increase the total umbra region considered.

Schaufler et al. [Scha00] create an octree discretization of the scene. The model is assumed to be watertight, which enables them to find the interior using a floodfill-like algorithm. During umbra calculation, interior (opaque) nodes and nodes in umbra are considered equivalent. The unique feature of the algorithm is that before the umbra of an occluder (i.e., an opaque node) is calculated, the occluder is extended into nearby opaque or umbra nodes in order to create a larger umbra region. This effectively allows occluder fusion by combining an occluder with an existing umbra, thereby creating a larger joint umbra. The algorithm is robust and runs on arbitrary scenes, provided they are watertight.

Durand et al. [Dura00] proposed extended projections as an extension of the hierarchical z-buffer for region visibility. The extended projection of an occluder is defined to be the intersection of all possible projections of the occluder onto an occlusion map when seen from a point within the view cell, while the extended projection of an occludee is the union. They show how to calculate the extended projection for some classes of objects and accelerate rendering the projections using graphics hardware. Occluder fusion is achieved by reprojecting the occlusion map several times farther away using image space convolution, rendering occluders into a plane where its extended projection is maximal. Essentially, this allows combining the umbrae of different occluders if the umbrae overlap in any of the reprojected image planes.

Finally, Wonka, Wimmer and Schmalstieg [Wonk00] have presented an algorithm based on the idea of occluder shrinking. They observe that the umbra of

an occluder with respect to a point conservatively represents its occlusion with respect to an ε-neighborhood around that point as long as the occluder is shrunk by ε before calculating its umbra. An occlusion culling algorithm based on this observation covers the view cell with point samples such that each point on the boundary is contained within the ε-neighborhood of at least one point sample. The region visibility problem is now reduced to a point visibility problem: an object is considered occluded if it is occluded with respect to all point samples, using shrunk occluders. The authors apply the method to the occluder shadow algorithm they presented earlier, and show how to efficiently combine the cull maps from each point sample using graphics hardware. They stress that their method is the only practical region visibility algorithm published so far that can deal with arbitrary occluder interactions. Their approach discretizes umbra boundaries, even if they are made up of curved surfaces (as mentioned above), and arguably provides the best approximation to the real umbra region.

Although region visibility algorithms provide better frame rates than online algorithms because they incur no runtime overhead, this advantage comes with a cost. The involved calculations are inherently complex and precomputation can run several hours for larger scenes with a big number of view cells. The results of visibility computations have to be stored and can take up considerable storage space. It is also not clear how to select a partition of the scene into view cells. Large view cells minimize storage space and allow the PVS to be transmitted over a network, for example, but smaller view cells provide for better occlusion, i.e., smaller PVSs. A future avenue of research is to calculate region visibility on the fly and amortize the cost for visibility calculations over several frames. A first step in this direction is the Instant Visibility system proposed by Wonka, Wimmer and Sillion [Wonk01], in which region visibility is calculated during runtime on a different computing resource, such as a computer connected through a local area network. The system allows the display host to run at full frame rates, while still providing a tight PVS.

Note that a more comprehensive and more general overview of visibility algorithms can be found in Durand's PhD Dissertation [Dura99]. A very critical review of many recent visibility culling techniques is contained in the reference manual for the Umbra system [Aila01], a commercial online occlusion culling system based loosely on the hierarchical z-buffer approach.

2.2.2 Levels of Detail

Visibility calculations reduce geometric complexity by eliminating objects not visible on the screen. In large virtual environments, however, many visible ob-

jects are very small or distant. The size of the geometric features of such objects often falls below the perception threshold or the size of a pixel on screen. To better utilize the effort put into rendering small details and to improve frame rates, simpler versions—commonly called levels of detail (LOD)—of an object can be prepared. The technique has first been utilized by Clark in 1976 [Clar76], and has been an important research topic ever since. Several questions have to be answered in order to apply levels of detail in an application: How to select an appropriate level of detail? How to stage the transition between two levels of detail? And, most importantly, how to generate levels of detail for a given model?

To tackle the question of LOD *selection*, heuristics are used because human perception and aesthetics are hard to catch in a single formula. Most commonly, the distance of the object from the observer or the size projected to the screen is used as a measure for the LOD. Unfortunately, these static heuristics do not adapt to variable load on the graphics pipeline. Reactive level-of-detail selection adaptively selects levels of detail according to the rendering time required by recent frames. To guarantee bounded frame rates and accommodate sudden changes in the rendering load, Funkhouser and Sequin proposed a predictive selection algorithm [Funk93] formulated as a cost/benefit optimization problem. Levels of detail are selected to produce the best image while keeping the cost for rendering all selected objects below the maximum capacity of the rendering system at the desired frame rate. The cost heuristic is based on the polygon and pixel capacity of the rendering system, and the benefit heuristic takes into account factors such as the size, accuracy, and importance of the object. The optimization problem is a variant of the knapsack problem and can be solved approximately with tractable computation effort for every frame.

Staging the *transition* between two successive LOD representations can most easily be done by hard switching: At some point, the simpler model replaces the more complex model. To reduce the resulting visual popping, for a short transition period the representations can be drawn blended together (which temporarily increases the rendering load). Best quality is achieved by geometrically morphing one object into another, but this requires levels of detail with well-defined geometric correspondences.

The question which has attracted most interest, however, is how to automatically *generate* simplified versions of a model from a detailed model while trying to preserve appearance. Important aspects in the classification of algorithms include whether they work on local features such as a vertex and its surrounding polygons, or on a global level, and what error bounds are used to control the quality of the resulting simplifications.

Vertex clustering algorithms ignore topology in input data and perform ro-

bustly even for degenerate data. The number of vertices in a polygonal model is reduced by creating clusters of vertices close to each other and replacing cluster members by a representative vertex. In the course of that process, degenerate triangles are replaced by lines or points respectively. Several selection criteria have been presented to choose the vertices to be clustered. Rossignac and Borrel [Ross93] propose a simple, yet efficient uniform quantization in 3D. Schaufler and Stürzlinger [Scha95b] and Luebke and Erikson [Lueb97] independently developed hierarchical clustering methods. Vertex clustering algorithms can achieve arbitrarily high compression, but also exhibit severe artifacts for higher compression rations, and local features are not preserved well.

Most *mesh simplification* algorithms perform local operations on the surface of the object, with an emphasis on the preservation of important visual features such as shape and topology. They usually expect topologically sound, manifold input meshes. This criterion is often not met by models generated with CAD packages, and also limits the simplification ratio by the requirement of not reducing the genus of the object.

The vertex decimation algorithm by Schroeder et al. [Schr92] analyzes the vertices of the original model for possible removal based upon a distance criterion. A local retriangulation scheme is then used to fill the hole resulting from the removed vertex. Turk's retiling method [Turk92] optimizes a triangle mesh by uniformly distributing new vertices on the surface of the original object. The original vertices are then iteratively removed, and the surface is locally retriangulated to best match the local connectivity of the surface. Hoppe et al. [Hopp93] introduced the concept of an energy function to model the opposing factors of polygon reduction and similarity to the original geometry. The energy function, used to provide a measure of the deviation between the original and the simplified mesh, is minimized to find an optimal distribution of vertices for any particular instantiation of the energy function.

Progressive algorithms represent the original object by a series of approximations that allow a near-continuous reconstruction and can be encoded incrementally in a very compact way. Lounsbery et al. [Loun97] use wavelets to represent polygonal objects by a multi-resolution data set of wavelet coefficients obtained from a triangular mesh with subdivision connectivity. Levels of detail can easily be constructed by omitting higher order detail coefficients in the reconstruction process. Eck et al. [Eck95] present a method to transform an arbitrary mesh into an equivalent one with the required subdivision connectivity. This work is taken further by Certain et al. [Cert96] to include colored meshes and support progressive reconstruction of the model.

Several authors present methods based on the edge-collapse operation. The

representation is generated as a sequence of repeated edge collapses, and is simply inverted in the progressive reconstruction process. The essential difference between these algorithms lies in how they choose an edge to collapse. The progressive meshes introduced by Hoppe [Hopp96] adopt the energy function introduced earlier [Hopp93]. Ronford and Rossignac [Ronf96] track the planes of the polygons adjacent to a vertex throughout the decimation process, and base the decision on a metric dependent on the distance to these planes. Garland and Heckbert [Garl97] generalize edge collapses for arbitrary models by allowing contraction of vertices not common to an edge. Consequently, their algorithm allows topological changes to the model and therefore achieves superior compression rates. They also introduce a very efficient and high-quality method to choose vertex pairs for contraction based on a quadric error metric. Their method is one of the most widely used today, also because source code is available on the web.

View-dependent simplification methods exploit the fact that during runtime, edge collapses and its inverse operation, vertex splits, do not necessarily have to be executed in the same order as used in their creation. Simplifications on different parts of the model may be completely independent and can be executed in any desired order. Instead of storing the progressive mesh as a sequence of vertex splits, Hoppe [Hopp97] orders the vertex splits in a tree based on the dependencies between the individual splits. Any front through this tree represents a valid simplification of the original model. View-dependent criteria such as silhouette preservation (based on the normal vector), back facing surfaces and surfaces outside the viewing frustum are used to determine which parts of the model to refine and which parts to simplify. A similar idea was proposed by Xia and Varshney [Xia96], and the whole process was formalized in a theoretical framework by De Floriani et al. [De F97].

Most of the methods presented here work purely on the geometrical structure of the model. Some algorithms take precautions to preserve other attributes of the mesh such as discrete values like material identifiers, or scalar values like texture coordinates and color values [Garl98, Hopp96]. This idea is taken further in the appearance preserving simplification method [Cohe98a], where textures and bump maps are actually used to represent small object features in simplified models. The method requires a parameterization of the model, which is not always readily available. Recently, Lindstrom and Turk showed a novel approach to simplification [Lind00]. They base the decision where to apply edge collapses on a comparison of images of the simplified and the original model. The simplified models are close to the original models according to image differences as well as geometrically. Note that their article also contains a critical review of many recent level-of-detail approaches.

Many specific algorithms have been published about both visibility culling and

levels of detail. However, there is not a lot of experience in combining different rendering techniques into a larger system. A commercial system that offers at least geometric simplification and occlusion culling at the same time is SGI OpenGL Optimizer. In an interesting research project, Aliaga et al. [Alia99a] have combined geometric simplification, hierarchical occlusion maps, image-based techniques (impostor meshes, see later) and others to accelerate the walkthrough of a very complex power plant model. They observe that the best combination of algorithms to use is not trivial to find—when combined with other methods, a particular algorithm might even slow down rendering in many situations.

2.3 Image-based rendering

The recent decade has seen a tremendous amount of research being invested into modeling and rendering with images. Image-based rendering methods claim to provide high-quality rendering of highly complex scenes at fast rendering rates. This promise sounds alluring; especially for researchers in traditional real-time graphics, where the struggle for high frame rates is not yet won and photorealistic rendering is still far out of reach. Indeed, many image-based rendering methods warrant a closer scrutiny. However, we observe a trend to use such methods in conjunction with and not as a replacement for traditional computer graphics rendering.

The advantage of using images for rendering rather than geometry-based approaches is that they are easy to acquire, either by using digitized photography or rendered images, with an option to mix both. Furthermore, while the complexity of geometric models can be arbitrary, images per se are of bounded complexity and tend to express complexity only where it actually appears.

Images have been used for several purposes in computer graphics:

- as a simplification for and approximation to complex geometric structures (the most prominent example being texture mapping)
- as databases, storing images for all views allowed in the system
- as full-fledged modeling primitives in systems based solely on images, accompanied with the ability to extract new views from existing ones

Before going into the details about different specific methods, it is instructive to introduce a framework in which all image-based rendering methods can be compared.

2.3.1 The Plenoptic function

The fundamental computation of traditional computer graphics is the simulation of the interaction between light and the objects in a scene. The rendering process simulates the notion of a camera with certain parameters capturing this light onto film. While this process is well described for geometry-based computer graphics, image-based rendering has only recently received a computational framework that can be used to express most of its techniques. Adelson and Bergen [Adel91] assigned the name *plenoptic function* (from the Latin root plenus, meaning complete or full, and optic pertaining to vision) to the set of rays visible from any point in space, at any time, and over any range of wavelengths. In traditional computer-graphics terms, the plenoptic function can be considered as the set of all possible environment maps that can be defined for a given scene. Formally, the plenoptic function can be described as a function

$$Plenoptic(x, \theta, \lambda, t)$$

with values in any (photometric) unit useful to describe intensity values of images. Here, x denotes a three-dimensional vector describing an eye positioned anywhere in space, θ is a two-dimensional orientation (consisting of azimuth and elevation angle), λ is the considered wavelength, and, in the case of a dynamic scene, t gives the time at which to evaluate the function.

Although the complete plenoptic function will hardly ever be available for any scene, it serves well to relate different images of a scene to each other. In fact, an image is nothing else than a subset of the plenoptic function, describing values for a given observer position and time over an interval of orientations and wavelengths.

The plenoptic function is important for image-based rendering because it allows us to interpret the task of generating new views from existing ones as a function reconstruction problem. The existing images provide samples for the plenoptic function, and synthesizing a new view means to locally reconstruct it and derive new samples from it.

Aliasing problems in image-based rendering were previously not fully understood. As the plenoptic function is a continuous function describing all possible images, signal theoretic concepts can now be applied to any method which can be expressed in terms of the plenoptic function framework.

Miller and Bishop [McMi95b] were the first to realize the importance of the plenoptic function for computer graphics in general and image-based rendering in particular. Their plenoptic modeling system emphasizes acquisition of a database

of cylindrical images and the different camera parameters that have to be taken into account to synthesize new views from the database.

Having laid the grounds for a systematic study, this section will deal first with solely image-based methods, presenting hybrid methods (where images are used for acceleration) later on.

2.3.2 Light fields

In 1996, Levoy and Hanrahan [Levo96] and Gortler et al. [Gort96] simultaneously introduced a new image-based primitive that has sparked the interest of researchers ever since: the light field (Gortler called it the lumigraph). The light field is appealing because it represents the first attempt to systematically describe a considerable subset of the plenoptic function. Leaving aside time and wavelength, they observe that 4 degrees of freedom (instead of 5) suffice to describe the plenoptic function if either its domain (for an outward looking light field) or the inverse of its domain (for an inward looking light field) is restricted to a convex region of space that contains no obstacles. Putting aside effects like participating media, the intensity of a light ray does not change when it traverses empty space, so only one value is required to describe this subset of the plenoptic function.

Both articles parameterize rays by their relative coordinates on two planes placed around the object (in the case of an inward looking light field). To capture all rays, three such plane slabs are needed. Conceptually, a light field is acquired by placing a camera on all positions of a regular grid on the first plane (the entry plane) and pointing it at the second plane (the exit plane). Light fields involve huge storage costs, and compression methods based on vector quantization combined with entropy coding have been proposed by Levoy and Hanrahan. Both articles deal with filtering issues. While Levoy and Hanrahan design an aperture filter around a camera analogy, in Gortler et al.'s method the reconstruction filter is given automatically as an integral by the mathematical framework they use. They also use available 3D geometry to improve reconstruction quality.

This is taken even further by Chai et al. [Chai00] in plenoptic sampling, by computing exactly the number of images (i.e., samples on the entry plane) needed for antialiased reconstruction of a light field. The work is based on Fourier analysis and assumes the minimum and maximum depth of the scene is known. It is also based on the assumption of a diffuse scene, which is a major practical limitation. The number of images can be further reduced if more depth information is available. The minimal rendering curve shows for each amount of depth information the corresponding number of images needed for antialiased rendering. A closer scrutiny of the minimal rendering curve reveals that if exact depth informa-

tion is available, only one image is needed to represent any scene. This is caused by a further hidden assumption: no occlusion is allowed between the different depth layers. Although this observation limits the theoretical value of the article, in practice plenoptic sampling still provides valuable insights for the design of light fields that include so many images that occlusion artifacts are negligible.

A different four-dimensional parameterization of the plenoptic function is the surface light field [Mill98, Wood00]. A parameterization for the surface of an object is assumed to be known and replaces the entry plane (in light field terms). The exit plane is replaced by a hemisphere over each point on the object surface. The surface light field can also be interpreted as encoding the exitant radiance in every direction at every point of a surface. Although limited to certain types of objects, the quality of surface light fields is superior to normal light fields because the geometric structure is not implicitly encoded in the light field, but explicitly by the surface of the object. So, while a surface light field is based on the geometric structure of an object and adds sophisticated lighting effects to it, a normal light field tries to capture lighting effects and geometric structure in the same data structure, at the expense of quality.

Numerous other papers have been published on light fields, including their application to store the appearance of complex light sources [Heid98], considerations on depth of field in light field rendering [Isak00], fast rendering methods for lumigraphs [Sloa97, Schi00], and several alternative sampling and compression strategies. However, the huge storage requirements for moderately small objects still limit their practical use.

2.3.3 Quicktime VR and View Interpolation

It is one of the most simple representations of the plenoptic functions that has gained the most widespread use because of its efficiency and availability on a wide range of platforms. The Quicktime VR system [Chen95] basically consists of a sequence of full panoramas. In terms of the plenoptic function, a full panorama is achieved by keeping the viewer position constant and considering all orientations (almost, as only a cylindrical parameterization is used in current Quicktime systems). The user is allowed to rotate and zoom, but movement is discrete between viewpoints.

Chen and Williams [Chen93] present a view interpolation method for three-dimensional computer graphics. It uses linear interpolation between corresponding points to map reference images to a desired viewpoint. In general, this interpolation scheme gives a reasonable approximation to an exact reprojection as long as the change in viewing position is slight. The method bears resemblances

to view morphing [Seit96], which also relies on linear interpolation, and is a special version of the more general image morphing [Wolb90], where corresponding points may be specified through various differing methods.

The earliest attempt to use images in a three-dimensional exploration setting is the movie map system [Lipp80], which does not allow synthesizing new views and basically constitutes an interactive slide show.

2.3.4 Images as approximation, impostors

Most of the methods discussed up to now were solely based on images, which can be provided by real photographs or any computer graphics renderer. Apart from the idea of using real photographs to display more convincing environments, one appeal of image-based rendering is the bounded complexity and near-constant rendering time of images. This has been explored separately by many researchers, and frequently, proposed methods work in conjunction with traditional real-time rendering to speed up frame rates.

The earliest and most prominent example in this category is of course texture mapping [Heck89], which simulates detail by mapping images (often defined using bitmaps) onto flat surfaces. Bilinear filtering, mip-mapping [Will83] and, more recently, anisotropic filtering is available in current rendering hardware to overcome the aliasing artifacts due to perspective warping when mapping the texture to the screen. The flexibility of image textures as three-dimensional computer graphics primitives has since been extended to include small perturbations in surface orientation (bump maps) [Blin78] and approximations to global illumination (environment and shadow mapping) [Blin76, Gree86, Sega92]. All these methods are implemented in hardware on current graphics accelerators.

The fast rendering speed of textures and their visual richness make them candidates to replace complex geometry in virtual environments. Maciel and Shirley [Maci95] introduced the concept of an impostor: An image of an object is used in place of the object itself by rendering it as a texture map onto a single polygon facing the observer. Schaufler extended this concept to the dynamic generation of impostors at runtime using graphics hardware [Scha95a]. For rotationally symmetric objects like trees, the impostor polygon can always be oriented towards the observer—in this case, the impostor is also called a billboard. Subsequently, Schaufler and Stürzlinger [Scha96] and Shade et al. [Shad96] concurrently developed a hierarchical image cache based on impostors to accelerate the rendering of very large polygonal scenes. A hierarchical spatial data structure such as an octree is traversed, and a cache of impostors for each node is created and updated as required by an error metric that decides on the validity of the impostor. The method

requires a large amount of texture storage and graphics hardware that supports fast direct rendering into a texture. It also depends strongly on temporal coherence: if there are substantial changes from one frame to the next, many impostors will have to be rebuilt, slowing down rendering. Aliaga and Lastra [Alia97] combined portal rendering (section 2.2.1) with impostors by replacing geometry behind a portal with one of several textures generated on the fly. Rafferty et al. [Raff97] later reduced the number of textures needed for each portal by warping two pre-computed depth images (instead of only one texture) into the new view.

The first attempt to implement impostors directly in hardware was the Talisman architecture [Torb96]. The concept of a frame buffer is abandoned in favor of small reusable image layers (basically, impostors) that are composed on the fly at full rendering speed. During the composition process, a full affine transformation is applied to the layers to allow translation, rotation and scaling to simulate 3D motion and to approximate perspective distortion. Although a commercial implementation of Talisman never saw the light of day, some of its ideas are implemented in low cost graphics accelerators like the PowerVR [Powe01].

2.3.5 Images as caches

Images can serve as caches for alternative rendering techniques. The method in chapter 4 will make use of a panoramic image cache to reduce ray casting costs for distant objects. The render cache proposed by Walter et al. [Walt99] allows near-interactive rendering, while still using high-quality image generation methods like ray tracing or path tracing. This is achieved by caching radiance values from one view in an image and reprojecting them to the next view, thereby exploiting temporal coherence. Fast sequences where a lot of new information comes into view each frame are handled by progressive refinement, only tracing a subset of the necessary rays and reconstructing an image from available information. The Holodeck presented by Ward and Simmons [Ward99] differs from the render cache by the use of a four-dimensional data structure instead of an image as a ray cache. Simmons and Séquin showed how to use graphics hardware to perform the reprojection of cached radiance values [Simm00]. The radiance values are stored in a triangulation of the unit sphere, but with associated depth values. The triangles can be rendered using graphics hardware, which means that linear interpolation between radiance values comes practically for free through Gouraud shading. However, moving the viewpoint (and consequently, the unit sphere that is always centered on the viewpoint) will lead to overlapping triangles on the unit sphere, making it necessary to update the triangulation between consecutive frames.

Finally, dynamic impostors as in section 2.3.4 can be considered as a form of image cache as well.

2.3.6 Enhancing images with depth

All impostor methods described in section 2.3.4 have one point in common: they represent a three-dimensional object by a planar polygon and discard all associated depth information. This can work for well-separated objects that do not show strong parallax effects. However, scenes that are not so well behaved will be problematical. Consequently, different methods have been proposed to remedy the situation. Most of these methods enhance the images used as impostors with per-pixel depth values, which allows a more realistic reprojection of impostors.

Schaufler addressed the problem of interpenetrating objects by introducing a new primitive called nailboard [Scha97]. Each pixel of an impostor texture is assigned a depth value, and during rendering this depth value is transformed into the eye coordinate system and used to modify the polygon depth value. Thus, the actual object depth values are used for visible surface determination. Nailboards can be rendered completely in hardware on current consumer-class graphics accelerators [Corp01].

Another possible extension to impostors is to add depth information to each pixel and then use the optical flow that would be induced by a camera shift to warp the object into an approximation of the new view [Chen93, McMi95b]. Forward mapping an image pixel by pixel can provide proper parallax, but will result in gaps in the image either due to visibility changes when some portion of the object becomes unoccluded, or when a surface is magnified in the new view. Several authors have proposed solutions to this problem, including gap-filling algorithms [Chan99], backward mapping [Lave94] (entailing an expensive search for each source pixel), or two-pass warping with so-called sprites with depth [Shad98].

For complex geometries, however, occlusion artifacts make a single image with depth inadequate to provide enough information to render the object from a wide range of viewpoints. While a possible solution would be to use multiple images of the object created from different viewpoints, and fill in information lacking in one image with pixels from the other images, such an approach requires storing several images for an object and possibly expensive pixel search operations [McMi97].

In contrast, a layered depth image (LDI), proposed by Shade et al. [Shad98] (and earlier in part by Max [Max96]), stores only one image, but provides several depth layers per pixel. For rendering, the authors rely on MacMillan's ordering

algorithm [McMi95a], which, given input and output camera information, establishes a warping order such that pixels that map to the same location in the output image are guaranteed to arrive in back to front order. This allows for an efficient incremental forward warping algorithm, using variable splat sizes to prevent holes from appearing in the final image. LDIs can be acquired using a conventional ray tracer or by registering multiple photographs of an object and warping them into a common camera view. Like most other representations that add depth to images, LDIs cannot be implemented in current hardware and are therefore of limited use for real-time visualization environments. Another issue with LDIs is that their level of detail (i.e., their sampling rate) is adapted only for the reference camera position. This is remedied in a later paper by a structure called LDI-tree [Chan99], where LDIs are stored at multiple resolutions in an octree-like fashion, albeit at the expense of considerable rendering times. Nevertheless, the LDI is a fundamental object representation, and its use is not limited to real-time rendering.

Oliveira and Bishop [Oliv00] propose a hardware extension to texture mapping, relief texture mapping, destined to augment normal texture maps with depth information. Their method is based upon intelligently rearranging the equation for warping an image with depth information into a new view: in a first step, the image is warped using a highly efficient 1D forward transform, and the resulting texture is mapped onto a polygon using standard texture mapping. The 1D warping functions work in texture space to handle the parallax and visibility changes that result from the 3D shape of the displacement surface.

An alternative to using per-pixel depth values is to add depth in layers. Schaufler [Scha98] observed that current graphics hardware can be used to warp objects with depth values grouped into discrete layers. A complex object is stored in several α-textures, containing the image of the object clipped to a small depth range associated with the depth-layer of the texture. A similar approach was independently developed by Meyer and Neyret [Meye98]. Their interactive volumetric textures equally slice complex objects into layers. The main focus lies on the application to repetitive objects like fur, landscapes or organic tissues.

2.3.7 Impostor meshes

Adding per-pixel depth values to an image and warping the resulting depth image directly does not map well to current graphics hardware. Although special purpose hardware, the warp engine [Pope00], has been created to this end as a proof of concept, it is unlikely that such technology will become mainstream in the near future. Therefore, researchers have tried to exploit the capabilities current graphics hardware supports best, i.e., standard texture mapping and triangle

rendering.

Mark et al. [Mark97] propose a runtime system which converts a depth image (acquired on the fly from a standard renderer) to a fine-grained mesh connecting each group of four pixels via two microtriangles. However, warping only one image will introduce so-called rubbersheet triangles between features that appear connected only in the original image. They propose a heuristic to determine the connectedness of pixels, and claim that composing two depth images along the user's path eliminates practically all occlusion artifacts. Still, custom hardware is required to implement the connectedness calculation and compositing operation.

To date, the only way to add depth information to images in a manner compatible with current graphics hardware is by *precomputation*. Darsa et al. [Dars97] present an interactive exploration system with limited user navigability. In a precomputation step, the system requires depth images placed on cubes around key observer positions. Those depth images are then triangulated as in Mark's method. Additionally, coherence in depth values is exploited by running a triangle simplification algorithm on the resulting mesh for improved runtime performance and reduction in storage. The resulting view will be correct for the key positions and only moderate errors occur if the observer doesn't move too far away. Since key positions have to be placed near each other, only limited observer freedom is possible. The advantage is that real images can be incorporated into the system, provided that depth data is available (e.g., using computer vision methods or range scanners).

In contrast, Sillion et al. [Sill97] propose a framework that makes better use of available triangle rendering capacities. The central concept is to dynamically segment the scene into a local three-dimensional model and a set of impostors used to represent distant scenery. Based on an assumed urban structure of the scene, the segmentation proceeds along streets in a virtual street graph. The impostors placed at the entry and exit of individual streets combine three-dimensional geometry to correctly model large depth discontinuities and parallax, and textures to provide visual detail. Unlike Darsa's method, the initial depth images used for the impostors are simplified using computer vision techniques to find depth discontinuities. The algorithm suffers from rubber-sheet triangles and distorted images, because the impostors have to be valid over a relatively large region of space (a street). Subsequently, Decoret et al. [Deco99] refined the method by making better use of parallax information of individual objects with respect to the view cell. Instead of generating one single impostor, the algorithm places objects into layers. The criterion used is the maximum parallax between two objects as perceived from any point in the validity region of the impostor. The resulting rucksack problem is solved using a heuristic approach.

The impostors described by Sillion and Decoret point to a promising avenue of research. The valid viewing regions for the impostors are limited to edges of the street graph (i.e., the observer can only move on a line), and there are still image quality issues with the method used to triangulate the depth images. In its current form, the system achieves respectable frame rates and allows interactive walkthroughs of large urban models.

2.4 Point-based rendering

A hot discussion topic that has been widely debated in the scientific community is whether triangles constitute the ultimate rendering primitive when it comes to displaying three-dimensional models. Of all the alternative primitives advocated in recent years, points seem to have been the most persistent in sparking the interest of the real-time rendering community, as evidenced by several recent papers. The appeal of points lies in their simplicity: very complex models can be represented with little effort, because topology information (e.g., triangle connectivity) can be completely ignored.

The clear advantage of triangles is that all current graphics accelerators do an excellent job of rendering textured triangles—so much so that mere theoretical advantages of another primitive in some scenes will not suffice to replace triangles. Nevertheless, the way future hardware developments go can never be predicted, and so it is definitely worthwhile to explore the capabilities of alternative primitives. Another interesting avenue lies in using alternative primitives for modeling or scene representation and converting it on the fly to triangles for the purpose of rendering.

The use of points as rendering primitives has a long history in computer graphics. As far back as 1974, Catmull [Catm74] observed that geometric subdivision may ultimately lead to points. Particles were subsequently used for objects that could not be rendered with geometry, such as clouds, explosions and fire. Levoy and Whitted [Levo85] use points to model objects for the special case of continuous, differentiable surfaces. They address the problem of texture filtering in detail.

More recently, Grossman and Dally [Gros98] describe a point-sampled representation for fast rendering of complex objects. Their approach consists of two major steps: sampling (i.e., converting an object to a point-sampled representation), and rendering. A significant contribution of their work is that they describe a minimal sampling criterion for point-sampled representations. A surface is said to be *adequately sampled* (for a specific set of viewing parameters), iff at least one

point is projected into the support of each output pixel under any projection of the surface. They prove the following sufficient condition: A surface is adequately sampled if in every possible projection there exists a Delaunay triangulation with a minimal edgelength smaller than the pixel edgelength. Grossman and Dally proceed to note that such a condition is difficult to guarantee in practice, and show a sampling approach motivated by this condition (and fulfilling it under certain circumstances), which consists of placing a certain number of cameras on a sphere enclosing the object and combining the samples so acquired. Rendering the point samples is achieved by an efficient incremental block-warping method, while visibility computations rely on a hierarchical push-pull algorithm.

The approach presented by Grossman and Dally was significantly extended by Pfister et al. [Pfis00], who introduced the concept of a surface element (surfel): a zero-dimensional n-tuple (i.e., a point) containing shape and shade information, and which locally approximates a surface. They sample geometric models from three sides of a cube into three orthogonal LDIs, called a layered depth cube (LDC [Lisc98]) using ray casting. Knowing that the maximum sidelength of a Delaunay triangulation of the surfels is $\sqrt{3}$ times the sample spacing of the LDC, an adequate sampling resolution can be calculated at least for orthographic projections. Samples with prefiltered texture information are actually stored in a hierarchy of LDCs, the LDC tree. The rendering algorithm, although based on the efficient block-warping algorithm by Grossman and Dally, separates visibility calculations from image reconstruction. Splatting is used only to establish the correct visibility order, whereas each surfel only contributes to one pixel. Any remaining holes are filled by interpolating between available samples.

Surfels achieve remarkable rendering quality on quite complex models. Rendering performance, however, leaves much to be desired. For similar output quality, the authors report rendering times in the range of an OpenGL software only implementation, but using bilinear filtering, which is a notoriously slow rendering path in any OpenGL software implementation. The problem lies with the number of surfels needed to faithfully represent a geometric model. The authors report at least twice as many surfels as triangles. For magnified viewing (higher resolutions), image reconstruction becomes a bottleneck. Another consideration is the motivation to use points in the first place. Points are used to easily represent and render very complex geometries, usually with the argument that rendering triangles is not a good option if there are several triangles per pixel and not the other way round. However, the surfel method implicitly assumes a surface structure that can be locally approximated well by its gradient (where “locally” is the same order of magnitude as a pixel). It might be argued that in such a case, triangles would again be a better solution because triangles are very good at approximating surfaces.

An alternative point rendering system, QSplat, was proposed by Rusinkiewicz and Levoy [Rusi00]. Conceived to display laser range scans of archeological exhibits to a broad audience, the QSplat system emphasizes practicability and efficiency more than rendering quality and theoretical foundation. Starting from a triangular mesh of the point cloud, the algorithm builds a hierarchy of bounding spheres by recursively splitting the set of remaining vertices. Colors and normal vectors are propagated to interior nodes by averaging. For rendering, the hierarchy is traversed until the sphere size approximately matches the output pixel size, in which case the sphere is splat to the screen using one of several different methods: simple points provide best performance, while tangent disks textured with a Gaussian splat achieve best image quality. Although the authors provide little theoretical discussion of point rendering, their system shows that point rendering can be competitive to triangle rendering on current graphics system under certain circumstances (the existence of many uniform geometric details at the pixel resolution is given as one example).

2.5 Ray tracing

Ray tracing is one of the best researched techniques in computer graphics, and it is renowned for producing high-quality images. It simulates geometric optics by tracing rays of light from a virtual viewpoint into an object space. For one ray, the goal is to find the nearest intersection of the ray with an object along the ray. Ray tracing was introduced by Appel [Appe68], however, it was the work of Whitted [Whit80] which made ray tracing popular in computer graphics. The problem is that in its naïve implementation, each ray must be intersected with each primitive in the scene. This approach is feasible only for scenes of modest size, and much research has focused on ways to make this technique more efficient for complex scenes.

Arvo and Kirk [Arvo89] give a very good survey of ray tracing acceleration techniques. They classify acceleration techniques in three categories: faster intersections, fewer rays and generalized rays. Faster intersections are obtained by reducing the intersection costs between a primitive and a ray, or by reducing the number of ray-object intersection tests. This last category contains the most diversity of proposed algorithms. It includes various space subdivision schemes, directional techniques and hierarchical bounding volumes.

2.5.1 Ray casting

The original ray tracing technique is concerned with creating photorealistic images. When the intersection of a camera ray with the closest object is found, a local illumination model is evaluated. The reflective, refractive and light source terms usually found in those models require a huge amount of recursive and light rays to be shot. Fortunately, the full generality of ray tracing will not be needed in this thesis.

The goal will rather be to use rays to simulate other rendering algorithms on a per pixel basis, and only primary rays (from the eye to the first intersection with an object) will be needed for that. Shooting only primary rays is generally referred to as ray casting. As a consequence, it is important to reduce the time to intersect a ray with the scene.

The cost of ray/primitive intersections is not a factor that can be tuned. Optimized intersection routines exist for the main primitives of interest, namely triangles (for scene geometry), and spheres and bounding boxes (for bounding volumes). Möller [Möll97] proposes a ray/triangle intersection routine that is efficient, does not require storing the plane equation of the triangle, and provides the ray-parameter as well as the barycentric coordinates of the intersection point within the triangle, which is useful for texture mapping and illumination calculations. The web page [Möll99a] for the book Real-Time Rendering [Möll99b] contains an up-to-date list of currently known intersection tests between several different primitives.

The main factor determining ray-casting performance is the efficiency with which the number of intersections necessary for a ray can be reduced through a spatial subdivision scheme. Such algorithms can be divided into two classes: uniform and non-uniform.

Uniform subdivision (i.e., regular grids) has the advantage of being easy to implement, and the cost of traversing each element of the regular grid is very small. Regular grids have been first used for ray tracing by Fujimoto et al. [Fuji86]. An alternative efficient grid traversal algorithm has been published by Amanatides and Woo [Aman87]. Unfortunately, traversal performance degrades when there are too many voxels because many empty voxels might be traversed and because of the cost of storage of the voxels. The performance degrades also when there are too few voxels because of the possibility of having a large number of objects to intersect within a single voxel. Moreover, there is not yet any good criterion to determine the optimal grid subdivision for a given scene.

Non-uniform space subdivision can adapt its resolution to the complexity of a scene and therefore it is less sensitive to the problems of uniform space subdi-

vision. Unfortunately, traversing a non-uniform structure is more expensive than traversing a regular grid. Various non-uniform subdivisions have been used, including irregular grids [Giga90], octrees [Glas84], BSP trees [Kapl87] and kD-trees [Fuss88]. Comparisons between uniform and non-uniform techniques can be found in various texts [Subr91, Sung91, Jeva89]. Snyder and Barr [Snyd87] propose to surround each ray with a bounding box, a technique which can be used along with most of these methods.

Directional techniques rely, as the name indicates, on the direction a ray takes. These directions are classified by direction cubes subdivided regularly or adaptively. The cubes can be located at specific point locations as for the light buffer (for secondary rays only) [Hain86], onto surfaces as for ray coherence [Ohta87] or in volumes as for ray classification [Arvo87]. Shaft culling [Hain91] is often used to test a bundle of rays contained in a shaft against object bounding volumes.

Other algorithms use hierarchical bounding volumes to reduce the number of ray/object intersection tests. Rubin and Whitted [Rubi80] were the first to use hierarchies of bounding volumes in ray tracing. Weghorst et al. [Wegh84] and Goldsmith and Salmon [Gold87] study criteria for choosing efficient bounding volumes for ray tracing. Kay and Kajiya [Kay86] use slabs as tighter bounding volumes. However, each of these approaches relies on sorting the intersections with sub-bounding volumes or primitives each time a bounding volume is entered. Additionally, intersection tests with bounding volumes are not cheap, and their number is highly dependent on the scene hierarchy. Often, a modeling hierarchy is used which is not optimal for ray tracing.

The most efficient algorithms proposed to date consist of combinations of uniform and non-uniform subdivisions to alleviate the disadvantages of each structure while trying to benefit from their respective advantages. Adaptive grids [Klim97], hierarchical grids [Caza95] and recursive grids [Jeva89] all use regular grids at different levels of a specific scene hierarchy, which can be made up of a standard spatial subdivision scheme or again of a grid.

One acceleration method proposed for ray casting is the item buffer [Wegh84]: a hardware z-buffer is used to determine the first intersections of primary rays. However, in this thesis we are concerned with speeding up exactly the hardware rendering step, therefore the item buffer is not a viable acceleration technique for the methods presented in this thesis.

2.5.2 Complexity bounds

Some researchers have tried to find a formal expression for the complexity of ray tracing. The naïve ray tracing implementation has a complexity of $O(np)$, where

n is the number of objects and p is the number of rays to be shot (equivalent to the number of pixels in a ray caster). One notable result is that, given a fairly uniform distribution of objects in the scene, ray casting can be shown to have complexity constant in the number of objects ($O(cp)$, with c a constant) when using a regular grid as spatial subdivision scheme [Ohta87, Kapl87]. In practice, the constants involved are quite large, and a complexity of $O(p \log n)$ has proven a more realistic estimate.

Cleary and Wyvill [Clea88] discuss the optimal number of grid cells for the regular grid. Again given a fairly uniform scene distribution, $\sqrt[3]{n}$ can be shown to be the optimal number of grid cells for a scene with n objects.

Finally, ray tracing is inherently amenable to parallelization. Each individual ray can be calculated on a separate processor, no interaction is needed with other rays. Parker et al. [Park99] present an interactive ray tracing system that achieves up to 20 frames per second on fairly complex scenes, and their system demonstrates almost linear speedup with the number of processors used.

2.6 Mathematical tools

This section is intended to briefly introduce mathematical notation and tools used later in this thesis. An excellent treatment of mathematical tools pertinent to computer graphics can be found in Glassner's book [Glas95]. It deals with the methods discussed in this section, and contains links to important mathematics textbooks for further reading.

2.6.1 Bases and their duals

The reader is assumed to be familiar with the basics of vector spaces. Functions can be added together and multiplied by scalars, thus functions form vector spaces. While the inner product of two vectors in R^3 is

$$\langle v, w \rangle = v_x w_x + v_y w_y + v_z w_z$$

the inner product of two functions is defined as

$$\langle f, g \rangle = \int fg$$

In what follows, if not otherwise noted, functions and vectors can be used interchangeably. So, the norm of a vector (or a function) is defined as

$$\|v\| = sqrt\langle v, v\rangle \tag{2.1}$$

Typical function spaces are made up of all functions over a certain domain for which expression 2.1 makes sense (i.e., is not infinite—such functions are said to have finite energy). Two vectors are said to be orthogonal iff $\langle v, w\rangle = 0$. A basis of a vector space is a set of vectors $B_1, \ldots, B_n$ such that each vector v can be uniquely expressed as a linear combination of the basis vectors. In other words, there exists a unique set of scalars $c_1, \ldots, c_n$, such that:

$$v = \sum_{i=1}^{n} c_i B_i \tag{2.2}$$

Each vector space has a basis, and the number of elements of a basis defines the dimension of the vector space. For function spaces, this number is frequently infinite. A basis is said to be orthogonal if any two elements of the basis are mutually orthogonal. The dual basis is a set of vectors $b_1, \ldots, b_n$ that fulfills

$$\langle b_i B_j\rangle = \delta_{ij}$$

and forms itself a basis [1]. Orthogonal bases are self-dual. Dual bases are important because they can be used to calculate the scalars in equation 2.2. As can easily be shown, the following equation always holds:

$$v = \sum_{i=1}^{n} \langle b_i, v\rangle B_i$$

The duals have another property very important for function spaces. Assume V is a finite dimensional subspace of a (not necessarily finite dimensional) vector space H (for the mathematically versed: H should be a Hilbert space). Then for any vector v in H there is exactly one best approximation v' in V, i.e., there exists exactly one vector $v' \in V$ so that

$$\|v' - v\| = \min_{w \in V} \|w - v\|$$

This element v' can be calculated using the duals exactly as in 1:

[1] Note that δ_{ij} denotes the Kronecker delta, defined 1 if $i = j$, 0 otherwise.

$$v' = \sum_{i=1}^{n} \langle b_i, v \rangle B_i$$

The vector v' is also called the projection of v onto V. The dual basis always exists and can be calculated by a process called Gram-Schmidt orthogonalization.

2.6.2 Monte Carlo integration

Monte Carlo integration is a powerful tool for the integration of arbitrary functions and has its roots in some interesting properties of the statistical expected value. Let X be a random variable and p be a function describing its statistical distribution. For a one-dimensional, real-valued random variable, for example, this means:

$$P(X < x) = \int_{-\infty}^{x} p(u)du$$

(where $P(X < x)$ denotes the probability that X is less than x). The expected value $E_p g(X)$ of a function g on X is then given by

$$E_p g(X) = \int g(u)p(u)du$$

It is interesting that the expected value can be approximated by the mean of random invocations of g. Assume $X_1, X_2, \ldots$ are all distributed according to p, then the mean μ_n is given by

$$\mu_n = \frac{1}{n} \sum_{i=1}^{n} g(X_i)$$

and it always holds that

$$\lim_{n \to \infty} \mu_n = E_p g(X)$$

Arbitrary integrals can be evaluated by generating n uniformly distributed (over the domain of integration) values and taking the mean of the function values. Better convergence and sample utilization can be achieved if the function to be integrated is composed of a general part g and a part p which can be easily integrated: the random values should then be distributed according to p and not uniformly, and subsequently the mean is taken of the values of g only.

The convergence of the integral is usually estimated via the variance s_n:

$$s_n = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \mu_n)^2 = \frac{1}{n-1}\sum_{i=1}^{n} X_i^2 - n\mu^2$$

The variance of the approximation μ_n is then given by $\frac{s_n}{\sqrt{n}}$.

2.6.3 Sampling theory and antialiasing

Since the days of early raster graphics, aliasing has plagued computer graphics professionals throughout. Jaggies, moiré patterns, strobe effects and staircase lines are only some of the names by which aliasing artifacts are generally known. Signal theory and in particular Fourier analysis has helped tremendously to understand the phenomenon and find remedies for it. Only the most important concepts will be mentioned here.

To analyze a signal, it is common to look at its spectrum. Each *periodic* signal can be written as an infinite linear combination of sinusoids, each a multiple of the base (lowest) frequency. *Aperiodic* signals can contain any (spatial) frequency, thus they are written as an integral over all possible sinusoids. The function giving the weight for each sinusoid is calculated by the Fourier transform, and is called the *spectrum* of the function.

Common image operations can be analyzed in terms of the spectrum: uniformly *sampling* a function (i.e., taking the value of the function at regular intervals) causes the spectrum to be replicated at intervals equaling the sampling frequency. A function is called *band limited* if there is a highest frequency in the spectrum. Obviously, if a function is not band limited, sampling it will cause the spectrum replica to overlap. This phenomenon is called aliasing, because high frequencies of the signal will be visible as low frequencies. A band-limited signal has to be sampled at twice its highest frequency to avoid such overlaps. This frequency is called the Nyquist frequency.

Signals that are not band limited or that contain higher frequencies than the Nyquist frequency have to be low-pass filtered by multiplying the spectrum with a box function (which lets only low frequencies pass). A result of Fourier analysis is that the same effect can be achieved by convolving the function with a sinc-function. The *convolution* of two functions is defined as

$$(f \star g)(x) = \int f(t)g(x-t)dt$$

and the sinc as

$$\mathrm{sinc}(x) = \frac{\sin(x)}{x}$$

The sinc function is the ideal low-pass filter, but usually has to be approximated, for example by a Gaussian. Low-pass filtering a function is usually what is meant by antialiasing.

To recover a continuous signal from a sampled one, the replica in the spectrum have to be removed, which can again be achieved by multiplying the spectrum with a box, or convolving the sampled signal with a sinc. This process is called *reconstruction* of the signal, and again, instead of the ideal reconstruction filter—the sinc—some other filter is usually used as an approximation.

Image operations on digital images usually involve reconstructing a continuous image from the sampled one, transforming it, low-pass filtering the result for the new resolution, and sampling it again. If the transformation is just a scale, this is called resampling at a higher (magnification) or lower (minification) resolution. As magnification does not introduce higher frequencies into a signal, low-pass filtering is not necessary here.

For instance, bilinear filtering of texture maps is a reconstruction filter, whereas mip-mapping is an example for a low-pass filter. The last step in a computer graphics system is usually the display on a computer screen. The reconstruction filter implied by the phosphors on a CRT resembles a Gaussian. On the other hand, the first step in acquiring a digital image, for example using a digital camera, involves a low-pass filter, usually also something like a Gaussian.

Finally, an important similarity between linear algebra and signal theory should be pointed out: low-pass filtering a function is similar to using dual basis functions to calculate the scalar coefficients in the representation of the function within a certain basis. The scalar coefficients can be seen as samples of the function, and reconstructing the function is similar to calculating the linear combination of the coefficients with the basis functions. Actually, the sinusoids do form a function basis, and it is an orthogonal basis, therefore self-dual.

2.7 Discussion

This chapter has presented a terse overview of the research pertinent to this thesis. Sections 2.6 and 2.5 constitute preliminary work in the sense that the techniques shown will be directly applied in the methods presented in later chapters. The

mathematical background will be necessary for point-based impostors, while ray casting is used in both methods. Section 2.2 presents important background material because it shows techniques that have to be used concurrently to our methods. Good knowledge of graphics hardware and the standard rendering pipeline is a prerequisite for any real-time rendering system, but is by itself sufficient only for scenes of modest complexity.

Visibility culling will be frequently used to reduce the geometric complexity of large models. However, the resulting potentially visible sets will typically still be too complex to render at monitor refresh rates, however. The situation can be alleviated by using levels of detail for certain well-defined types of objects. Point rendering provides an alternative to levels of detail. The point rendering algorithms presented in this chapter bear resemblances to level-of-detail techniques in that they are intended to efficiently render very complex, but well contained individual objects.

Indeed, the point-based impostors shown in chapter 5 adapt the idea of point rendering (section 2.4) to a more general setting. By refining the geometric sampling and appearance filtering aspects of point rendering, point-based impostors will be able to represent a large number of disconnected objects, typical for distant views.

Both methods presented in this thesis are in their nature image based. However, few of the image-based techniques from section 2.3 have been used to simplify distant geometry while working in conjunction with the standard rendering pipeline. While impostors based on single texture maps are intended for that purpose, they cannot capture parallax effects within objects and provide low overall visual quality. Similarly, visual quality is also a concern with precomputed impostors meshes. Layered depth images do provide correct parallax, but no hardware implementation is available, preventing fast rendering and integration with the standard rendering pipeline. We will use LDIs, however, to obtain geometric samples for point-based impostors. Light fields, on the other hand, require geometric information for scenes of large depth variations, and prohibitive amounts of images would be needed to prevent depth-of-field effects. The point-based impostor method can also be seen as an efficient way to add geometric information to a light field.

Chapter 3

Motivation—General Terms

This chapter is intended to lay the ground for the following chapters. It introduces some terms that will be recurrent throughout the rest of the thesis, such as the near field and the far field. There will be some clarifications on mathematical notations for later chapters. Furthermore, the terms discussed here will motivate in the most general fashion the methods presented later on.

3.1 Near field and far field

Virtually all real-time rendering applications provide the user with a view on the scene that tries to closely mimic the characteristics of human eyesight. The most distinguishing feature of visual perception is its ability to recognize depth, particularly through perspective. Unlike in orthogonal projections, the projected size of objects varies with their distance to the viewer in perspective projections, so that objects that are far away do not contribute as much to the image as nearby objects. Furthermore, the user's attention is usually more focused on nearby objects, with which he will typically interact, and not on far away objects.

In contrast to that, traditional rendering pipelines usually spend a large amount of time on rendering objects that are far away, because the rendering time is proportional to their world-space complexity, not to their screen-space area. This warrants a distinction of nearby and distant objects.

For the remainder of this thesis, we will use the following terms (this terminology was also used by Chamberlain et al. [Cham96], who described an algorithm to render the far field with a different rendering method):

- near field: designates all objects close to the viewer

- far field: designates all objects farther away from the viewer (usually all objects not in the near field)

Obviously, a more precise definition of near field and far field is needed in a particular application.

One choice is to simply draw the border between near field and far field at a predefined distance d to the viewer. This distance could be the Euclidean distance from the viewpoint to the center of the bounding box of the object. The shape of the near field would then be a sphere with radius d, intersected with the viewing frustum. Another possibility is to choose the distance along the viewplane normal. In this case, the near field would be a standard pyramidal viewing frustum with the far plane placed at d. The far field would also be a standard viewing frustum, but with the near plane at d and the far plane at infinity.

Near and far field need not be defined through a fixed borderline distance. It could be defined on a frame-by-frame basis through the characteristics of the scene and the rendering algorithms used. For instance, objects could be sorted according to their distance, or alternatively to their projected screen area. The near field could then be defined to contain objects up to a certain polygon budget. This is especially interesting for frame-rate control: if the rendering speed of objects is mainly determined by their polygon count, a maximum frame time could be guaranteed by drawing only the near field.

When a distinction of a scene into near field and far field has been made, the question remains how to make use of this distinction. A common approach used in games is to fade out the near field with a fog function, and not to draw the far field at all. Some games use environment maps to represent very distant objects such as mountain ranges or sky, but in general, this approach is limited in the quality that can be achieved. Important information can reside in the far field, and the resulting popping artifacts from geometry crossing the line between far field and near field disturb the user. Other alternatives to draw the far field are impostor meshes, ray casting as proposed in the next chapter, or point-based impostors.

3.2 Online and offline calculation

Accelerating real-time rendering applications requires some a priori knowledge about the scene. This knowledge can be generated automatically in an offline phase, or it is already available, in which case it usually needs to be converted into a form usable by the algorithm. Furthermore, some algorithms execute the bulk of their computation online, during the walkthrough, while other algorithms

precompute most of the needed information beforehand, and only require a table lookup in the online phase.

Both variations have advantages and disadvantages:

- An online algorithm can be applied immediately to any dataset. Many datasets are volatile and change frequently, so any lengthy precomputations as required in offline algorithms would hinder their usability. This may not always be a concern: fixed datasets in commercial software are a good candidate for precomputation, because the result of the precomputation will be used for a long time.
- Online algorithms can be applied to datasets that change during runtime. In many cases, this makes for a more flexible environment, and is a prerequisite for modeling applications where the goal is to change the environment. Offline algorithms usually don't allow the environment to change. Again, this may not always be a concern.
- However, online time is a scarce resource. In order to maintain a frame rate of 60 Hz, for example, only 16 ms are available in total per frame. An online algorithm that takes 20 ms to execute might achieve large speedups in rendering, but it will never allow the application to run at the desired frame rate. Even if some concurrency with the graphics accelerator is possible, the bulk of the available frame time should be dedicated to rendering geometry, not on deciding what geometry to render. This does not preclude online algorithms—it only means that they should be fast enough. Offline algorithms have the advantage that they can spend large amounts of time on optimization. Often, this results in better speedups for the final application. Yet even for offline algorithms, computation time is a concern: several minutes of precomputation might be inconvenient, but acceptable, whereas several hours usually are a nuisance. Still, in many cases the result is worth the wait.

The final decision on online or offline algorithms depends on the application requirements, if there is a choice at all. This thesis presents examples of both variants. Ray casting is a typical example of an online algorithm, while point-based impostors are calculated offline.

3.3 Notation of vectors in this thesis

To simplify notation, vector parameters will be frequently used for functions throughout this thesis. For example, the plenoptic function shown earlier will

be written as $P(s, \varphi)$, where s is a three-dimensional position vector and φ a three-dimensional orientation consisting of azimuth and elevation.

Chapter 4

Ray Casting with Image Caching and Horizon Tracing

4.1 Introduction

An important application of real-time rendering is the so-called “walkthrough-scenario”. The user has control over camera position and camera orientation, and navigates an interactively rendered virtual environment.

Of the methods discussed in chapter 2, many are applicable to the walkthrough scenario, depending on the type of virtual environment: Indoor scenes can be efficiently handled using portal rendering. For sparsely populated outdoor scenes, the level-of-detail approach is viable, providing a number of representations for the same object with different rendering costs. Image-based rendering has been proposed for very general, complex scenes. More recently, occlusion culling methods have been investigated to handle densely occluded, yet unrestricted scenes, for example urban environments.

In this chapter, we present a new approach to the problem of interactively rendering large virtual environments. We begin by noting several basic observations, valid especially in the case of densely occluded outdoor environments, such as urban environments:

- A large part of the screen is covered by a small set of polygons that are very near to the observer (in the near field).
- Another large part of the screen is covered by sky.
- Pixels that do not fall into one of these two categories are usually covered by very small polygons, or even by more than one polygon.

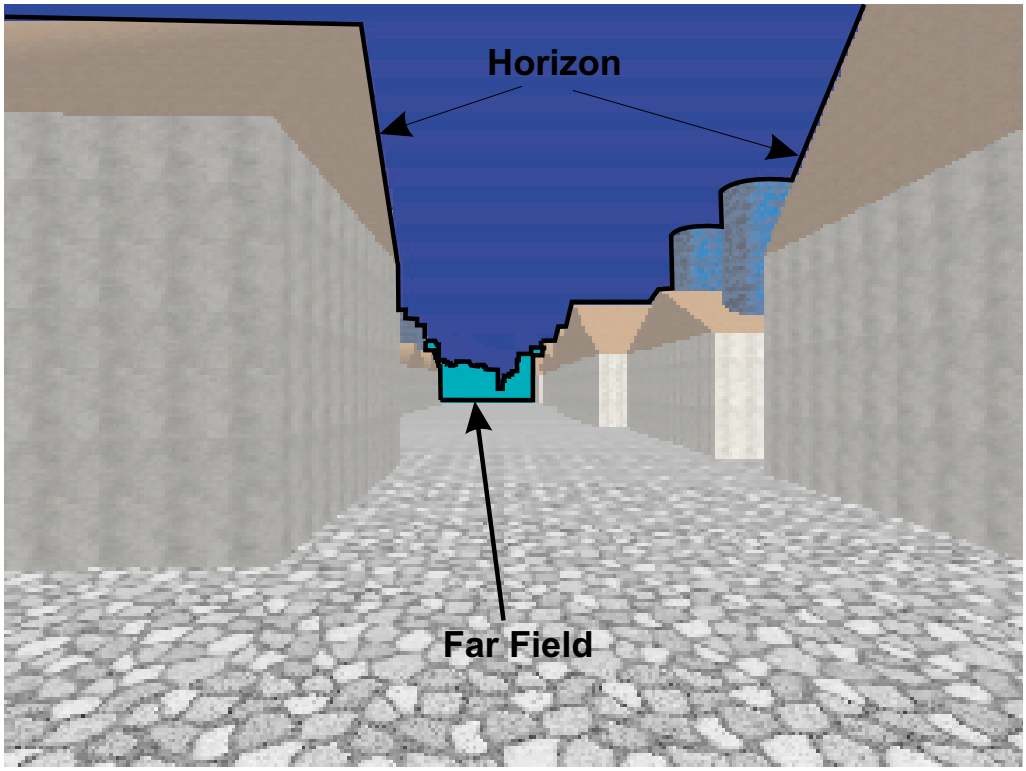


Figure 4.1: Ray casting is used to cover the pixels of the far field (beyond 100 m). Pixels above the horizon are culled early.

- The number of polygons that fall outside a certain “area of interest” is usually much larger than what a polygonal renderer can handle—but they still contribute to the final image.

The main contribution of this chapter is a new algorithm for accelerated rendering of such environments. It exploits the observations listed above: the scene is partitioned into near field and far field. The near field is rendered using traditional graphics hardware, covering many pixels with polygons, whereas the far field is rendered using an alternative method. Every pixel not covered by a near-field polygon undergoes a multi-stage image-based rendering pipeline in which it is either culled early or sent to the last stage, a ray casting algorithm (see figure 4.1).

The method can be seen as a hybrid hardware / image-based rendering algorithm that uses a new way to obtain images on the fly with very low memory overhead. The algorithm is in its nature output sensitive [Suda96]: by restricting hardware rendering to the near field, approximatively constant load of the hardware graphics pipeline can be achieved. The remaining pixels are also obtained in

Figure 4.2: The traditional rendering pipeline consists of three steps.

an output-sensitive manner: both the culling stage and the ray casting stage can be shown to have linear time complexity in the number of pixels only. Ray casting, if combined with an acceleration structure, is less than linear in the number of objects.

4.2 System overview

The traditional polygonal rendering pipeline consists of three basic steps (figure 4.2). Depending on the architecture, each of them may or may not be accelerated in hardware. What is obvious, though, is that the time complexity of rendering the scene is always linear in the number of primitives, because arbitrarily many objects may be visible at any one time.

We introduce a second stage to the pipeline, which is not primitive-based, but purely image-based (figure 4.3). Each of the two stages is able to render the whole scene alone, but both would be equally overloaded by the whole database. Thus, rendering of the scene is distributed to the two stages by partitioning the scene into near and far field. All primitives within a distance less than a certain threshold are sent to the polygonal renderer. The second stage passes over all the pixels in the image and fills those that have not yet been covered by near-field polygons with an appropriate color.

In the pixel stage, various mechanism exist to allow early exits before a ray is cast to obtain the color information (figure 4.4):

- Pixels already covered by polygons are recognized by an opacity buffer that is created during the first stage.
- Pixels which fall into an area covered by sky are recognized by a horizon map created before the second stage.
- If a pixel fails those two tests, but its validity is still deemed to be within a valid error range, the pixel color is looked up in an image cache.

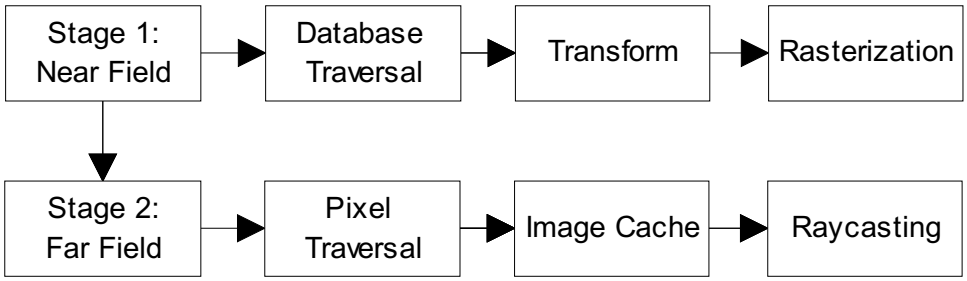


Figure 4.3: The extended rendering pipeline uses a polygonal pipeline for the near field and an image-based pipeline for the far field.

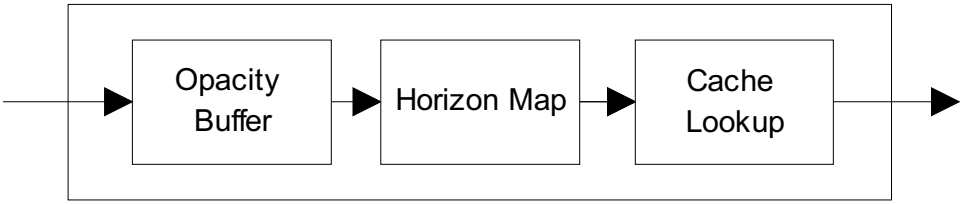


Figure 4.4: The image cache is composed of three stages.

Only if all these three tests fail, a pixel is sent to the final step in the pipeline, the ray caster. It is important to note that, while ray casting is a very costly technique in itself because it has to be done purely in software, the time complexity of the ray caster is still less than linear in the number of objects. Also, by restricting the number of polygons sent to the graphics hardware, the time spent in the polygonal graphics pipeline is bounded as well (given a reasonably uniform distribution of primitives in the scene). Thus, the overall algorithm can be said to be output-sensitive, i.e., its time complexity is linear in the number of visible primitives only, but less than linear in the total number of primitives.

4.3 Ray casting

4.3.1 Near field / far field

As already noted in section 3.1, when rendering a large scene, a huge number of objects can reside in the area defined by the viewing frustum, but only a small

amount of those objects actually contribute to the appearance of the image. Large, near objects usually have much more impact on appearance than small and far away objects.

It should therefore be possible to determine a set of objects that have the most contribution to the image, and only render this set of objects. The simplest way to achieve this is to select all objects within a maximum distance of the viewer. We define the near field to be the space where these particular objects reside. This approach is very popular, especially in computer games, and it is often combined with fogging, so that the transition between the near field and the space where objects are simply not rendered is not so sudden.

Obviously, culling away all objects that do not belong to the near field introduces severe visual artifacts. The far field contains objects beyond the near field, but not so far away as to be totally indiscernible. An important property of the far field is that it usually

- consists of much more polygons than the graphics hardware can render, but
- contributes to only very few pixels on the screen, because most of the pixels have already been covered by near field polygons.

To take advantage of this fact, a separate memory buffer, the *opacity buffer*, is used. For every frame, it records which pixels have already been covered by the near field.

The basic algorithm for our image-based rendering technique using ray casting is as follows:

1. Find objects in the near field using a regular grid.
2. Render those objects with graphics hardware.
3. Rasterize them into the opacity buffer.
4. Go through the opacity buffer and cast a ray for each uncovered pixel (enter the resulting color in a separate buffer).
5. Copy the pixels gained by ray casting to the frame buffer.

4.3.2 Ray casting

We claim that ray casting is an appropriate technique for acquiring images for image-based rendering on the fly. This might seem strange at first glance, because ray casting (ray tracing) is known to be a notoriously slow technique. The reason for that is its high complexity: in a naïve approach, every object has to be intersected with a ray for every pixel, so the complexity is $O(pn)$ as discussed in section 2.5.2. In our approach, we cast only primary rays into the scene through individual pixels and find the first hit, i.e., the first intersection with an object in the far field. No secondary rays have to be cast, and we are interested in so-called first-hit acceleration techniques.

From the first days of ray tracing, acceleration structures have been used to reduce the number of ray-object intersection tests for individual rays. As discussed in section 2.5.1, the two most popular are bounding volume hierarchies and hierarchical space subdivision. Of all the methods proposed, the regular grid approach is the most interesting for our purpose: space is partitioned uniformly into a grid structure, and all objects are entered into the grid cells with which they intersect.

It has been shown (see section 2.5.2) that theoretically, using an appropriate acceleration structure, the time complexity of ray tracing can be reduced to $O(1)$, i.e., constant, in the number of objects (although this constant may be very large). In our experiments we have observed a sublinear rise in the time to cast rays into a very large scene.

The advantage of the regular grids is their speed. Also, given a more or less uniform distribution of objects—which we can safely assume for many types of virtual environments—the memory overhead is very low. Tracing through a grid is fast, using for example Woo's incremental algorithm [Aman87] which only requires few floating point operations per grid cell. If more objects are added, runtime behavior can even improve because rays will collide earlier with objects than if there were huge empty spaces in the grid.

The regular grid also provides a simple solution to view-frustum culling, which is necessary to quickly find the objects that have to be rendered in the near field.

Note that since ray casting and hardware rendering provide color values for the same image, care has to be taken to implement the same lighting and shading model for both. The color values produced in the frame buffer by an OpenGL system, for example, are well specified and can easily be simulated in the shading module of the ray caster, so that the distinction between near and far field is not visible on screen.

For certain scenes, ray casting alone might already be sufficient and moderate gains can be observed. But generally, this still leaves too many pixels for

which rays have to be cast, and while the time required for ray casting is relatively independent of scene complexity, casting a single ray is expensive compared to polygonal rendering and thus only tractable for a moderate number of pixels. The following sections explain how image caching and horizon tracing can be used to drastically reduce the number of rays that have to be cast.

4.4 Image caching

4.4.1 Panoramic image cache

Usually, walkthrough sequences exhibit a considerable amount of temporal coherence: the viewpoint changes only by a small amount between successive frames. We exploit this coherence in our system: instead of tracing every far-field pixel every frame, we retain all the color values of the previous frame and try to retrace only those pixels that are outdated according to some error metrics.

The validity of pixels depends strongly on the type of viewpoint motion:

Forward/Backward motion: This makes up for a very large amount of motions in a walkthrough sequence. The farther away an object is, the smaller is the amount of pixels it moves on the screen due to forward/backward motion. Many pixels will even remain at the same location, so just reusing the pixels from the previous frame is already a good first approximation.

Head Rotation: Rotation is quite different from forward/backward motion: reusing the contents of the frame buffer would indeed be a very bad solution, because all pixels would be wrong. But actually, many pixels are still valid, they have just moved to a different position. So what is needed is a method to store traced pixels that does not depend on the orientation of the viewer.

Panning (left/right, up/down): This type of movement is similar to rotation in that most pixels move to a different place.

Our assumption is that forward/backward motion and rotation will be the major types of motion in a walkthrough sequence. We therefore choose a representation which is independent of viewpoint rotation: a panoramic image cache.

Panoramic images have been demonstrated to be a very efficient tool to store full views of a scene where rotation is allowed. We use the panoramic image cache not for presenting a precomputed panorama as for example in the Quicktime VR system [Chen95], but we use it as a rotation-independent image cache.

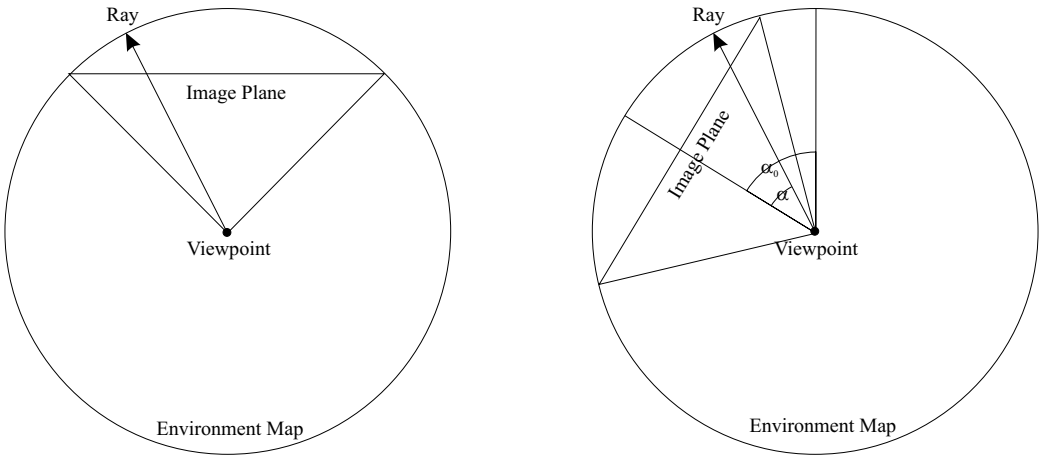


Figure 4.5: Indexing into the panoramic image cache: Given a pixel on the image plane, an angle α can be calculated with respect to the image plane center. This angle does not depend on the initial viewer orientation α_0, therefore it can be precomputed and stored in a lookup table. So, indexing into the image cache consists of looking up α in a table, adding the viewer orientation α_0 and rescaling this value to fit the resolution of the image cache.

When a ray is cast through a pixel on the (flat) screen, its position on the (curved) map is calculated and the color value obtained by the ray is entered in this position (figure 4.5). If, at a later time, another screen pixel projects to the same position in the curved map, its value can be reused if it is still good enough.

The major advantage of using a panoramic image as an image cache is that the validity of pixels stored in the map is invariant under rotation. This means that, as long as the viewpoint does not change, all image elements already calculated and stored in the map can be reused in the new image, provided they still fall into the viewing frustum after rotation. This speeds up rotation considerably: only the very small amount of pixels that appears newly at the border towards which the viewer is rotating has to be retraced. All other pixels can be reused, and their values will be correct.

In the case of forward/backward movement, the behavior of the map resembles that of a normal, flat image map: reusing the previous panoramic map will be a good approximation to the image and many pixels will be in the correct location. Panning causes more pixels to be invalidated if no costly reprojection is used.

4.4.2 Cache update strategy

Assuming that pixels which have been traced in a previous frame are retained in an image cache, the algorithm has to decide which pixels are considered good enough according to a certain error metric, and which pixels have to be retraced. In an interactive walkthrough system, the decision can also be based on a given pixel-budget instead of an error metric: which pixels are the most important ones to retrace, given a maximum amount of pixels available per frame.

As with any speedup-algorithm, worst-case scenarios can be constructed that do not benefit from the algorithm. In such a case, our approach allows progressive refinement by iteratively retracing all necessary pixels every frame. As soon as the scenery gets more suited to the algorithm or the observer does not move for a short moment, the system is able to catch up again.

To select an appropriate set of pixels to retrace, we assign a *confidence value* to each pixel in the map. The pixels are then ordered according to their confidence values and tracing starts with the pixels that have the lowest confidence, proceeding to better ones until the pixel budget is exhausted. Finding a good heuristic for the confidence value of a pixel is not trivial. We have chosen the following approach:

Every frame in which the observer moves more than a certain distance (rotation is not taken into account, as the image cache is rotation-independent), a new confidence record is created, which contains the current observer position (figure 4.6). All pixels which are traced during this particular frame are assigned a pointer to the current confidence record (pixels which have not been traced at all point to a “lowest-confidence” record).

After a few frames, there will be a certain amount of confidence records (as many as there were distinct observer positions), and each pixel in the image cache will reference exactly one of those records. This information is used as follows:

During the polygonal rendering stage, a new opacity buffer is created. All pixels of this buffer are visited sequentially. If a pixel is covered by a polygon, it is ignored. If not, a lookup is done into the image cache to find out the confidence record associated with this pixel. The confidence record also contains a pixel counter which is then incremented.

After all pixels have been visited, each confidence record contains the number of pixels that refer to it in its internal counter. Now, the observer position stored in each confidence record can be compared to the current observer position and the distance between the two is remembered as the current confidence value of this confidence record. All confidence records are sorted according to this confidence value.

```
CacheElement {
    Color;
    Pointer to ConfidenceRecord;
};
ConfidenceRecord {
    ObserverPosition;
    PixelCounter;
    CurrentConfidence;
};
```

Figure 4.6: The basic data-structure used in the cache and for keeping track of confidence values.

Scanning through the confidence records from worst (farthest away) to best (nearest), we add up the counted pixels until the pixel budget is met. This gives a threshold distance: all pixels farther away than this threshold distance will be retraced. Nearer pixels will be reused from the image cache.

This is accomplished by going again through the opacity buffer, indexing into the image cache for every unoccluded pixel and casting a ray for the pixel if its distance (which can be found out by following its pointer to the associated confidence record) is greater than the threshold distance.

One problem with this approach is that it occurs quite often that all pixels share the same confidence value: if the observer stands still for a while and then suddenly moves, all pixels will be assigned the same new confidence value. In this case, the confidence values are not a good indication of where ray casting effort should be spent. We therefore only trace every n-th pixel that has the same distance, such that the pixel budget is met. In the subsequent frame, the remaining pixels will then be selected automatically for retracing.

The distance between the current and previous observer position is used as an error estimation because the panoramic image map is rotation-independent, hence the only value that changes between frames with respect to the map is the observer position. A more elaborate scheme could also store orientations with each confidence record and compare this to the current orientation. For reasons of efficiency, we have chosen the more simple approach of keeping the error estimation independent of rotation. It would be interesting to investigate whether performance improves if one takes additional information about the hit object or the distance to the intersection point into account.

To sum up, our update strategy makes sure that pixels are retraced in the order

of their distances to the current observer position, taking into account a pixel budget that allows for graceful degradation if the demand for pixels to be retraced is too high in a particular frame. Note that on average, the area left for ray casting only covers a small portion of the screen.

4.5 Horizon tracing

Let us reiterate some typical properties of virtual environments: they have

- a polygonal floor, and
- either a polygonal ceiling or
- empty sky.

For indoor scenarios with a polygonal ceiling, the system as presented so far would already be sufficient. Problems arise, however, for outdoor scenarios with large areas of empty sky. Theoretically, the ray tracing acceleration structure should take care of rays that do not hit any object in the scene. But in fact, even the overhead of just setting up a unique ray for every background pixel is much too large as to be acceptable. The usual case in outdoor scenes is that between one third and one half of the pixels are covered by polygons. A very small part is covered by far-field pixels that do hit objects, but the rest of the screen is covered by sky.

If it were possible to find out where the sky actually starts, most of the sky pixels could be safely ignored and set to a background color or filled with the contents of a static environment map.

We assume that the viewer only takes upright positions, i.e., there is no head tilt involved. This is a reasonable restriction in a walkthrough situation. Then, we observe that the screen position where the sky starts only depends on the x-coordinate in screen space, i.e., on the pixel column. So, for every pixel column we have to find out the y-coordinate of the horizon.

This, again, is a problem that can be solved by ray casting, but in two-dimensional space. In addition to the 3D regular grid that is used for tracing pixels, a 2D regular grid is created that contains the height value of the highest point in each grid node—a two-dimensional height field.

For every frame, a two-dimensional ray is traced through this height field (figure 4.7) to find the grid node that projects to the highest value in screen space

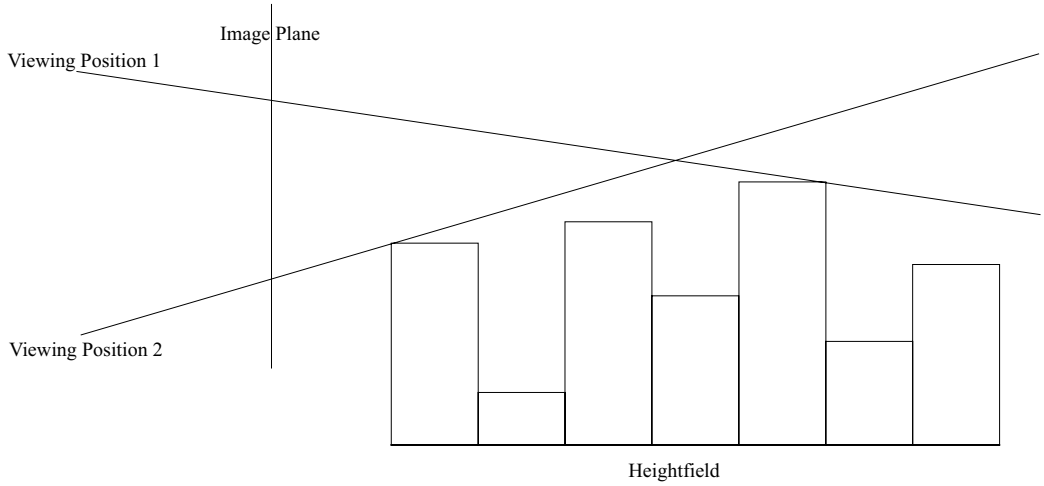


Figure 4.7: The image shows a cut through the height field along the path of one particular horizon ray cast from two viewing positions with different heights. Note that it is not always the highest point in the heightfield that determines the height of the horizon on the screen.

(note: this need not be the highest point in absolute coordinates!). All pixels with a height above this value can be ignored and set to the background color.

Our results indicate that the reduction in the number of pixels to trace was so substantial that the total time spent ray casting and the time spent horizon tracing were comparable. This makes horizon tracing itself a further candidate for acceleration.

One way to speed up horizon tracing is to carefully adjust the resolution of the height field. As opposed to pixel ray casting, rays cast through the height field have to travel through the whole 2D grid so as to find the point whose projection has the highest y-value on the screen. Whereas the 3D grid profits from higher resolution because of improved intersection culling, it is detrimental for horizon tracing because of the large number of grid cells that have to be visited. Even though a coarser grid tends to overestimate the horizon height, the speedup gained through faster horizon tracing makes up for this.

Another way to speed up horizon tracing is to apply the principle of graceful degradation to the horizon map in the same manner as to the image cache: as long as the viewer is moving, the horizon is subsampled and the locations between samples are filled with the maximum of the adjacent samples.

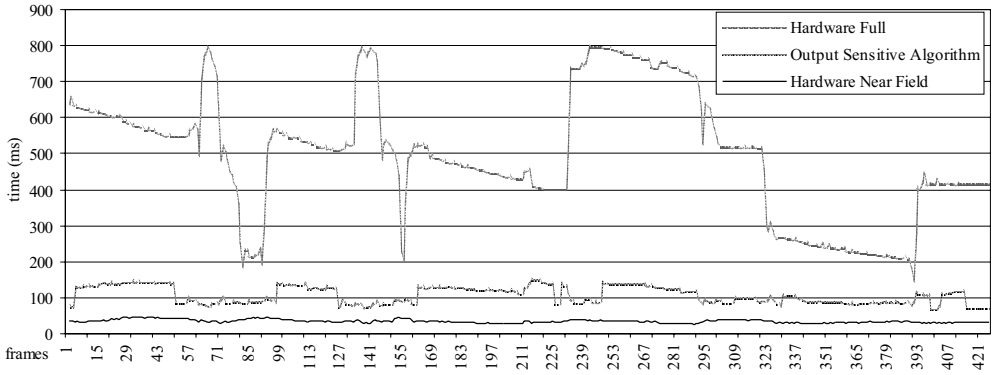


Figure 4.8: The chart compares full hardware rendering (far plane set to infinity), our new output sensitive algorithm, and hardware rendering (far field not rendered) with the far plane at 100 m. The image resolution was 640x480 pixels for all tests. The average frame rates were 2.0 fps for full hardware rendering and 9.25 fps for the new algorithm, so the speedup is about 4.6.

4.6 Results

The algorithms described in this chapter have been implemented and tested in an application environment for creating professional computer games. The system was tested with a Pentium 233MMX processor, which was moderately fast for a consumer PC at the time when the tests were conducted. The 3D board used was a 3DFX Voodoo Graphics, one of the first PC graphics boards providing significant pixel fill rate. Both CPU and graphics power have increased about tenfold since the implementation of these tests, so the results shown here are still a good estimate of the speedup that is possible on current systems. The implementation is quite crude, and performance gains are likely to be achieved by careful optimization of critical per-pixel operations. New SIMD instructions available in current CPUs give further room for optimization.

One problem to be solved is how to create an occlusion map, and how to reuse it for rendering. Surprisingly, graphics hardware is not of much help in this case: transfers from frame-buffer memory to main memory are usually very slow, except theoretically in some specialized architectures which incorporate a unified memory concept (e.g., the Silicon Graphics O2 [Kilg97]).

Therefore, while the near field is rendered in the frame buffer using graphics hardware, we create a 1-bit opacity buffer with a very fast software renderer, taking advantage of the fact that neither shading nor depth information is required for the opacity buffer. Our current implementation is limited to upright view-

ing positions only. This restriction is inherent to the horizon tracing acceleration, and we believe that it does not severely infringe on the freedom of movement in a walkthrough environment. With respect to the image cache, a spherical map could easily be used instead of the cylindrical map that we chose to implement, allowing the viewer to also look up and down.

Although our ray caster can simulate the color values produced by hardware rendering exactly, we observed that typical far-field triangles do not cover more than one or two pixels. As we are also using a prelit model, we decided to store for each triangle a precalculated color value with lighting and texture (on the lowest mip-map level) already applied. This avoids having to find the exact intersection point and evaluating the lighting model for each triangle intersection. We did not notice a significant degradation in image quality due to this optimization.

The first graph (figure 4.8) shows the time taken to render each frame of a recorded walkthrough sequence (about 400 frames) through a moderately large environment, a city containing approximately 150,000 triangles. Two of the series are for pure hardware rendering only, with the far plane set to infinity in one case and 100 meters in the other case. The far field is not rendered at all, and our algorithm disabled completely (so there is no overhead for tracing horizon pixels, creating or going through the opacity buffer etc.). It shows that up to a certain distance, graphics hardware can render the scene very quickly, but of course misses out on a considerable amount of the background. But if the whole scene is rendered indiscriminately, the hardware simply cannot cope with the amount of triangles, and the frame rate drops to an unacceptably low value.

Obviously, these are the two extremes between which our algorithm can or should operate. It will certainly not get faster than just rendering the near field, but it should be considerably faster than rendering the whole scene with triangles only. The third series shows how the algorithm performs for the same walkthrough sequence with the far plane set at 100 meters—ray casting, image caching and horizon tracing enabled. A speedup of up to one order of magnitude over rendering the full scene in hardware can be observed.

To give a feeling for the operating behavior of the algorithm, the second graph (figure 4.9) shows frame times for our algorithm with different near-field sizes (i.e., the far plane set to different values). Increasing the far-plane distance beyond a certain limit reduces performance, because more triangles have to be rendered, but they do not further reduce the number of pixels that have to be traced. Setting the far plane too near gives a very non-uniform frame rate.

The third graph (figure 4.10) gives an impression of what the algorithm is capable of if the screen resolution is reduced and the scenery is more complex (in this walkthrough sequence, almost all of the polygons were in the viewing

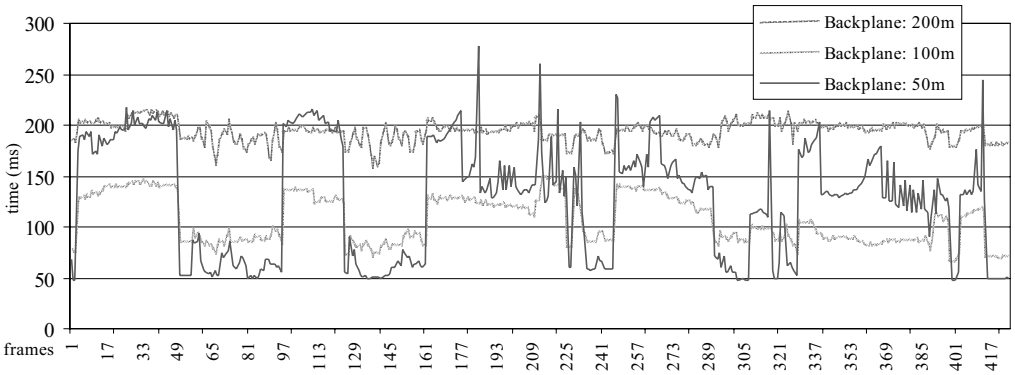


Figure 4.9: The chart compares the behavior of the algorithm with respect to the size of the near field at a resolution of 640x480 pixels.

Of 307,200 pixels ...	Min [%]	Avg. [%]	Max [%]	Avg. [#]
ray cast and hit an object	0.00%	0.13%	0.49%	400
ray cast and missed	0.00%	0.10%	0.38%	307
taken from the image cache	0.00%	0.13%	0.81%	399
culled by horizon tracing	7.38%	23.76%	39.21%	72,990
covered by polygons	60.37%	75.98%	92.62%	233,410

Table 4.1: Illustrates that very few pixels have to be calculated using ray casting in a densely occluded environment (measured at a resolution of 640x480 pixels)

frustum most of the time, so view-frustum culling is not able to cull geometry). Table 4.1 shows that the performance of the output sensitive algorithm is due to heavy occlusion in the walkthrough sequence.

The images at the end of this chapter (figures 4.11 and 4.12) show two views of the virtual city the walkthroughs were recorded in. The border to the far field is indicated by a line. In figure 4.11 there is a view from an elevated position, which does not satisfy the assumptions because there is no significant occlusion. The expected frame rate for such a view is about 4 frames per second for a resolution of 640x480 pixels (hardware rendering alone would be about 1 frame per second). Figure 4.12 represents a typical shot from a walkthrough sequence that does fulfill our assumption of dense occlusion.

Note that for the walkthroughs shown and with the parameters chosen, the algorithm did not need to resort to graceful degradation (i.e., not tracing all rays every frame). This did occur, however, in some parts of the model with wide open spaces, and when the near field was too small.

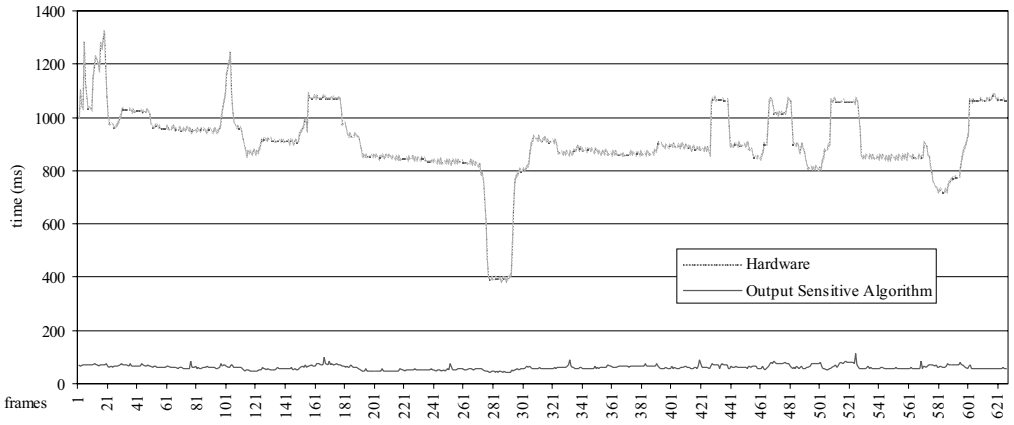


Figure 4.10: Frame times for a different walkthrough sequence at a resolution of 320x240 pixels. The average number of polygons in the viewing frustum was higher than in the first sequence, making hardware rendering even slower. The average frame time for hardware rendering was 1.1 fps, for the new algorithm 16.3 fps, so the speedup is 14.8.

4.7 Discussion—advantages and shortcomings

4.7.1 Scalability

The rendering algorithm described in this chapter is applicable to a wide range of environments (see applications). The same is true for the type of platforms it can be used on. Originally, it has been designed with the consumer PC in mind, where almost every new computer is equipped with a 3D accelerator. These accelerators share a common property: they are very good at triangle rasterization, but the transformation step has to be done by the main CPU. Rendering scenes that contain a lot of primitives easily overloads the transformation capabilities of the CPU, and the 3D card is idle. Instead of transforming all primitives with the CPU, the algorithm can put this processing power to better use: by using the methods described, and some a priori knowledge about the scene, the 3D accelerator is used to quickly cover the near field with polygons, and the remaining CPU time is used for the pixel-based operations.

The algorithm is not restricted to such a platform, though. As the power of the 3D pipeline increases, the size of the near field can be increased as well, thus leveraging the additional triangle processing power of the pipeline. More pixels will be covered by polygons, and even fewer pixels will be left to send to the ray casting step. This is especially true if the geometry transformation stage is implemented in hardware, as is more and more the case even for consumer PCs

and 3D workstations.

But even if a high-end graphics pipeline exists, the ideas of this paper are valid: there will always be scenes too large and too complex to handle even with the best graphics hardware. Adjusting the size of the near field to the speed of the polygon pipeline provides a good parameter for tuning an application for speed on a specific platform.

This means that the approach scales very well with CPU processing power as well as with graphics pipeline speed, and the result is an output-sensitive algorithm that can be used in many different setups.

4.7.2 Aliasing

No speedup comes without a cost. There are two reasons why aliasing occurs in the algorithm: first, ray casting itself is a source of aliasing because the scene is point sampled with rays (the same is true for hardware geometry rendering, by the way). The other reason is aliasing due to the projection of the flat screen into a curved image map and back.

In both cases, antialiasing would theoretically be possible, but it would have a heavy impact on the performance of the algorithm, thus defying its purpose, i.e. to accelerate interactive walkthroughs. In chapter 5, we will present a method to remedy aliasing effects. In contrast to the online method presented here, however, this method will require a significant amount of preprocessing.

4.8 Applications

There is a variety of applications where the algorithms presented in this chapter could be applied. Foremost, there is:

4.8.1 Walkthroughs

Many types of virtual environment walkthroughs fulfill the basic preconditions the algorithm requires. First and foremost, urban environments are ideal to showcase the points of this chapter. Especially in a city, most of the screen is covered by the buildings that are near to the viewer. But there are also several viewpoints where objects are visible that are still very far away—imagine looking down a very long street. Polygons cover the right, left and lower part of the image, a good part of the sky is caught by horizon tracing, and the remaining part can be efficiently

found by ray casting. Note that under normal circumstances, such scenes are either excruciatingly slow to render, or the far-plane distance is simply set so near that the result does not look very convincing.

Any other scenery that is densely occluded is also suitable. For example, walking through virtual woods is very difficult to do with graphics hardware alone—but with our algorithm, a good number of trees could be rendered in the near field, and the remaining pixels traced.

4.8.2 Computer Games

In recent times, first person 3D computer games have gained immense popularity. Many of them are restricted to indoor environments, because portal rendering provides a very good solution for the complexity problem in this case. But few have ventured to outdoor scenarios, and most of those who have rely on heavy fogging to reduce the amount of polygons to render. Sometimes the far plane is not set much farther than 10–20 meters, which does not provide for a very realistic feeling. Using the described algorithm, the perceived far plane can be pushed back to the horizon, or at least a considerable distance further away, as the space between the previous far plane and the horizon can be covered by far-field rendering.

Neither graphics hardware nor processing power will be lacking for computer games in the near future, as both are rapidly catching up with workstation standards. The benchmarks were done on a system whose performance is by no means state of the art even for a PC environment (see section 4.6), but they show that good results can be achieved nevertheless.

4.8.3 Portal Tracing

Previous work [Alia97] has suggested the use of textures as a cache for rendering portals in indoor environments. Those textures are calculated by using graphics hardware. We believe that it might be advantageous to use ray casting to trace through the portals: far away portals cover only a small amount of space on the screen, so there are very few pixels to trace—but the amount of geometry behind a portal can still be quite large, especially if portals have to be traced recursively. Of course in this case, the horizon tracing stage can be omitted.

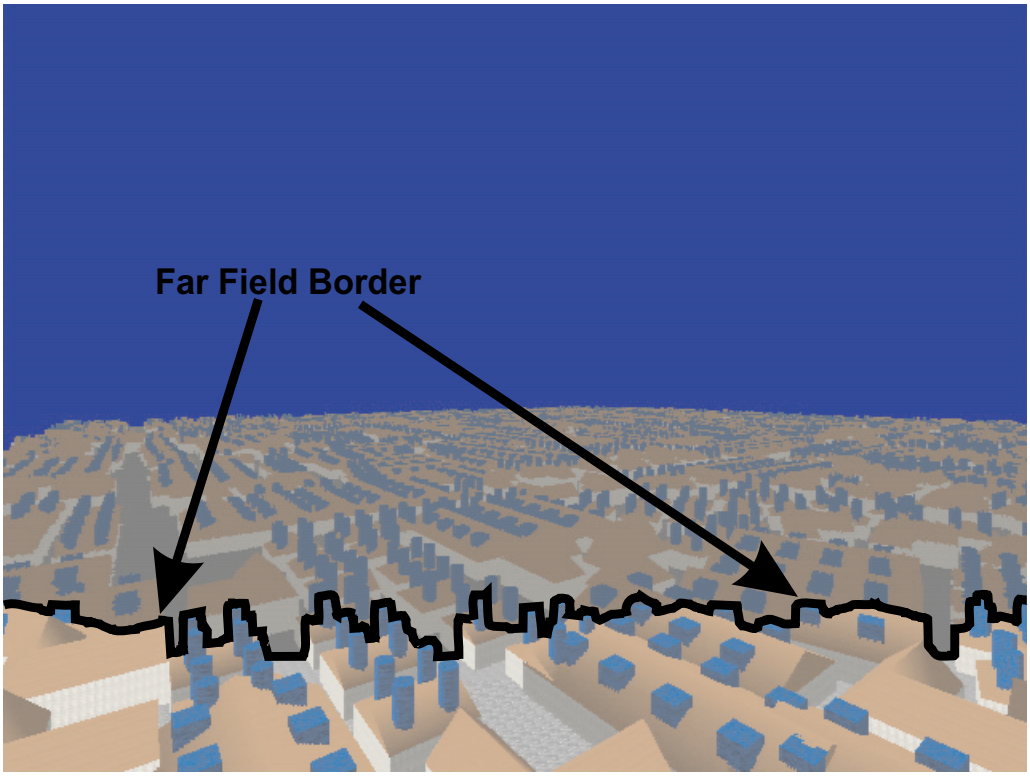


Figure 4.11: A view over much of the virtual city that was used for the walkthroughs.

4.8.4 Effects

A potential application for some of the ideas of this chapter is to render certain special effects that do not require high accuracy: reflections on partly reflective surfaces can be adaptively ray traced using only a small number of rays—the effect would be visible, but one can avoid having to rerender the whole scene multiple times as is usually necessary for such effects.

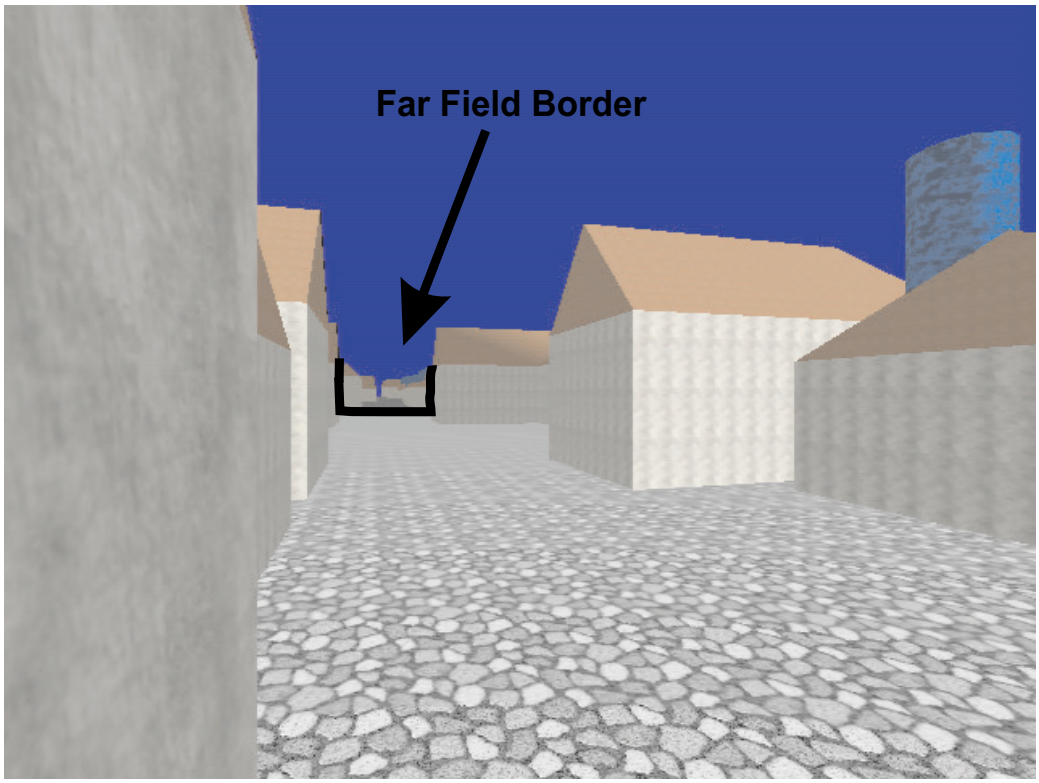


Figure 4.12: A typical view from a walkthrough sequence in the city.

Chapter 5

Point-Based Impostors

5.1 Introduction

Figure 5.1: A scene from an urban walkthrough. Geometry in the red region has been replaced by a point-based impostor, providing for fast, antialiased rendering of the far field.

A general and recently quite popular approach to handle the interactive display of complex scenes is to break down the view space into cells (*view cells*) and compute optimizations separately for each view cell. A typical example is region visibility, which calculates the potentially visible set (PVS) of objects from a specific view cell.

In this chapter we address the problem of simplifying distant geometry (the far field) for view cells (see figure 5.1). This is an important problem because rendering distant geometry often leads to aliasing artifacts (see section 3.1) and, even after visibility calculations, the amount of geometry remaining in the PVS is often overwhelming for current graphics accelerators.

We begin by defining our goals in this process. First, we set out to build a representation that provides a high-quality rendered image for all views from a specific view cell. This means that we try to avoid artifacts such as holes, aliasing or missing data, and that we require high fidelity in the rendering of view-dependent appearance changes. Second, we insist that our representation be as compact as possible, while being amenable to hardware acceleration on consumer-level graphics hardware.

Distant geometry has some peculiarities resulting from its complexity that make simplification difficult:

- One aspect is that because of limited image resolution and perspective projection, typically several triangles project to a single pixel. This makes antialiasing and filtering an important issue.
- Another aspect is that we can not rely on surfaces to drive the simplification process: the triangles that project to one pixel can stem from disconnected surfaces and even from different objects. This means, for example, that there is no well-defined normal for such a pixel.

Generally, simplification can be based either on geometric criteria as in most level-of-detail approaches, or on image-based representations. Geometric simplification is useful for certain models, but has its limitations. Scenes with alpha textures, regular structures as found in buildings, chaotic disconnected structures as found in trees, or new shading paradigms such as pixel and vertex shading are not easy to deal with even in recent appearance-based simplification methods. We will therefore concentrate on image-based simplification approaches.

We observe, however, that none of the simplified representations proposed to date meets all the goals defined above:

- Layered depth images (LDIs) and similar representations introduce bias in the sampling patterns, and cannot accommodate situations in which many primitives project onto each pixel of a view. In the presence of such *microgeometry*, we also have to be aware that appearance can change with the viewing direction. This requires a representation with directionally dependent information, such as a light field.
- Light field approaches are powerful enough to cope with complex and directionally dependent appearance. However, geometric information needs to be incorporated into the light field in order to obtain acceptable quality within a reasonable amount of storage space [Chai00]. A possible starting point to build an impostor would therefore be a surface lightfield [Wood00],

but as the name of this primitive already indicates, this requires a surface (and its parameterization).

In this chapter we introduce a new representation for a complex geometric model that is especially useful to build impostor objects to replace the far field for a given view cell. The proposed representation effectively decouples the geometry, represented as a set of 3D points, and the appearance, represented as a simplified light field computed from the original model. In our algorithm, the points are selected from a number of viewpoints, chosen so as to approach a minimal sampling criterion in image space, across the entire view cell. Therefore our sampling mechanism prevents the appearance of holes. In the context of very complex models composed of a great many independent primitives, 3D points can not be considered as representatives of a well-defined surface, as in the surfel or QSplat techniques [Pfis00, Rusi00]. In contrast, we define a proper characterization of the radiance contribution to the image from these points, which can be computed using Monte Carlo integration, and is encoded as a texture map to model view-dependent effects. The resulting representation is compact, renders quickly using existing graphics hardware, produces a high quality image with little aliasing, has no missing or disappearing objects as the viewpoint moves, and correctly represents view-dependent changes within the view cell (such as occlusion/disocclusion effects or appearance changes).

After a short overview of the proposed representation, its construction and its usage (section 5.2), section 5.3 provides an in-depth analysis of the point sample rendering primitive, its meaning in terms of appearance modeling and theoretically founded means of representing image radiance information at these points. Section 5.4 presents the details of the current implementation, which computes impostor representations for the far field. Results are presented and discussed in section 5.5.

5.2 Overview of the algorithm

We propose a rendering primitive based on points to replace geometry as seen from a view cell. The system implements the following pipeline:

- Sampling geometry: we calculate sample points of the geometry using three perspective LDIs to obtain a dense sampling for a view cell (figure 5.2). We call the plane defined by the three camera locations the *sampling plane*.
- Sampling appearance: The antialiased appearance of each point is calculated for different view cell locations using Monte Carlo integration. We

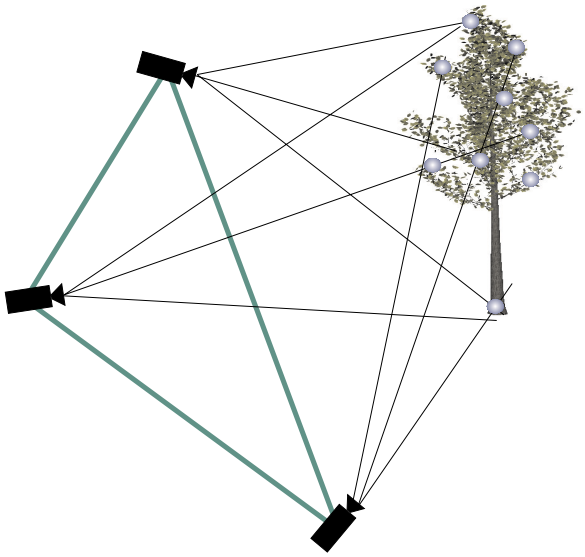


Figure 5.2: Geometry is sampled using three perspective LDI cameras from a view cell.

shoot rays from a rectangular region which is contained in the triangle formed by the three cameras (see figure 5.3). Each point is associated with a texture map which encodes the appearance contributions for different view cell locations.

- Real-time display: point-based impostors make full use of existing rendering hardware for transforming each point as well as shading it with its associated texture map depending on the current viewer location. No additional software calculations are necessary.

5.3 Point-Based Impostors

5.3.1 Complexity of appearance

The plenoptic function $P(s, \varphi)$ completely models what is visible from any point s in space in any given direction φ (see section 2.3.1). Rendering is therefore the process of reconstructing parts of the plenoptic function from samples. Both geometry and appearance information of scene objects contribute to the plenoptic function. Current rendering algorithms usually emphasize one of those aspects:

- Light fields record rays regardless of geometric information, even if they all hit the same diffuse surface.

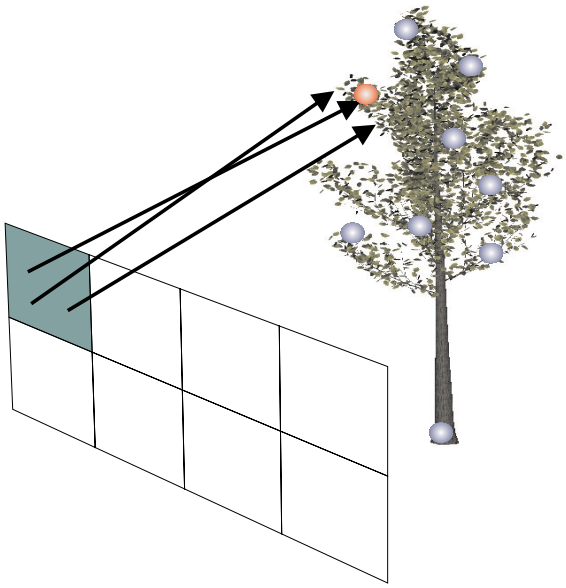


Figure 5.3: Rays are cast from the view cell to calculate a Monte Carlo integral. A texture map on the sampling plane records contributions for different view cell locations.

- Other, geometric approaches usually model appearance only to a degree allowed by their lighting model. They cannot account for *microgeometry*, i.e., geometry that projects to less than a pixel. Microgeometry can have different appearance when viewed from different angles (see figure 5.4). This usually results in aliasing artifacts or is dealt with by blurring, thereby discarding possibly important information.

The proposed point-based representation contains both geometric information (the 3D location) and directionally dependent appearance information. It therefore captures all of these aspects:

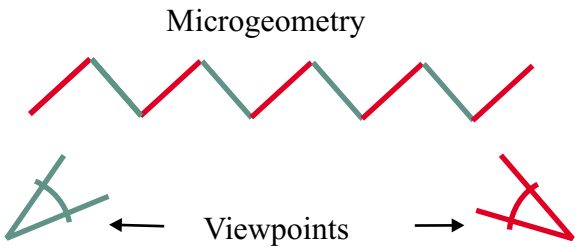


Figure 5.4: Microgeometry: if the structure in the figure projects only to a pixel for the view cell, directional appearance information is required to let it appear green in the left camera and red in the right camera.

- Point sampling can easily adapt to unstructured geometry in the absence of surfaces or a structured model.
- A point on a diffuse, locally flat surface which projects to more than a pixel has the same appearance from all directions and can be encoded with a single color.
- Objects with view-dependent lighting effects (e.g. specular lighting) show different colors when viewed from different angles.
- Microgeometry is also view dependent and is encoded just like lighting effects.

5.3.2 Geometric sampling

The first step in the creation of a point-based rendering primitive is to decide on the geometric location of the sample points. The points encode the part of the plenoptic function defined by the view cell and the geometry to be sampled. To allow hardware reconstruction, there should be no holes when projecting the points into any possible view from the view cell.

A sufficient criterion that a point-sampled surface will have no holes in a projection is that the projected points can be triangulated so that the maximum projected edgelength is smaller than the side length of a pixel [Gros98].

It is in general difficult to prove that a particular point-based representation fulfills this criterion. For unit magnification and orthographic viewing, three orthogonal LDIs provide such an adequate sampling [Lisc98]. Although this sampling strategy works with our method, it is not well suited for far-field objects because the sampling density is not adapted to the possible viewing directions from the view cell and typically results in a large number of point samples. Therefore, our method chooses perspective LDIs to better distribute the samples with respect to the view cell.

The camera viewpoints are chosen so as to form a triangle which bounds the shaft formed by the view cell and the far-field object (see also figure 5.6). In this way, the sampling rays from the three cameras most closely resemble the rays possible from a viewpoint inside the view cell.

Note that three cameras are required in order to adequately sample arbitrarily oriented surfaces. To see why this is necessary, assume for example that only the left and right camera were used. In this case, any surface lying on a plane through the centers of projection of the two cameras would not be sampled at all (if the

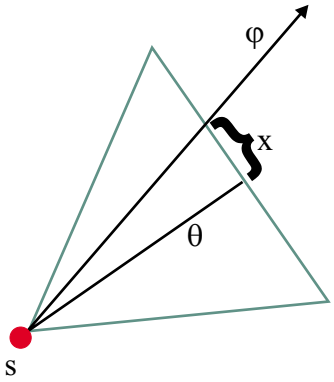


Figure 5.5: The figure shows the relation of the plenoptic function and plenoptic image function parameters.

two cameras have the same view direction) or very sparsely—thus, a third camera is required.

5.3.3 Appearance: the plenoptic image function

The geometric sampling step results in a set of n points $p_1, \ldots, p_n$ with fixed locations in 3D space. In this section, we derive a method to determine a color value for each of these points. We examine the meaning of a point in light of our goal: to reconstruct slices of the plenoptic function $P(s, \varphi)$ corresponding to different viewing locations and camera positions in a view cell. The derivation will take into account the rendering algorithm used to render the point.

Most rendering algorithms use a z-buffer or depth ordering to determine the visibility of the primitives they render. A point, however, is infinitely small. Its meaning is mainly determined by its reconstruction. In many reconstruction methods (as, for example, in hardware rendering) only one point is visible per pixel and determines the color of the pixel. Due to finite image resolution, a point can be visible or occluded depending only on the viewing camera orientation. The plenoptic function is not powerful enough to model this form of occlusion.

Consequently, we introduce the *plenoptic image function* $PIF(s, \theta, x)$. The parameter s represents the 3D viewer position, θ is a camera orientation, and x is a 2D pixel coordinate in the local camera coordinate system. Figure 5.5 illustrates how the domains of the *PIF* and the plenoptic function relate to each other: any pair (θ, x) corresponds to one ray orientation $\varphi(\theta, x)$ from the plenoptic function, so under ideal conditions (i.e., infinitely precise displays etc.), the *PIF* is related to the plenoptic function P via

$$PIF(s,\theta,x) = P(s,\varphi(\theta,x))$$

Note that this mapping is many to one. The *PIF* with its additional parameters will allow us to incorporate the visibility term inherent to current reconstruction techniques directly into our calculations.

We now interpret points via the images they produce (their image functions) when rendered with a specific rendering algorithm—in our case, z-buffered hardware rendering.

If we consider only one point p_j alone, the point defines an image function $r_j(s,\theta,x)$. This function specifies a continuous image for each set of camera parameters (s,θ). Each of those images is set to the typical reconstruction filter of a monitor, a Gaussian centered at the pixel which the point projects to. However, it is very crucial to see a point p_j in relation to the other points in the point set. We have to take into account a visibility term v_j which evaluates to 1 if the point p_j is visible in the image defined through the camera parameters (s,θ) and 0 otherwise. This gives us the actual image function Q_j of a point:

$$Q_j(s,\theta,x) = v_j(s,\theta) r_j(s,\theta,x)$$

From an algebraic point of view, we can regard the functions $Q_j, 1 \le j \le n$, as basis functions spanning a finite dimensional subspace [1] of the space of all *PIF*s. One element PIF_{finite} of this finite dimensional subspace can be written as a weighted sum of the basis functions of the points:

$$PIF_{finite}(s,\theta,x) = \sum_j c_j Q_j(s,\theta,x)$$

Note that the weight c_j is the color value assigned to the point p_j. The weights c_j should be chosen so as to make PIF_{finite} as close as possible to *PIF*. This can be achieved by minimizing $\|PIF_{finite} - PIF\|$ (with respect to the Euclidean norm on the space of functions over (s,θ,x)), and the resulting weights can then be found via the dual basis functions as

$$c_j = \iiint q_j(s,\theta,x) PIF(s,\theta,x) ds\, d\theta\, dx$$

where $q_j(s,\theta,x)$ is the dual basis function (the dual basis functions are defined via $\langle q_j, Q_k \rangle = \delta_{jk}$).

[1] The finite dimensional subspace describes all different images that can be produced by the particular set of geometric samples (points) from the geometric sampling step.

This representation only assigns one color value per point, regardless of the location s in the view cell. However, because of view-dependent shading effects and micro-geometry as discussed in section 5.3.1, a point might need to show very different appearance from different viewing locations. Thus, we make the model more powerful by replacing the weights c_j by functions $c_j(s)$. In order to represent those functions, they are discretized with respect to the space of possible viewing locations (this is similar in spirit to the camera aperture necessary when sampling light fields). Given a basis $B_i(s)$ for this space, we can write each $c_j(s)$ in terms of this basis, with coefficients found via the dual basis $b_i(s)$:

$$c_j(s) = \sum_i c_{ij} B_i(s), \text{ where } c_{ij} = \int c_j(s) b_i(s) ds$$

One possible choice for the basis $B_i(s)$ is to use a regular subdivision of a plane with a normal directed from the view cell towards the objects. The weights c_{ij} for each point p_j can then be stored in a texture map. This means that $c_j(s)$ will be looked up in a texture map, which is typically reconstructed with a bilinear kernel, so $B_i(s)$ is actually just a bilinear basis.

Putting all parts together, the rendering process is described with the formula

$$PIF_{finite}(s, \theta, x) = \sum_{i,j} c_{ij} Q_j(s, \theta, x) B_i(s)$$

and the weights to make the PIF_{finite} resemble most closely the PIF are calculated as

$$c_{ij} = \iiint PIF(s, \theta, x) q_j(s, \theta, x) b_i(s) ds \, d\theta \, dx \tag{5.1}$$

The final task is finding the dual basis functions. If our basis functions $Q_j B_i$ were orthogonal, we would have $q_j = Q_j$ and $b_i = B_i$ (apart from normalization issues). In the non-orthogonal case, however, calculating the real duals is tedious: geometric occlusion means each one would be different; one would really have to invert a large matrix for each one to find a (discretized) version of the dual. We have opted for an approximate approach inspired by signal theory: both the bilinear basis B_i and the monitor reconstruction filter r_j can be regarded as an approximation to the ideal (and orthogonal) reconstruction filter, a sinc-function. As is common practice, we approximate the ideal lowpass filter (the signal-theoretic version of dual basis functions) using Gaussians.

The integral (5.1) can be estimated using Monte Carlo integration with b_i and q_j as importance functions. Samples $PIF(s,\theta,x) = P(s,\varphi(\theta,x))$ are calculated with a simple ray tracer.

5.4 Implementation

In this section we describe the choices made in our particular implementation of the method.

5.4.1 Obtaining geometric samples

Before placing the three LDI cameras used to obtain geometric samples, we choose a sampling plane to parameterize viewer positions within the view cell: as in light fields, we use a plane with a normal oriented from the view cell towards the far field. The *supporting planes* [Coor99] of view cell and far-field object bounding box form a shaft between the two. The supporting planes can be found for example by iteratively creating planes between an edge of the view cell and a vertex of the far-field bounding box, and testing whether both the view cell and the bounding box lie on the same side of the plane (and vice versa with an edge of the bounding box and a vertex of the view cell).

Then, we calculate the intersection of the shaft with the sampling plane. The resulting polygon on the sampling plane can be tightly bounded with a triangle (we use the triangle with the top edge parallel to the ground and minimum circumference). The three LDI cameras are now placed at the corners of this triangle (see figure 5.6). Note that camera placement is generally well behaved, as long as the view cell does not intersect the far-field bounding box (which should never occur).

In order to determine the LDI resolution necessary to avoid holes in the reprojection of the sampled points, our method offers a sampling mechanism based on a 2D observation: Suppose we have two cameras on both ends of a segment of possible viewing locations in 2D, and a short edge viewed by the two cameras. These two cameras are used to sample the endpoints of the edge. Resampling happens from a camera placed anywhere on the segment between the two cameras. It can now be shown that the "worst" discrepancy between the resampling angle α and the sampling angle β appears if the edge is parallel to the segment joining the two cameras, and if the center of the edge is equidistant to both cameras (see figure 5.7).

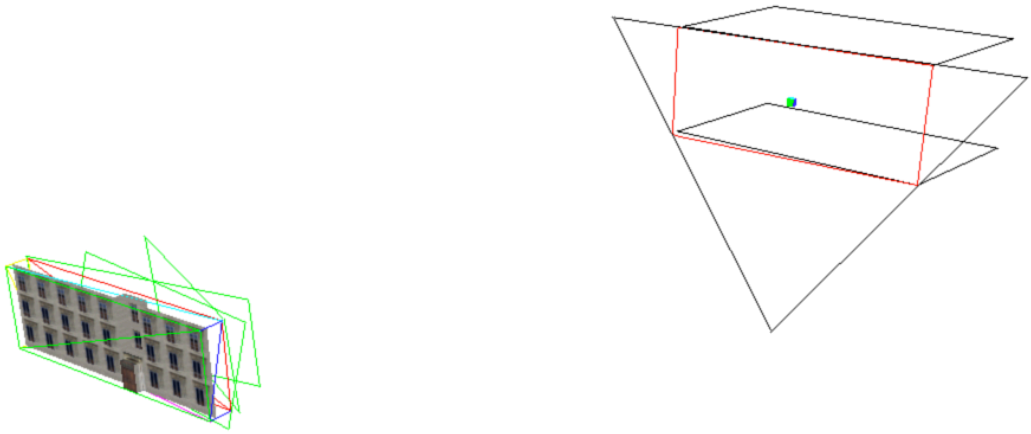


Figure 5.6: A screenshot showing the arrangement of an impostor object, the view cell, camera placement and the sampling plane. The view cell is a box. The red rectangle is the sampling plane which is directed towards the impostor. The cameras are placed at the corners of the triangle that tightly surrounds the sampling plane. The green rectangles on the object bounding box constitute the actual projection planes used by the cameras.

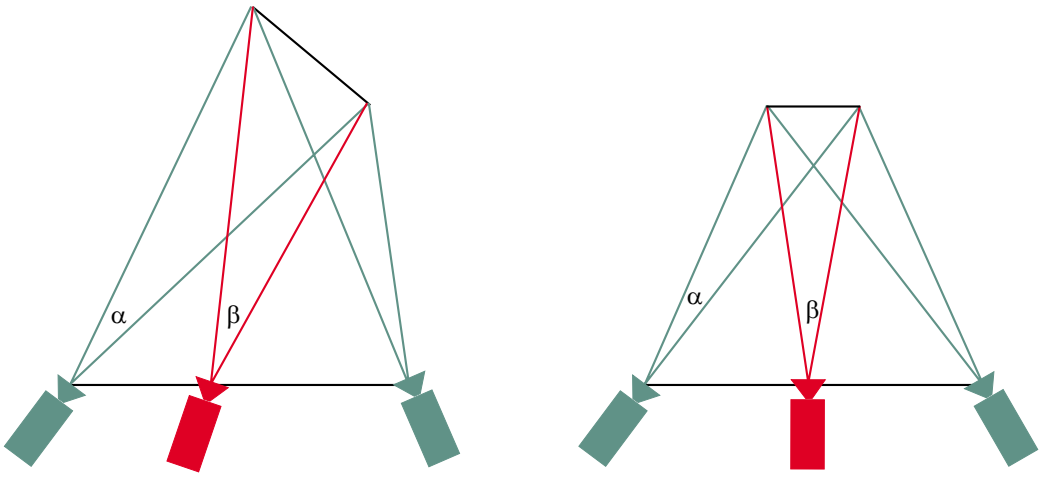


Figure 5.7: The *left* figure shows an arbitrary configuration of sampling (green) and resampling (red) cameras. In the *right* configuration the ratio β/α is maximized. This means that to obtain an angular resolution of at β when resampling, the object has to be sampled using an angular resolution of at least α.

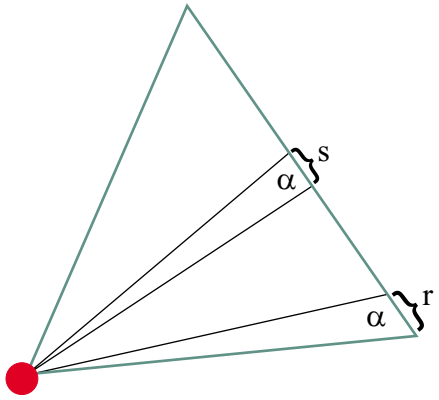


Figure 5.8: Sampling resolution caused by rotation and perspective cameras: a feature seen under the angle α can project to r at the corner of the screen and s at the center of the screen. To ensure the projection for resampling is not larger than r, the pixel size for sampling has to be chosen smaller than s.

Inspired by this, we have developed a heuristic in 3D which takes the following effects into account:

- Movement within the view cell: for three cameras looking at a plane parallel to the sampling plane, the worst mutual sampling resolution occurs at the point on the plane that is equidistant from the three cameras. This point is the circumcenter of the triangle defined by the three cameras, projected on the plane in question. The sampling angle of the three cameras has to be chosen so that neighboring samples project to less than a pixel when viewed directly from the circumcenter (where the maximum projection occurs).

- Perspective: perspective cameras have varying angular resolutions in space: a feature projects to more pixels at the border of the viewing frustum than in the center when seen under the same viewing angle α. The sampling resolution has to be increased accordingly by the factor between angular resolution in the center of the camera and at the border of the viewing frustum. For a field of view of 45° for example, this factor is about 1.17 (see figure 5.8).

- Sampling pattern orientation: if perspective cameras are used for sampling, the sampling pattern on objects in space is tilted with respect to the viewing camera. Therefore, given a minimum sampling angle from a point, the resolution has to be calculated from the pixel diagonal and not the pixel edge. This accounts for a factor of $\sqrt{2}$.

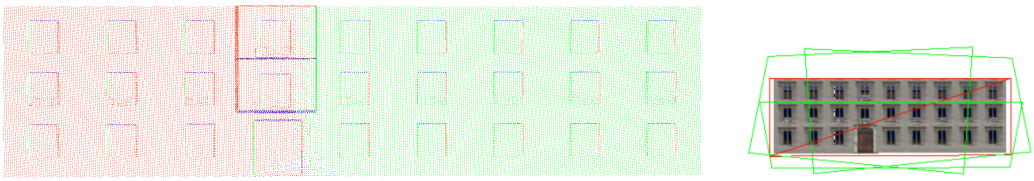


Figure 5.9: The *left* side shows a false color image of a point-sampled building front after removing all points better sampled by a different camera. Red points are from the left camera, green points from the right camera, and blue points from the bottom camera. The *right* side shows the geometry together with the three camera image planes. Note that for the windows, left-facing surfaces are sampled best with the left camera and right-facing surfaces with the right camera.

Sampling resolution is chosen according to these factors. In practice, we employ a simple heuristic to reduce the number of points: many points are sampled sufficiently already by a single camera. This means that a triangulation of the sampling pattern from this camera in a small neighborhood contains no edge which projects to more than a pixel in any view. If a point recorded by one camera lies in a region which is sampled better and sufficiently by another camera, we remove it. This typically reduces the number of points by 40–60% (see figure 5.9), leading to a ratio of about 2–2.5 points projected to a screen pixel (see section 5.5 for more detailed results).

5.4.2 Monte Carlo integration of radiance fields

The goal of the Monte Carlo integration step is to evaluate integral (5.1) to obtain the weights c_{ij} of the reconstruction basis functions. The domain in s is a rectangle on the sampling plane. The index c_{ij} corresponds to one point p_j and one rectangle on the sampling plane represented by a texel t_i in the texture map for point p_j.

We select a viewpoint s on this rectangle (according to $b_i(s)$, a Gaussian distribution), a random camera orientation θ in which the point is in the frustum, and shoot an occlusion ray to test whether this point is visible in the selected camera (this corresponds to an evaluation of $v_j(s,\theta)$). If it is visible, we select a ray according to $q_j(s,\theta,x)$ (a Gaussian distribution centered over the pixel which the point projects to in the selected camera) and add its contribution to t_i. Rays are shot until the variance of the integral falls below a user-selectable threshold.

5.4.3 Compression and Rendering

Points can be rendered with z-buffered OpenGL hardware. For each point, the directional appearance information is saved in a texture, parameterized by the sampling plane. The texture coordinates of a point are calculated as the intersection of a viewing ray to the point with the sampling plane. This can also be interpreted as the perspective projection of the sampling point into a viewing frustum where the apex is defined by the viewpoint and the borders of the projection plane by the four sides of the bounding rectangle on the sampling plane (see figure 5.10).

Perspective projections can be expressed using the 4x4 homogeneous texture-matrix provided by OpenGL. However, since switching textures for every point is costly, we pack as many point textures into one bigger texture as the implementation allows (see figure 5.11). This requires adding a fixed offset per point to the final texture coordinates, which, although not available in standard OpenGL, can be done using the vertex program extension [NVID00a].

Interpolation between the texture samples (which corresponds to evaluating the basis function $B_i(s)$) is done using bilinear filtering. A lower quality, but faster preview of the representation can be rendered by using only one color per point and no texture. However, directional information will be lost in this case.

To compress our representation, we calculate the variance of the color information of a texture to identify points that only require one color for the whole view cell. Further compression can be obtained by using hardware supported vector quantization [NVID00b], which provides for a fixed compression ratio of 8:1. Note that this adapts to the scene complexity: regions with low perceived geometric complexity will always be represented by simple colored points.

5.5 Results

We used an 800 MHz Pentium III with a GeForce II GTS graphics card for our tests. The vertex program used to calculate texture coordinates is simulated in software. To give an impression of the expected performance of a hardware implementation, we also rendered simple textured points.

Three different test scenes are used to demonstrate the behavior of point-based impostors. Table 5.1 shows results for each scene, based on an output screen resolution of 640x480 pixels. It includes the number of points in the impostor, the approximate number of pixels covered by the impostor for a view in the center of the view cell (based on the screen bounding box), and the resulting number of points per projected pixel. The table also shows the time required for computing

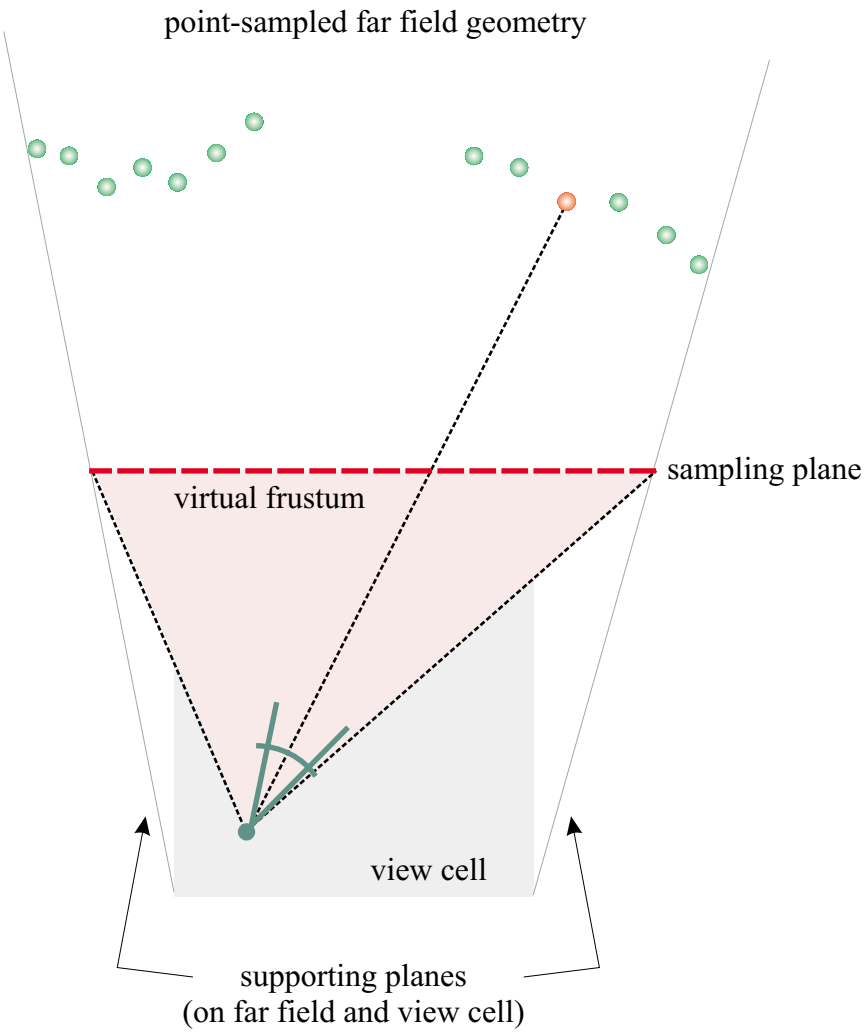


Figure 5.10: The texture coordinates for a point can be found using the off-axis frustum shown in the figure: the texture matrix is set to implement this frustum, and the current point is used as source texture coordinate. After transformation with the texture matrix, the texture coordinates correspond to the correct location on the sampling plane.

Figure 5.11: A 232x128 pixel (29x64 points) part of the packed texture for the impostor in figure 5.13. For each point, there is an 8x2 block in this packed texture. Note the bright blocks from the specular highlight. The inlay in the lower right corner is a zoomed 24x16 pixel part of the texture, showing the blocks for 24 points.

the impostor, and the memory requirements. The last two rows represent average frame rates achieved for some positions within the view cell.

Generally, it can be observed that it takes about 80,000 points to cover a screen area of about 300x100 pixels.

Although an unoptimized ray tracer (based on regular grid acceleration) is used, preprocessing times are still reasonable. For each point, an 8x2 texture map is used. The memory requirements are listed without applying the variance-based reduction of textures to single colors.

One of the strong points of our representation is high rendering quality in the presence of detailed geometry. Figure 5.13 shows the filtering of many thin tree branches in front of a moving specular highlight. Figure 5.14 demonstrates correct antialiasing even for extreme viewing angles on building fronts.

Figure 5.15 shows how a point-based impostor can be used to improve the rendering quality of the far field in an urban scene which contains several polygons per pixel. It should be noted that the improvement over geometry is even more noticeable when moving the viewpoint. Furthermore, the impostor in figure 5.15 replaces a geometric model of about 95,000 vertices, but consists only of about 30,000 points. This shows that the impostor not only improves the rendering quality of the far field, but also reduces the rendering load on the graphics

Results	scene1	scene2	scene3
#points	80,341	87,665	31,252
#points/screen pixel	2.35	2.42	2.8
approx. #screen pixels	34,000	36,000	11,000
Preproc. time (min)	22	41	31
Memory (MB)	1.6	1.75	0.6
Rendering performance SW (Hz)	36	32	98
Rendering performance HW (Hz)	60	54	160

Table 5.1: The table shows results from impostors of three scenes, calculated for a screen resolution of 640x480 pixels. Hardware rendering is emulated by rendering simple textured points.

hardware. Figure 5.12 shows the placement of this impostor and its view cell in the city model and gives an impression of the typical relative sizes of view cells and impostors.

In the current experimental system, view cells are formed from street segments, and impostors placed at the ends of street segments, in a fashion similar to previous impostor systems [Sill97, Deco99]. Our test scene is 4 square kilometers large and consists of 2.1 million polygons. After the application of a conservative region-visibility algorithm [Wonk00], we identified view cells with too many visible polygons (as was demonstrated in a previous system [Alia99b]). A rough estimate of the memory requirements using a brute force impostor placement strategy results in about 165 MB used for 437 impostors.

Note, however, that the development of good impostor placement strategies is not straightforward and subject to ongoing research. Future work also aims at reducing memory requirements by sharing information between neighboring view cells, and finding guarantees to ensure a minimum frame rate.

5.6 Discussion

This chapter introduced point-based impostors, a new high-quality image-based representation for real-time visualization.

The value of this representation lies in the separation between the geometric sampling problem and the representation of appearance. Sampling is performed by combining layered depth images to obtain proper coverage of the image for the entire view cell. Based on the mathematical analysis of point-based models, we compute the rendering parameters using Monte Carlo integration, eliminating

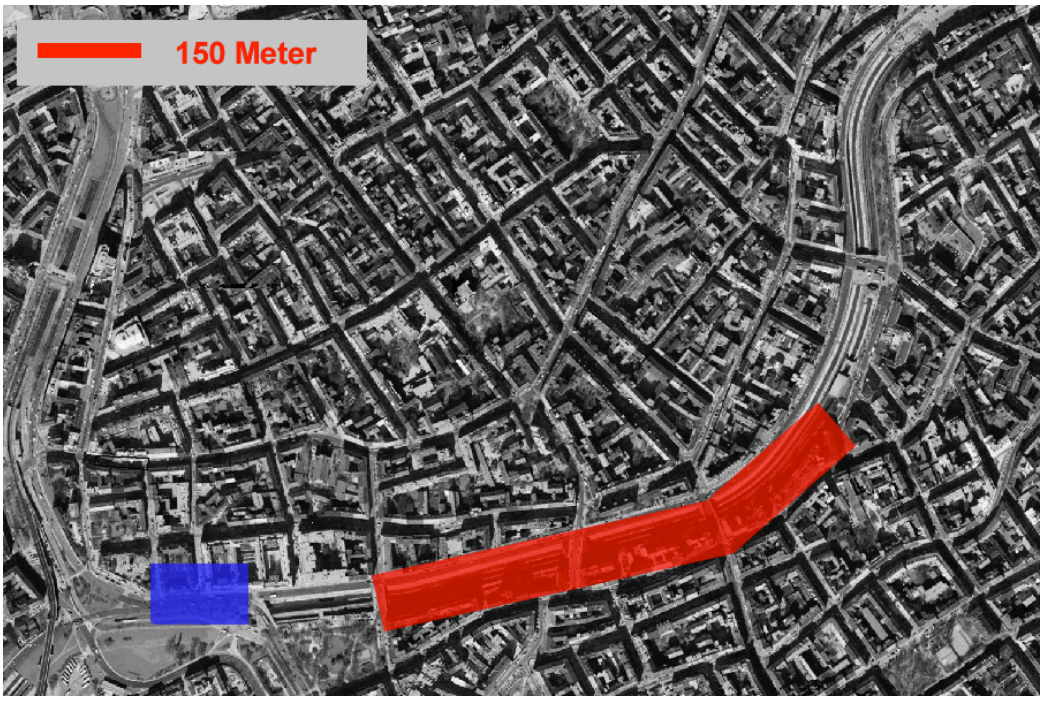


Figure 5.12: The figure shows the impostor from scene 3 placed in the city. The view cell shown in blue is 63 meters wide and 120 meters long. The impostor shown in red is about 700 meters long and placed 200 meters from the view cell.

Figure 5.13: Note the correct filtering of the trees against the building front and the specular highlight in the windows for the impostor (top), and severe aliasing in the trees for geometry (bottom).

most of the aliasing artifacts. Rendering information is compactly encoded and can be rendered on contemporary hardware. Point-based impostors show great promise for all situations in which a geometrically very complex model constitutes the far field, such as in urban walkthroughs of detailed models. The rendering times indicate that this representation is applicable to real-time visualization applications, where frame times above 60 Hz are required.

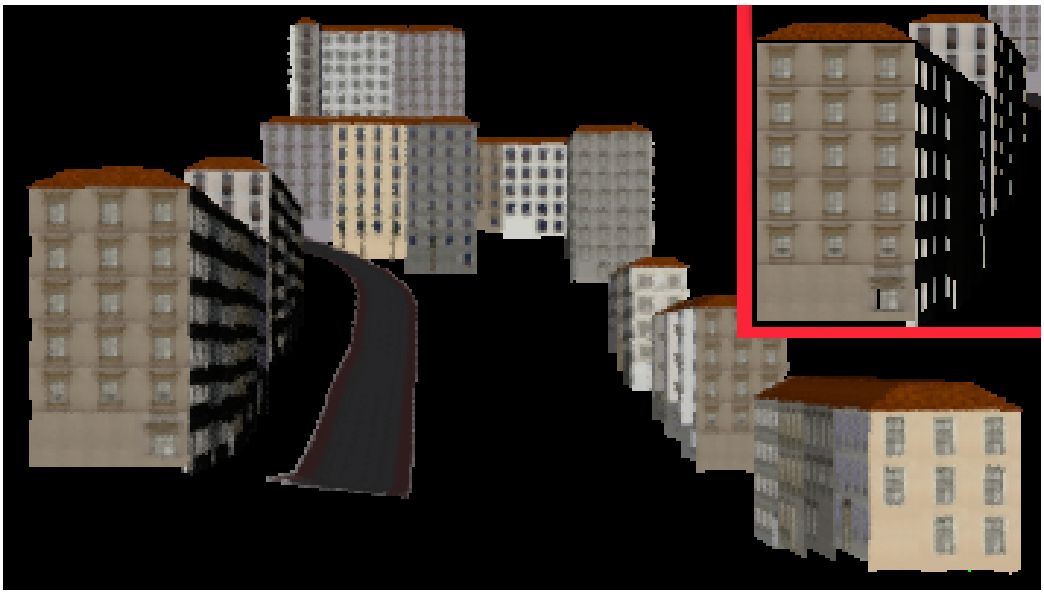


Figure 5.14: An impostor for a number of buildings. The inset shows the aliasing that would result if geometry were used.

Figure 5.15: One frame of a city walkthrough. Note the correct filtering of the impostor (top) compared to geometry (bottom).

Chapter 6

Conclusions and Future Work

Although computers continue to get faster and faster, real-time rendering is far from being a solved problem. As long as the geometric model of a car or a simple mechanical part is able to overwhelm the rendering capabilities of the fastest available graphics hardware, this situation is not going to change. Currently, it is possible to render small urban models, but at the expense of visual richness—we are therefore still striving for bigger models, and more visual detail at the same time.

The quest for true real-time rendering on arbitrary models is marked by the necessity to identify opportunities to reduce complexity. This automatically brings the human visual system and its interface to the computer—images—into play: Details can only be omitted if they are not important to the viewer, wherever he is located. Purely image-based techniques follow this road to the end and equate perceived complexity with the resolution of an image, and totally abandon geometry. Such approaches have their limitations, however, because viewer locations cannot be easily predicted in a real-time application, and the complexity of many images can quickly be disproportionate to the complexity inherent to the geometric model.

In this thesis, we take a different approach. Instead of being bound to the complexity of the underlying model, or the complexity of an image, we exploit a general property of several types of scenes: due to perspective viewing, complexity per screen area increases with increasing distance, but at the same time, the importance of distant objects diminishes. Distant objects are therefore perfectly amenable to simplification. This thesis presents two ways to do exactly that: simplify distant geometry.

6.1 Synopsis

The topic of this thesis is a set of algorithms for fast and high quality display of distant geometry. Two major algorithms were presented: an online method that requires no preprocessing, and an offline method that offers very high visual quality.

The online method is based on the observation that in many walkthrough scenarios, only a small area of the screen is covered by actual distant geometry. The pixels in that area are found via an opacity buffer, and a horizon map is used to include only pixels which belong to actual geometry and not to sky. A panoramic image cache allows rotational motion with correct reprojection. Translational movement exploits the coherence between successive images to keep errors low. Finally, incorrect pixels are traced via a fast ray caster.

The offline method uses precomputed point-based impostors to represent distant geometry. The impostors are calculated so that they provide a good approximation to the real geometry for a specific view cell. Point samples are obtained by carefully sampling the geometry with three perspective LDIs, ensuring a well-distributed density of point samples over the range of valid viewpoints. Holes are avoided by tightly oversampling the LDIs depending on the desired output resolution. A new mathematical formulation for the appearance of a point is shown, based on the plenoptic image function. This allows us to encode directionally dependent appearance information in a texture map for each point. The texture values are chosen so as to provide the best possible appearance of the impostor for any viewpoint located in the view cell. The result is an antialiased representation for distant geometry that can be fully rendered with current graphics hardware.

The remainder of this chapter will deal with the specific advantages and disadvantages of the two methods. Some limitations are discussed, and finally, a range of suggestions for future research is presented.

6.2 Advantages

Distant geometry can be enormously complex, and both methods presented result in a drastic reduction of geometric primitives that have to be rendered overall. This can be used twofold: It can make a scene practical that was originally too complex to be rendered interactively. For other scenes that can be rendered interactively already, it allows adding visual detail to the near field and making the environment much more appealing.

Both methods introduce the notion of output sensitivity into the setting of large scenes. While the distinction between near field and far field is not adaptive, it is still valid to assume that near-field geometry is of bounded complexity. In the first method, far-field geometry is recognized directly in the image, therefore the rendering time for the far field only depends on the number of pixels covered by it. The complexity of point-based impostors, on the other hand, can not be expressed in terms of the pixels of a single frame, as such an impostor has to be valid for all viewpoints of a view cell. Their complexity is, however, output sensitive in the sense that only "relevant" pixels are taken into account when constructing the representation, i.e., pixels visible in any possible view from the view cell.

While output sensitivity is usually touted as a desirable feature of image-based methods, the algorithms in this thesis avoid a common pitfall associated with this: if there is too much "output", then a strictly "output-sensitive" method will still be slow, even if the underlying scene complexity is little. By concentrating on geometry that inherently covers only few pixels on the screen (i.e., little "output"), we make sure that output sensitivity is really desirable.

The ray casting method from chapter 4 is easy to set up and integrate into an existing rendering system. There is no preprocessing overhead, so the scene may be changed often or even dynamically. Furthermore, the technique is amenable to parallelization and can profit from a wealth of ray tracing acceleration research.

6.2.1 Specific advantages of point-based impostors

Point-based impostors, on the other side, require preprocessing and are therefore limited to applications with little changing scenery. They blend with traditional rendering techniques, so dynamic objects can always be rendered using polygons, even in the far field. Two significant advantages of point-based impostors are that they provide much higher quality rendering, and that they do this in much less time, because no online calculations are necessary. More precisely, the small amount of computations left for each frame can be offloaded to the graphics hardware, incurring almost no additional cost. Point-based impostors are particularly suited for distant geometry, because the mathematical formulation of points specifically considers microgeometry, i.e., geometric features that project to less than a pixel. For distant geometry, several triangles usually project to a single pixel.

We are aware of no other computer graphics method that can simplify distant geometry in such a way, i.e. with:

- correct warping of geometry (so parallax effects are not lost)

- capturing of view-dependent effects such as specular highlights
- correct representation of microgeometry
- inherent antialiasing
- fast, hardware-accelerated rendering

For the purpose of real-time rendering, point-based impostors can be seen as the missing link between purely image-based representations such as light fields, and purely geometry-based representations. Directional radiance information is stored where it is needed, i.e., at the geometric location of the real object. Unlike surface light fields, however, no restriction on the type of geometry exists—for example, trees can be nicely captured and rendered using point-based impostors. Unlike other approaches to incorporate geometry into light fields [Gort96, Chai00], this parameterization of light fields (i.e., via points) can exploit the coherence in directional radiance usually found for many object locations. On a diffuse surface point, only one value is necessary to store all the information. In the presence of microgeometry, the directionally dependent appearance information can help to create the correct view of the "soup" of polygons from different viewpoints.

6.3 Disadvantages and limitations

The techniques presented do have some limitations and offer ample room for future research. Both methods are not fully general, i.e., they do not work for arbitrary scenes. A certain amount of occlusion is required to make the distribution between near field and far field work. This is to be expected, most acceleration algorithms are designed to exploit certain features of some class of scenes.

The ray casting method, in particular, suffers from a considerable amount of online computation, which reduces the time available to render triangles. This could be alleviated by using multiple CPUs to compute the opacity map, image cache lookups and the rays themselves. The ray casting approach also depends on the efficiency of the ray caster for a particular scene. While most ray tracing optimization schemes achieve sublinear performance with many scenes, severely non-uniform scenes or other worst-case scenarios can easily thwart any optimization scheme.

The image quality of ray casting is at best equal to rendering geometry. If too many pixels are in the far field and the method resorts to graceful degradation, not all necessary rays can be cast every frame. The result is a kind of cross-dissolve

effect on edges that move in the image (due to translational movement of the observer). The effect looks similar to frameless rendering [Bish94], a technique where pixels are updated independently of the display refresh signal. We observed graceful degradation mainly if the border between near and far field was set too near, and in wide open spaces.

Point-based impostors can be said to lie at the other end of the spectrum. They offer vastly superior image quality to the ray casting technique and even to standard geometry rendering, and almost no overhead during runtime. The cost for this quality and efficiency is a significant amount of preprocessing time. To compute all impostors for a larger urban scene, several hours of precomputation might be common.

Furthermore, point-based impostors cannot be used simply as a drop-in component to an existing rendering system. They require a good partitioning of the viewing space into view cells. Such a partitioning might be straightforward in the case of some urban environments, but in general, the size and location for good view cells is not obvious.

As in any offline method, there is the issue of storage space. Impostors for a larger city model might take up several hundreds of megabytes, depending on visibility: if wide open views exist in many locations, more impostors have to be used. If only a few locations offer views that stretch far away, the number of impostors will be limited. This problem is directly related to the partitioning into view cells. Since impostor information is not shared between view cells, a fine-grained view space partition can significantly increase storage requirements.

Finally, points might not be the ultimate answer to far-field rendering. Since we rely on a sufficiently dense sampling of objects to avoid holes during reconstruction, an impostor usually requires twice as many points than pixels in a projection from the view cell. This adds up to a lot of points, given that many of those points will not significantly enhance the depth effect of the impostor. Neither is it possible to approach the impostor nearer than the view cell allows, for example if a close-up view is required and a new model (e.g., a new impostor for a different view cell or simply geometry) is not yet available. In such a case, the representation would show holes. Splatting with a kernel greater than a pixel could be used in such a case, but only square splat kernels are available in hardware, providing only mediocre quality.

A representation of the far field using triangles might be more desirable because it maps better to current graphics hardware and has the ability to exploit hardware interpolation and rasterization. Indeed, such representations have been proposed [Sill97] for city models. However, we are aware of no triangle-based far-field representation that correctly captures geometry and appearance attributes

of the far field and that can deal with microgeometry. Point-based impostors can deal with arbitrarily complex geometry and do not rely on coherence of depth values from any viewpoint. They avoid common artifacts like rubber-sheet triangles, occlusion and disocclusion errors and others. Still, a triangle-based solution should be more efficient to render and take up less storage space.

6.4 Future work

In this section, future research directions and potential derivations from this work are discussed.

6.4.1 Triangle-based impostors

First and foremost, we believe that future research should concentrate on doing for triangles what this thesis did for points. As already hinted at in the previous section, triangles map better to current hardware (and this situation is not likely to change in the near future). Triangles could exploit the rasterization capabilities of graphics cards, while points only make use of geometry acceleration. The following problems would have to be solved:

- Geometric sampling of the far field (the geometric sampling method presented in this thesis would serve as a good starting point).
- Creation of connectivity (i.e., decide where to put triangles): this is actually a difficult task, because no unique solution exists. The triangles would have to faithfully represent the depth structure of the object from all points of the view cell.
- Simplification of the resulting triangle mesh. Standard polygonal simplification tools could be used for this step, but it is likely that better results can be achieved if the error metric used in the simplification process is adapted to the possible views on the object—this could be called “view cell-dependent” simplification.
- Calculation and storage of appearance information in the resulting mesh, preferably with textures. This might of course influence the simplification step as well.

Point-based impostors avoid having to deal with these problems, at the cost of increased complexity (in terms of number of points) of the representation.

6.4.2 View cell partitioning

This thesis has introduced point-based impostors, a primitive destined to replace distant geometry, providing fast and high quality rendering of such geometry. We have, however, not dealt with the question where to place these impostors in an actual city model. This leads directly to the second open question, how to partition the view space into view cells?

Both of these questions warrant closer attention in future research, not only for point-based impostors, but for many impostor-based methods. Sillion et al. [Sill97] have proposed basing the view-space partitioning on the street graph. In this thesis, we have loosely followed this suggestion and obtained good results, but a serious fully automated system would have to tackle some unaddressed issues:

- If no street graph is available, what other a priori information about the scene can be used to aid the partitioning?
- How can impostor information be shared among adjacent view cells in order to reduce storage requirements?
- What part of the far field should be converted to impostors, and what part should be rendered using triangles? One suggestion is to set a polygon budget for each view cell and render the rest as impostors [Alia99b]. However, the size of the generated impostor should probably also enter into the calculation.
- Should view cells be rather large or small? How does the size of the chosen impostor affect the efficiency of the representation?

At the moment, a semi-automatic process depending on the street graph is viable for medium-sized urban scenes. Larger and more general scenes require an automatic process that places impostors where they are needed, optimizing rendering speed for all view cells and at the same time reducing overall storage requirements.

6.4.3 Near-field and far-field distance

The ray casting and image caching technique would profit from studying the behavior of the system with respect to scene complexity, overall “type” of the scene and to the algorithm parameters. In particular, it would be interesting to automatically determine such parameters as back plane distance, number of rays to trace

per frame and grid resolution, so as to always provide near optimal performance. Determining the border between near field and far field automatically would adapt the algorithm perfectly to different hardware systems. Buying a better graphics card would allow to increase the far-field distance, while it could be decreased if a faster CPU were used. The automatic level-of-detail management system by Funkhouser and Séquin [Funk93] provides valuable hints on how to do this.

Another interesting avenue of research for the ray casting method is the use of graphics hardware for the image-based operations that were introduced. With systems that allow access to frame buffer and texture memory with the same speed as to the system memory, it might be possible to let the hardware do the reprojection of the environment map onto the screen. For example, a simple extension to the current system would be to use graphics hardware to create the opacity buffer.

6.5 Conclusion

We have presented two algorithms capable of considerably speeding up rendering of large virtual environments. Both algorithms are in their nature image-based. Both make some assumptions about the scenes they handle. We believe that image-based techniques such as the ones presented here are the way to go for the simplification of distant geometry. We have also demonstrated that distant geometry is indeed a very good candidate for simplification. What's more, the second method even significantly improves the image quality of such geometry.

This thesis demonstrates clearly the tradeoff between online and offline techniques. Each of the two methods presented has its individual advantages, but none of the two comes without a cost. It is up to the application to decide where the focus lies: whether online or offline calculation is more important.

We have also shown that points are a valuable rendering primitive, but they cannot be used indiscriminately. Rendering points, especially for distant geometry, requires a terse mathematical treatment. Decoupling geometry and appearance has proven crucial for high-quality rendering of distant geometry, and points are well suited to this particular task.

Concerning chapter 4, it seems that ray tracing is not to be discounted as a tool in real-time rendering. Supported by other image-based rendering techniques like a panoramic image cache or the horizon map, ray casting has proven to accelerate walkthroughs of large virtual environments by an order of magnitude.

In conclusion, we observe that the last words on real-time rendering are not spoken, but we think that this thesis is a step in the right direction towards achieving this valuable goal.

Bibliography

[Adel91] Edward H. Adelson and James R. Bergen. The Plenoptic Function and the Elements of Early Vision. *Computational Models of Visual Processing*, 1991. 20

[Aila01] Timo Aila and Ville Miettinen. SurRender Umbra – Reference Manual. http://www.surrender3d.com/umbra/index.html, 2001. 15

[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. 11

[Alia97] Daniel G. Aliaga and Anselmo A. Lastra. Architectural Walkthroughs Using Portal Textures. In Roni Yagel and Hans Hagen, editors, *Proceedings of the conference on Visualization '97*, pages 355–362. IEEE, October 1997. 24, 60

[Alia99a] Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Keny Hoff, Tom Hudson, Wolfgang Stürzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manoclia. MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration. In Stephen N. Spencer, editor, *1999 Symposium on interactive 3D Graphics*, pages 199–206. ACM SIGGRAPH, ACM Press, April 1999. ISBN 1-58113-082-1. 19

[Alia99b] 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. 79, 89

[Aman87] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In G. Marechal, editor, *Eurographics '87*, pages 3–10. North-Holland, August 1987. 31, 48

[Appe68] Arthur Appel. Some Techniques for Shading Machine Renderings of Solids. In *AFIPS 1968 Spring Joint Computer Conference*, volume 32, pages 37–45, 1968. 30

[Arvo87] James Arvo and David Kirk. Fast Ray Tracing by Ray Classification. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH 87 Proceedings)*, volume 21, pages 55–64, July 1987. 32

[Arvo89] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 201–262. Academic Press, 1989. 30

[Bish94] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless Rendering: Double Buffering Considered Harmful. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 175–176. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0. 87

[Bitt98] Jiri Bittner, Vlastimil Havran, and Pavel Slavík. Hierarchical Visibility Culling with Occlusion Trees. In Franz-Erich Wolter and Nicholas M. Patrikalakis, editors, *Proceedings of the Conference on Computer Graphics International 1998 (CGI-98)*, pages 207–219, Los Alamitos, California, June 22–26 1998. IEEE Computer Society. ISBN 0-8186-8445-3. 13

[Blin76] James F. Blinn and Martin E. Newell. Texture and Reflection in Computer Generated Images. *Communications of the ACM*, 19(10):542–547, October 1976. ISSN 0001-0782. 23

[Blin78] J. F. Blinn. Simulation of Wrinkled Surfaces. In Richard L. Phillips, editor, *Computer Graphics (SIGGRAPH 78 Proceedings)*, volume 12, pages 286–292. ACM SIGGRAPH, ACM Press, August 1978. 23

[Catm74] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. Thesis, University of Utah, December 1974. 28

[Catm75] Edwin E. Catmull. Computer Display of Curved Surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pages 11–17, May 1975. 11

[Caza95] Frédéric Cazals, George Drettakis, and Claude Puech. Filtering, Clustering and Hierarchy Construction: a New Solution for Ray-Tracing Complex Scenes. *Computer Graphics Forum (Proc. Eurographics '95)*, 14(3):371–382, August 1995. ISSN 1067-7055. 32

[Cert96] Andrew Certain, Jovan Popović, Tony DeRose, Tom Duchamp, David Salesin, and Werner Stuetzle. Interactive Multiresolution Surface Viewing. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 91–98. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. 17

[Chai00] Jin-Xiang Chai, Xin Tong, Shing-Chow Chan, and Heung-Yeung Shum. Plenoptic Sampling. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 307–318. ACM SIGGRAPH, Addison Wesley, 2000. 21, 64, 86

[Cham96] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast Rendering of Complex Environments Using a Spatial Hierarchy. In Wayne A. Davis and Richard Bartels, editors, *Proceedings of Graphics Interface '96*, pages 132–141. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1996. ISBN 0-9695338-5-3. 39

[Chan99] Chun-Fa Chang, Gary Bishop, and Anselmo Lastra. LDI Tree: A Hierarchical Representation for Image-Based Rendering. In Alyn Rockwood, editor, *SIGGRAPH 99 Conference Proceedings*, Annual Conference Series, pages 291–298. ACM SIGGRAPH, Addison Wesley, August 1999. 25, 26

[Chen93] Shenchang Eric Chen and Lance Williams. View Interpolation for Image Synthesis. In James T. Kajiya, editor, *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 279–288. ACM SIGGRAPH, Addison Wesley, August 1993. ISBN 0-201-51585-7. 22, 25

[Chen95] Shenchang Eric Chen. Quicktime VR - An Image-Based Approach to Virtual Environment Navigation. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series,

pages 29–38. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995. 22, 49

[Clar76] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10):547–554, October 1976. ISSN 0001-0782. 11, 16

[Clea88] John G. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4(2):65–83, July 1988. 33

[Cohe98a] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-Preserving Simplification. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 115–122. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8. 18

[Cohe98b] Daniel Cohen-Or, Gadi Fibich, Dan Halperin, and Eyal Zadicario. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. *Computer Graphics Forum (Proc. Eurographics '98)*, 17(3):243–254, September 1998. ISSN 1067-7055. 14

[Coor97] Satyan Coorg and Seth Teller. Real-Time Occlusion Culling for Models with Large Occluders. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 83–90. ACM SIGGRAPH, April 1997. ISBN 0-89791-884-3. 13

[Coor99] Satyan Coorg and Seth Teller. Temporally Coherent Conservative Visibility. *Computational Geometry: Theory and Applications*, 12(1-2):105–124, February 1999. 13, 72

[Corp01] NVIDIA Corporation. NVIDIA OpenGL Specifications, 2001. available at http://www.nvidia.com/Marketing/Developer/DevRel.nsf/Program 10, 25

[Dars97] Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney. Navigating Static Environments Using Image-Space Simplification and Morphing. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 25–34. ACM SIGGRAPH, ACM Press, April 1997. ISBN 0-89791-884-3. 27

[De F97] Leila De Floriani, Paola Magillo, and Enrico Puppo. Building and Traversing a Surface at Variable Resolution. In Roni Yagel and Hans Hagen, editors, *Proceedings of the conference on Visualization '97*, pages 103–110. IEEE, October 1997. 18

[Deco99] Xavier Decoret, François Sillion, Gernot Schaufler, and Julie Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum (Proc. Eurographics '99)*, 18(3):61–73, September 1999. ISSN 1067-7055. 27, 79

[Dura97] Frédo Durand, George Drettakis, and Claude Puech. The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 89–100. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7. 14

[Dura99] Fredo Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Universite Joseph Fourier, Grenoble, France, July 1999. 15

[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. 14

[Eck95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution Analysis of Arbitrary Meshes. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 173–182. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995. 17

[Ecke98] George Eckel, Renate Kempf, and Leif Wennerberg. *OpenGL Optimizer Programmer's Guide: An Open API for Large-Model Visualization*. SGI techpubs library, 1998. Document Number 007-2852-002. 10

[Ecke00] George Eckel and Ken Jones. *OpenGL Performer Programmer's Guide*. SGI techpubs library, 2000. Document Number 007-1680-060. 10

[Fuch80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by a Priori Tree Structures. In James J. Thomas, ed-

itor, *Computer Graphics (SIGGRAPH 80 Proceedings)*, volume 14, pages 124–133. ACM SIGGRAPH, ACM Press, July 1980. 11

[Fuji86] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986. 31

[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. 16, 90

[Fuss88] Donald S. Fussell and K. R. Subramanian. Fast Ray Tracing Using K-d Trees. Technical Report CS-TR-88-07, University of Texas, Austin, March 1, 1988. 32

[Garl97] Michael Garland and Paul S. Heckbert. Surface Simplification Using Quadric Error Metrics. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 209–216. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7. 18

[Garl98] Michael Garland and Paul S. Heckbert. Simplifying Surfaces with Color and Texture using Quadric Error Metrics. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of the conference on Visualization '98*, pages 263–270. IEEE, October 1998. ISBN 1-58113-106-2. 18

[Giga90] Michael Gigante. Accelerated Ray Tracing Using Non-Uniform Grids. In *Proceedings of Ausgraph '90*, pages 157–163, Melbourne, Australia, September 1990. 32

[Gigu91] Ziv Gigus, J. Canny, and R. Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542–551, 1991. 14

[Glas84] Andrew S. Glassner. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984. ISSN 0272-1716. 32

[Glas95] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, San Francisco, CA, 1995. 8, 33

[Gold87] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987. ISSN 0272-1716. 32

[Gort96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The Lumigraph. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 43–54. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. 21, 86

[Gree86] Ned Greene. Environment Mapping and Other Applications of World Projection. *IEEE Computer Graphics and Applications*, 6(11):21–29, November 1986. ISSN 0272-1716. 23

[Gree93] Ned Greene and Michael Kass. Hierarchical Z-Buffer Visibility. In James T. Kajiya, editor, *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 231–238. ACM SIGGRAPH, Addison Wesley, August 1993. ISBN 0-201-51585-7. 12

[Gros98] J. P. Grossman and William J. Dally. Point Sample Rendering. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98 (Proceedings of the Eurographics Workshop on Rendering 98)*, pages 181–192. Eurographics, Springer-Verlag Wien New York, June 1998. 28, 68

[Hain86] Eric A. Haines and Donald P. Greenberg. The Light Buffer: A Ray Tracer Shadow Testing Accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, September 1986. 32

[Hain91] Eric A. Haines and John R. Wallace. Shaft Culling for Efficient Ray-Cast Radiosity. In Pere Brunet and Frederik W. Jansen, editors, *Photorealistic Rendering in Computer Graphics (Proceedings of the Eurographics Workshop on Rendering 91)*, pages 122–138. Eurographics, Springer-Verlag Berlin Heidelberg New York, 1991. 32

[Heck89] Paul S. Heckbert. Fundamentals of Texture Mapping and Image Warping. M.Sc. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June 1989. 23

[Heid98] Wolfgang Heidrich, Jan Kautz, Philipp Slusallek, and Hans-Peter Seidel. Canned Lightsources. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98 (Proceedings of the Eurographics Workshop on Rendering 98)*, pages 293–300. Eurographics, Springer-Verlag Wien New York, June 1998. 22

[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. 7, 8, 9

[Hopp93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. In James T. Kajiya, editor, *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 19–26. ACM SIGGRAPH, Addison Wesley, August 1993. ISBN 0-201-51585-7. 17, 18

[Hopp96] Hugues Hoppe. Progressive Meshes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. 18

[Hopp97] Hugues Hoppe. View-Dependent Refinement of Progressive Meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7. 18

[Huds97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated Occlusion Culling using Shadow Frusta. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pages 1–10. ACM Press, 4–6 June 1997. ISBN 0-89791-878-9. 13

[Isak00] Aaron Isaksen, Leonard McMillan, and Steven J. Gortler. Dynamically Reparameterized Light Fields. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 297–306. ACM SIGGRAPH, Addison Wesley, 2000. 22

[Jeva89] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89*, pages 164–172, June 1989. 32

[Jone94] Michael Jones. Lessons Learned from Visual Simulation. *ACM SIGGRAPH 94 Course Notes - Designing Real-Time Graphics for Entertainment*, 23:2.1–2.34, July 1994. 8

[Kapl87] Michael R. Kaplan. The Use of Spatial Coherence in Ray Tracing. In David E. Rogers and Ray A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 173–193. Springer Verlag, 1987. 32, 33

[Kay86] Timothy L. Kay and James T. Kajiya. Ray Tracing Complex Scenes. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH 86 Proceedings)*, volume 20, pages 269–278, August 1986. 32

[Kilg97] Mark J. Kilgard. Realizing OpenGL: Two Implementations of One Architecture. In Steven Molnar and Bengt-Olaf Schneider, editors, *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 45–56, New York City, NY, August 1997. ACM SIGGRAPH / Eurographics, ACM Press. ISBN 0-89791-961-0. 55

[Klim97] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster Ray Tracing Using Adaptive Grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, January 1997. 32

[Lave94] Stéphane Laveau and Olivier Faugeras. 3-D Scene Representation as a Collection of Images and Fundamental Matrices. In *Twelfth International Conference on Pattern Recognition (ICPR'94)*, pages 689–691. IEEE Computer Society Press, October 1994. 25

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

[Levo85] Marc Levoy and Turner Whitted. The Use of Points as Display Primitives. Technical Report TR85-022, Department of Computer Science, University of North Carolina - Chapel Hill, October 1 1985. 28

[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. 21

[Lind00] Peter Lindstrom and Greg Turk. Image-Driven Simplification. *ACM Transactions on Graphics*, 19(3):204–241, July 2000. 18

[Lipp80] Andrew Lippman. Movie-Maps: An application of the optical videodisc to computer graphics. In James J. Thomas, editor, *Computer Graphics (SIGGRAPH 80 Proceedings)*, volume 14, pages 32–42. ACM SIGGRAPH, ACM Press, July 1980. 23

[Lisc98] Dani Lischinski and Ari Rappoport. Image-Based Rendering for Non-Diffuse Synthetic Scenes. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98 (Proceedings of the Eurographics Workshop on Rendering 98)*, pages 301–314. Eurographics, Springer-Verlag Wien New York, June 1998. 29, 68

[Loun97] Michael Lounsbery, Tony D. DeRose, and Joe Warren. Multiresolution Analysis for Surfaces of Arbitrary Topological Type. *ACM Transactions on Graphics*, 16(1):34–73, January 1997. ISSN 0730-0301. 17

[Lueb95] David P. Luebke and Chris Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM SIGGRAPH, ACM Press, April 1995. ISBN 0-89791-736-7. 11

[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. 17

[Maci95] Paulo W. C. Maciel and Peter Shirley. Visual Navigation of Large Environments Using Textured Clusters. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 95–102. ACM SIGGRAPH, ACM Press, April 1995. ISBN 0-89791-736-7. 23

[Mark97] William R. Mark, Leonard McMillan, and Gary Bishop. Post-Rendering 3D Warping. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 7–16. ACM SIGGRAPH, ACM Press, April 1997. ISBN 0-89791-884-3. 27

[Max96] Nelson Max. Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. In Xavier Pueyo and Peter Schröder, editors, *Rendering Techniques '96 (Proceedings of the Eurographics Workshop on Rendering 96)*, pages 165–174. Eurographics, Springer-Verlag Wien New York, June 1996. ISBN 3-211-82883-4. 25

[McMi95a] Leonard McMillan. Computing Visibility Without Depth. Technical Report TR95-047, Department of Computer Science, University of North Carolina - Chapel Hill, October 1 1995. 26

[McMi95b] Leonard McMillan and Gary Bishop. Plenoptic Modeling: An Image-Based Rendering System. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 39–46. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995. 20, 25

[McMi97] Leonard McMillan. *An Image-based Approach to Three-Dimensional Computer Graphics*. Ph.D. Thesis, University of North Carolina at Chapel Hill, 1997. also available as UNC Technical Report TR97-013. 25

[Meye98] Alexandre Meyer and Fabrice Neyret. Interactive Volumetric Textures. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98 (Proceedings of the Eurographics Workshop on Rendering 98)*, pages 157–168. Eurographics, Springer-Verlag Wien New York, June 1998. 26

[Mill98] Gavin Miller, Steven Rubin, and Dulce Ponceleon. Lazy Decompression of Surface Light Fields For Precomputed Global Illumination. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98 (Proceedings of the Eurographics Workshop on Rendering 98)*, pages 281–292. Eurographics, Springer-Verlag Wien New York, June 1998. 22

[Möll97] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1), 1997. ISSN 1086-7651. 31

[Möll99a] Tomas Möller and Eric Haines. Real-Time Rendering, 1999. Webpage for book at http://www.realtimerendering.com. 31

[Möll99b] Tomas Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters Limited, 1999. 31

[NVID00a] NVIDIA Corporation. NV_vertex_program extension specification, 2000. available at http://www.nvidia.com/Marketing/Developer/DevRel.nsf/Program 76

[NVID00b] NVIDIA Corporation. Using Texture Compression in OpenGL, 2000. available at http://www.nvidia.com/Marketing/Developer/DevRel.nsf/Whitepa 76

[Ohta87] Masataka Ohta and Mamoru Maekawa. Ray Coherence Theorem and Constant Time Ray Tracing Algorithm. In Tsiyasu L. Kunii, editor, *Computer Graphics 1987 (Proceedings of CG International '87)*, pages 303–314. Springer-Verlag, 1987. 32, 33

[Oliv00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief Texture Mapping. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 359–368. ACM SIGGRAPH, Addison Wesley, July 2000. 26

[Park99] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive Ray Tracing. In Stephen N. Spencer, editor, *1999 Symposium on interactive 3D Graphics*, pages 119–126. ACM SIGGRAPH, ACM Press, April 1999. ISBN 1-58113-082-1. 33

[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. 29, 65

[Plan90] Harry Plantinga and Charles R. Dyer. Visibility, Occlusion, and the Aspect Graph. *International Journal of Computer Vision*, 5(2):137–160, November 1990. ISSN 0920-5691. 14

[Pope00] Voicu Popescu, John Eyles, Anselmo Lastra, Joshua Steinhurst, Nick England, and Lars Nyland. The WarpEngine: An Architecture for the Post-Polygonal Age. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 433–442. ACM SIGGRAPH, Addison Wesley, July 2000. 26

[Powe01] PowerVR Technology Overview and Roadmap. PowerVR Technology Overview and Roadmap, 2001. available at `http://www.powervr.com/WhitePapers/Whitepapers.htm`. 24

[Raff97] Matthew M. Rafferty, Daniel G. Aliaga, and Anselmo A. Lastra. 3D Image Warping in Architectural Walkthroughs. Technical Report TR97-019, Department of Computer Science, University of North Carolina - Chapel Hill, September 03 1997. 24

[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. 10

[Ronf96] Remi Ronfard and Jarek Rossignac. Full-range Approximation of Triangulated Polyhedra. *Computer Graphics Forum (Proc. Eurographics '96)*, 15(3):67–76, September 1996. ISSN 0167-7055. 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. 17

[Rubi80] Steven M. Rubin and Turner Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In James J. Thomas, editor, *Computer Graphics (SIGGRAPH 80 Proceedings)*, volume 14, pages 110–116. ACM SIGGRAPH, ACM Press, July 1980. 32

[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. 30, 65

[Same89] Hanan J. Samet. *Design and Analysis of Spatial Data Structures: Quadtrees, Octrees, and other Hierarchical Methods*. Addison–Wesley, Redding, MA, 1989. 11

[Scha95a] Gernot Schaufler. Dynamically Generated Impostors. In Dieter W. Fellner, editor, *GI Workshop on Modeling, Virtual Worlds,*, pages 129–135, November 1995. 23

[Scha95b] Gernot Schaufler and Wolfgang Stürzlinger. Generating Multiple Levels of Detail for Polygonal Geometry. In M. Göbel, editor, *Virtual Environments '95 (Second Eurographics Workshop in Virtual Environments 1995)*, pages 33–41. Springer Verlag, January 1995. 17

[Scha96] Gernot Schaufler and Wolfgang Stürzlinger. A Three-Dimensional Image Cache for Virtual Reality. *Computer Graphics Forum (Proc.*

Eurographics '96), 15(3):227–235, September 1996. ISSN 0167-7055. 23

[Scha97] Gernot Schaufler. Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes. In Julie Dorsey and Philipp Slusallek, editors, *Rendering Techniques '97 (Proceedings of the Eurographics Workshop on Rendering 97)*, pages 151–162. Eurographics, Springer-Verlag Wien New York, June 1997. ISBN 3-211-83001-4. 25

[Scha98] Gernot Schaufler. Per-Object Image Warping with Layered Impostors. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98 (Proceedings of the Eurographics Workshop on Rendering 98)*, pages 145–156. Springer-Verlag Wien New York, June 1998. 26

[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. 14

[Schi00] Hartmut Schirmacher, Wolfgang Heidrich, and Hans-Peter Seidel. High-Quality Interactive Lumigraph Rendering Through Warping. In *Proceedings of Graphics Interface 2000*, pages 87–94. Morgan Kaufman, May 2000. 22

[Schr92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH 92 Proceedings)*, volume 26, pages 65–70. ACM SIGGRAPH, ACM Press, July 1992. ISSN 0097-8930. 17

[Sega92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast Shadows and Lighting Effects Using Texture Mapping. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH 92 Proceedings)*, volume 26, pages 249–252. ACM SIGGRAPH, ACM Press, July 1992. ISSN 0097-8930. 23

[Seit96] Steven M. Seitz and Charles R. Dyer. View Morphing: Synthesizing 3D Metamorphoses Using Image Transforms. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 21–30. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. 23

[Shad96] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 75–82. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. 23

[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. 25

[Sill97] François Sillion, G. Drettakis, and B. Bodelet. Efficient Impostor Manipulationfor Real-Time Visualization of Urban Scenery. *Computer Graphics Forum (Proc. Eurographics '97)*, 16(3):207–218, August 1997. ISSN 1067-7055. 27, 79, 87, 89

[Simm00] Maryann Simmons and Carlo H. Séquin. Tapestry: A Dynamic Mesh-based Display Representation for Interactive Rendering. In Bernard Péroche and Holly Rushmeier, editors, *Rendering Techniques 2000 (Proceedings of the Eurographics Workshop on Rendering 2000)*, pages 329–340. Eurographics, Springer-Verlag Wien New York, June 2000. ISBN 3-211-83535-0. 24

[Sloa97] Peter-Pike Sloan, Michael F. Cohen, and Steven J. Gortler. Time Critical Lumigraph Rendering. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 17–24. ACM SIGGRAPH, ACM Press, April 1997. ISBN 0-89791-884-3. 22

[Snyd87] John M. Snyder and Alan H. Barr. Ray Tracing Complex Models Containing Surface Tessellations. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH 87 Proceedings)*, volume 21, pages 119–128, July 1987. 32

[Subr91] K. R. Subramanian and Donald S. Fussell. Automatic Termination Criteria for Ray Tracing Hierarchies. In *Proceedings of Graphics Interface '91*, pages 93–100, June 1991. 32

[Suda96] Oded Sudarsky and Craig Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality.

Computer Graphics Forum (Proc. Eurographics '96), 15(3):249–258, August 1996. ISSN 0167-7055. 44

[Sung91] Kelvin Sung. A DDA Octree Traversal Algorithm for Ray Tracing. In Werner Purgathofer, editor, *Eurographics '91*, pages 73–85. North-Holland, September 1991. 32

[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. 11

[Tell92] Seth J. Teller. Computing the antipenumbra of an area light source. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH 92 Proceedings)*, volume 26, pages 139–148. ACM SIGGRAPH, ACM Press, July 1992. ISSN 0097-8930. 14

[Torb96] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 353–364. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. 24

[Turk92] Greg Turk. Re-tiling polygonal surfaces. In *Computer Graphics (SIGGRAPH 92 Proceedings)*, volume 26, pages 55–64. ACM SIGGRAPH, ACM Press, July 1992. ISSN 0097-8930. 17

[Walt99] Bruce Walter, George Drettakis, and Steven Parker. Interactive Rendering using Render Cache. In Dani Lischinski and Greg Ward Larson, editors, *Rendering Techniques '99 (Proceedings of the Eurographics Workshop on Rendering 99)*, pages 19–30. Eurographics, Springer-Verlag Wien New York, June 1999. ISBN 3-211-83382-X. 24

[Ward99] Gregory Ward and Maryann Simmons. The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments. *ACM Transactions on Graphics*, 18(4):361–398, October 1999. ISSN 0730-0301. 24

[Wegh84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984. ISSN 0730-0301. 32

[Whit80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980. 30

[Will83] Lance Williams. Pyramidal Parametrics. In *Computer Graphics (SIGGRAPH 83 Proceedings)*, volume 17, pages 1–11. ACM SIGGRAPH, ACM Press, July 1983. 23

[Wimm99a] 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. Springer-Verlag Wien, June 1999. 5

[Wimm99b] 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. 5

[Wimm01] Michael Wimmer, Peter Wonka, and François Sillion. Point-Based Impostors for Real-Time Visualization. Eurographics, Springer-Verlag Wien New York, June 2001. 5

[Wolb90] George Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 10662 Los Vaqueros Circle, Los Alamitos, CA, 1990. IEEE Computer Society Press Monograph. 23

[Wonk99] Peter Wonka and Dieter Schmalstieg. Occluder Shadows for Fast Walkthroughs of Urban Environments. *Computer Graphics Forum (Proc. Eurographics '99)*, 18(3):51–60, September 1999. ISSN 1067-7055. 12

[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 of the Eurographics Workshop on Rendering 2000)*, pages 71–82. Eurographics, Springer-Verlag Wien New York, June 2000. ISBN 3-211-83535-0. 6, 14, 79

[Wonk01] Peter Wonka, Michael Wimmer, and François Sillion. Instant Visibility. *Computer Graphics Forum (Proc. Eurographics 2001)*, 20(3), September 2001. 6, 15

[Wood00] Daniel N. Wood, Daniel I. Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H. Salesin, and Werner Stuetzle. Surface Light

Fields for 3D Photography. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 287–296. ACM SIGGRAPH, Addison Wesley, 2000. 22, 64

[Xia96] Julie C. Xia and Amitabh Varshney. Dynamic View-Dependent Simplification for Polygonal Models. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of the conference on Visualization '96*, pages 327–334. IEEE, October 1996. ISBN 0-7803-3673-9. 18

[Zhan97] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility Culling Using Hierarchical Occlusion Maps. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 77–88. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7. 12

Curriculum vitae

Name:	Dipl.-Ing. Michael Wimmer
Date of birth:	December 23, 1973 in Vienna, Austria
Nationality:	Austria
Email:	wimmer@cg.tuwien.ac.at
Languages:	German, English, French, Spanish

Education

Sep. 1980–June 1984:	Primary school (Volksschule) in Vienna.
Sep. 1984–June 1992:	Secondary school (AHS) in Vienna, BGXVIII.
June 6, 1992:	Graduation (Matura) with distinction.
Oct. 1992–Jan. 1997:	Studies of computer science (Informatik) at the Vienna University of Technology.
Oct. 1992–Jan. 1994:	Studies of economics (Handelswissenschaften) at the Vienna University of Economics and Business Administration.
Jan. 30, 1997:	Graduation (with distinction) to "Diplom-Ingenieur der Informatik" (computer science). Diploma thesis: "Interactive Techniques in Three-dimensional Modeling" at the Institute of Computer Graphics.
Since March 1997:	Doctoral program (Ph.D.) in computer science at the Vienna University of Technology. Member of the scientific staff of the Institute of Computer Graphics.
Since Oct. 1998:	Studies of Technical Mathematics at the Vienna University of Technology (completed first part in June 2000).

Jobs

Oct. 1993–June 1997:	Working part-time for Vobis Austria.
1992–1997:	Various technical consulting activities for IBM Austria.
Summer 1993:	Implementation of a student management system for the ÖAD.
1995:	2 database projects for 2 different departments at IBM Austria.
Mar. 1997–June 2000:	Research assistant at the Institute of Computer Graphics.
July 2000–Feb. 2001:	EU Research position (TMR project "PAVR") at iMAGIS, Grenoble, France.
Since March 2001:	Universitätsassistent (assistant professor) at the Institute of Computer Graphics, Vienna University of Technology.