Accelerating the Rendering Process Using Impostors

Dissertation

zur

Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik

der Universität Rostock

vorgelegt von

Stefan Jeschke, geb. am 7. Mai 1977 in Rostock

aus Dändorf

Rostock, Dezember 2004

Gutachter:
Prof. Dr.-Ing. habil. Heidrun Schumann, Universität Rostock
Prof. Dr. techn. habil. Werner Purgathofer, Technische Universität Wien
Prof. Francois Sillion, INRIA Rhône-Alpes

Datum der Verteidigung: 31.03.2005

Abstract

The interactive rendering of three-dimensional geometric models is a research area of big interest in computer graphics. The generation of a fluent animation for complex models, consisting of multiple million primitives, with more than 60 frames per second is a special challenge. Possible applications include ship-, driving- and flight simulators, virtual reality and computer games. Although the performance of common computer graphics hardware has dramatically increased in recent years, the demand for more realism and complexity in common scenes is growing even faster.

This dissertation is about one approach for accelerating the rendering of such complex scenes. We take advantage of the fact that the appearance of distant scene parts hardly changes for several successive output images. Those scene parts are replaced by precomputed image-based representations, so-called *impostors*. Impostors are very fast to render while maintaining the appearance of the scene part as long as the viewer moves within a bounded viewing region, a so-called *view cell*.

However, unsolved problems of impostors are the support of a satisfying visual quality with reasonable computational effort for the impostor generation, as well as very high memory requirements for impostors for common scenes. Until today, these problems are the main reason why impostors are hardly used for rendering acceleration.

This thesis presents two new impostor techniques that are based on partitioning the scene part to be represented into image layers with different distances to the observer. A new error metric allows a guarantee for a minimum visual quality of an impostor even for large view cells. Furthermore, invisible scene parts are efficiently excluded from the representation without requiring any knowledge about the scene structure, which provides a more compact representation. One of the techniques combines every image layer separately with geometric information. This allows a fast generation of memory-efficient impostors for distant scene parts. In the other technique, the geometry is independent from the depth layers, which allows a compact representation for near scene parts.

The second part of this work is about the efficient usage of impostors for a given scene. The goal is to guarantee a minimum frame rate for every view within the scene while at the same time minimizing the memory requirements for all impostors. The presented algorithm automatically selects impostors and view cells so that for every view, only the most suitable scene parts are represented as impostors. Previous approaches generated numerous similar impostors for neighboring view cells, thus wasting memory. The new algorithm overcomes this problem.

The simultaneous use of additional acceleration techniques further reduces the required impostor memory and allows making best use of all available techniques at the same time. The approach is general in the sense that it can handle arbitrary scenes and a broad range of impostor techniques, and the acceleration provided by the impostors can be adapted to the bottlenecks of different rendering systems.

In summary, the provided techniques and algorithms dramatically reduce the required impostor memory and simultaneously guarantee a minimum output image quality. This makes impostors useful for numerous scenes and applications where they could hardly be used before.

Kurzfassung

Die Darstellung dreidimensionaler geometrischer Modelle zur Erzeugung glaubwürdiger Bilder ist in der Computergrafik ein Gebiet von großem Interesse. Eine besondere Herausforderung ist hierbei die Erstellung einer flüssigen Animation mit mindestens 60 Bildern pro Sekunde für sehr komplexe Modelle. Anwendungen hierfür finden sich in vielfältigen Bereichen wie zum Beispiel in Schiffs-, Fahr-, oder Flugsimulationen, virtuellen Realitäten oder auch Computerspielen. Obwohl gebräuchliche Grafikhardware in den vergangenen Jahren an Leistung stark zugenommen hat, wachsen die Ansprüche nach realistischeren und damit komplexeren Modellen in noch höherem Maße.

Diese Dissertation beschäftigt sich mit einem Ansatz zur beschleunigten Ausgabe solch komplexer Modelle. Es wird ausgenutzt, daß sich das Erscheinungsbild insbesondere entfernter Szenenteile über mehrere Ausgabebilder kaum verändert. Diese Szenenteile werden durch vorberechnete bildbasierte Repräsentationen, sogenannte *Imposter*, ersetzt. Imposter bieten den Vorteil der schnelleren Darstellbarkeit bei gleichem oder zumindest ähnlichem Erscheinungsbild für einen räumlich abgegrenzten Bereich, dem sogenannten *Sichtbereich*. In bisherigen Ansätzen hierzu wurde jedoch die visuelle Qualität der Impostor (d.h. die visuelle Unterscheidbarkeit zur Originalgeometrie) für den Sichtbereich nur unter sehr hohem Aufwand sichergestellt, und die Speicherplatzanforderungen für alle Imposter einer Szene sind oft unerwünscht hoch. Diese Punkte sind beim heutigen Stand der Impostertechnik als Hauptprobleme einer breiten Anwendbarkeit zu sehen.

In dieser Arbeit wurden zwei neue Impostortechniken entwickelt, die auf einer Einteilung des zu repräsentierenden Szenenteils in Bildschichten mit unterschiedlichem Abstand zum Betrachter basieren. Durch die Einführung spezieller Fehlermetriken wird die visuelle Qualität der Imposter für einen großen Sichtbereich quantifizierbar und garantierbar. Gleichzeitig können unsichtbare Szenenteile effizient entfernt werden, was den Speicherbedarf für die Repräsentation verringert. Dabei werden keinerlei Informationen über die Struktur des originalen Szenenteils benötigt. Bei der einen Technik wird jede Bildschicht separat mit Geometrieinformation verknüpft. Hierdurch wird eine schnelle Impostererstellung sowie ein sehr geringer Speicherbedarf für entfernte Szenenteile erreicht. Bei der anderen Technik erfolgt die Verknüpfung der Geometrieinformation unabhängig von den Bildschichten. Dies reduziert die geometrische Komplexität und den benötigten Speicherplatz für die Repräsentation nahe gelegener Objekte wesentlich.

Der zweite Teil der Arbeit beschäftigt sich mit dem effizienten Einsatz von Impostern. Das Ziel ist hierbei, durch den Impostereinsatz eine Mindestbildwiederholrate für jeden möglichen Blickpunkt in einer Szene zu garantieren und gleich-

zeitig den Speicherplatzbedarf für alle Imposter zu minimieren. Dazu wurde ein Algorithmus entwickelt, der automatisch Imposter und dazugehörige Sichtbereiche so auswählt, daß nur solche Szenenteile als Imposter für jeden Blickpunkt repräsentiert werden, die sich hierfür besonders eignen. Speziell wird dabei der Fehler bisheriger Ansätze vermieden, für benachbarte Sichtbereiche mehrere sehr ähnliche Imposter für entfernte Objekte zu generieren. Außerdem werden parallel zu Impostern weitere Darstellungsbeschleunigungsverfahren eingesetzt, was den Speicheraufwand für die Imposter weiter reduziert. Der Algorithmus ist dabei so allgemein gehalten, daß ein effizienter Impostereinsatz in beliebigen dreidimensionalen Szenen sowie auch mit unterschiedlichen Impostertechniken ermöglicht wird.

Zusammenfassend erlauben die entwickelten Techniken and Algorithmen eine flüssige Animation bei einer garantierbaren Mindestausgabebildqualität sowie einen wesentlich geringeren Speicheraufwand für Imposter in einer Szene. Dies erlaubt den Einsatz von Impostern in verschiedenen Szenen und Applikationen, in denen diese Technik bisher nicht anwendbar war.

Acknowledgements

This work was financially supported by the German Research Foundation (DFG) in the frame of the postgraduate program *"Graduiertenkolleg 466: processing, administrating, visualization and transfer of multimedia data–technical basics and social implications"* at the University of Rostock.

First of all, I would like to express my gratitude to my supervisor Heidrun Schumann, who offered me a position in the postgraduate program, always helped me with problems I encountered during my PhD studies and allowed me room for development. Great thanks go to Werner Purgathofer for many useful hints and comments that influenced my scientific development, and for reviewing this thesis. The same applies to François X. Sillion, who was willing to review this work in spite of the short notice.

Special thanks go to Michael Wimmer for countless fruitful discussions, his great help for publishing the results of this work and excellent proof reading of this dissertation. I also thank Peter Wonka and Jiří Bittner for interesting discussions, Gerke Preussner for implementing a helpful optimization framework, and Konrad Engel for interesting discussions and insights on optimization problems.

I would also like to thank all people at the Institute of Computer Graphics at the University of Rostock, and the people at the RVR Group at the Institute of Computer Graphics and Algorithms at the Vienna University of Technology. I profited greatly from the creative atmosphere and enthusiasm these two research groups offered to me.

Finally, I would like to thank Mariya who gave me much strength for my work, and my parents for always supporting me in everything I did.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In computer graphics, the interactive visualization of three-dimensional models is a research area of big interest. The basic concept is that a user interactively explores a 3D model, thus taking advantage of immersion, which is not possible for still images. Applications include computer games, flight and driving simulations, virtual reality scenarios, architectural visualization and computer aided design. The 3D models are indoor (for instance, architectural scenes) or outdoor (for instance, urban scenes) environments, or single objects (for instance, molecular structures). The user navigation consists of either object exploration, on-ground navigation (so-called *walkthrough*) or flying through a scene (so-called *flythrough*).

Nowadays, visualization uses specialized graphics hardware that comes with practically every off-the-shelf personal computer and games console. The target output resolution lies between 640x480 pixel resolution for NTSC video games and about 1280x1024 pixel displays of today's personal computers. During the interaction with the model, a smooth animation is desired in order to allow a convincing immersion. We understand output frame rates of at least 10 frames per second as *interactive* and of at least the refresh rate of the output display as *real-time*. Real-time frame rates range from 50 Hz for TV display over 60 Hz for a thin-film transistor display (TFT) up to more than 85 Hz for a cathod ray tube (CRT) monitor. While interactive frame rates are necessary to give the user a rough feeling for scene exploration, real-time frame rates provide much better immersion into the virtual scene. This way, Temporal aliasing effects noticeable as "jerky" animations are avoided.

3D models consist of graphical primitives including points, lines, polygons

or splines. These primitives are typically organized in a data structure, called a *scene graph*. One main challenge for real-time rendering systems is to provide appropriate frame rates even for very complex 3D models. The performance of consumer-level hardware has increased dramatically in recent years, allowing the rendering of several Million graphical primitives per second. Furthermore, graphics hardware has become more programmable, which allows better scene realism. On the other hand, due to the continuing desire for more detail and realism, the model complexity for common scenes has not reached its peak by far. Because of this fact, scenes are often too complex to be displayed at interactive or even real-time frame rates. The rendering acceleration of such scenes has been a hot topic in computer graphics in recent years and it seems like this is not going to change in the near future.

The main problem is that the output frame rate for a rendering system is bound to the scene complexity. Consequently, the only way to obtain high frame rates is to reduce this complexity for every output image. Many relevant algorithms have been proposed, following one of four main strategies:

- The complexity of today's graphics drivers and hardware behavior leaves much space for optimization in many cases. Removing so-called *bottlenecks* in the rendering process might dramatically increase the performance.
- Visibility calculations remove invisible portions of a scene before they are sent to graphics hardware. While this provides dramatic rendering acceleration for some cases, the actually visible geometry may already overwhelm the hardware.
- Geometric simplification techniques take advantage of the fact that distant scene parts are highly complex but contribute only little to the output image. Consequently, coarser geometric representations are used with increasing distance without loss of image quality. Unfortunately, these techniques are not applicable for arbitrary scene parts. For instance, if multiple objects are merged during the simplification process, preserving the appearance (for instance, textural information) is not possible in an efficient way.
- If geometric simplification techniques cannot be applied, image-based representations (so-called *impostors*) can be used instead.

The term "impostor" has been used in literature in different ways. It is defined and characterized for this thesis in the following section.

1.2 Impostor Definition

In 1995, Maciel and Shirley [Maci95] defined impostors as entities that are faster to draw than the *true* object, but retaining the same visual characteristics. In this definition, geometry-based levels of detail (see Section 2.2.3) are a special kind of impostors. In subsequent literature, impostors were usually referenced as *image-based* representations in contrast to geometry-based simplifications. The term image based means that the representation is generated from appearance samples, typically acquired by rendering the scene part. Today, the name *impostor* is loosely used in the computer graphics community, mostly for referring to image-based scene descriptions. Because this thesis deals with the acceleration of rendering, impostors are defined here as replacements of geometry-based scene parts. This means, the geometry-based representation is treated as the *true* object in Maciel's definition described above. The following informal definition of impostors is used throughout this thesis:

> *Impostors are image-based entities used as alternative representations for 3D scene parts for accelerating their rendering process.*

Note that the definition excludes common *billboarding* techniques often used in computer games for representing fire, smoke, trees and grass, or static background images that show hills or city skylines. In these cases, the image-based representation is already considered as the original (*true*) object, rather than a replacement we concentrate on.

1.3 Impostor Basics

An impostor can either be generated in a preprocess (a so-called *static impostor*) or dynamically at runtime (a so-called *dynamic impostor*). During the model visualization, the impostor is displayed instead of the original scene part. While this provides the same visual output, the impostor can be rendered much faster. The rendering time of an impostor basically depends on the size of the impostor on screen rather than on the geometric complexity of the original scene part. This allows a very fast display of objects with arbitrary geometric complexity.

In order to generate an impostor, the respective appearance as well as the geometric structure is acquired. This is typically done from a single viewpoint, called the *reference viewpoint*. The acquired information is combined into an impostor representation. There exist a number of different *impostor techniques*, mainly characterized by the type and amount of geometric information associated with

an impostor. In the simplest case, the scene part is rendered into a texture (the *impostor texture*) and combined with a quadrilateral (the *impostor geometry*) for placing it into the scene.

The main advantage of impostors compared to geometry-based simplification techniques is that the generation process does not rely on any knowledge about the geometric structure of the original scene part. Because of this, they can be used for arbitrary scenes.

For a convincing representation, the original scene part should be represented as correct as possible. In general, the region a particular impostor can be displayed for is bounded. Figure 1.1 shows examples for image artifacts that can occur

Figure 1.1: Image artifacts caused by impostors. Top-left: original image rendered without impostors. Top-right: blocky impostor texels. Bottom-left: image gap (the impostor was generated to the right of the actual viewpoint). Bottom-right: parallax errors result in wrong perspective and scene integration (the right row of houses should be invisible).

during impostor display. The following aspects have to be considered in order to avoid such artifacts:

- The impostor texture resolution should not fall below the output image resolution. Otherwise, “blocky” pixels become visible and the illusion of displaying a true object is destroyed. This means that a minimum distance to the impostor must be maintained. Furthermore, the acquisition of the appearance should provide a similar result compared to the originally rendered object.

- If the viewpoint is changed, parallax effects (kinetic depth effects) occur as nearby scene parts seem to move compared to distant scene parts. This effect can be supported in an impostor by applying appropriate impostor geometry, depending on a particular impostor technique. Because visible parallax effects grow with increasing distance between the reference viewpoint and the position of the viewer, this effect further limits the viewing region an impostor can be displayed for with sufficient accuracy.

- Parallax movements cause new scene parts to appear. This effect is called *disocclusion*. Of course, it is highly desirable that an impostor includes *all* scene parts that may become visible. Otherwise, so-called *image gaps* or *rubber-sheet effects* occur. The avoidance of these artifacts is not simple and current solutions to this problem are computationally very costly. It is furthermore desirable to exclude scene parts that never can become visible in order to keep the amount of memory needed for the impostor as low as possible.

- For a seamless integration of an impostor into a scene, the border between impostor and adjacent geometry should not show any artifacts. This is especially important if objects are partially represented by impostors and partially by the original geometry in the same output image. Furthermore, for scenes containing non-static content such as human characters, cars etc., the visibility between this dynamic content and the impostor must be solved. Several related application-dependent approaches have been presented in recent literature. For instance, Harris and Lastra [Harr01] show an example for clouds represented as impostors and an aeroplane flying through the clouds.

If the difference between an impostor and the original rendered object is not noticeable, the impostor is called *valid*. The viewing region an impostor is valid for is called *view cell*. The size and shape of a view cell depends on how a particular impostor technique implements the image quality issues described above. Two view-cell shapes that have been used for different impostor techniques are considered throughout this thesis. *Box-shaped* view cells have been widely used in previous work (see Chapter 2). *Shaft-shaped* view cells have often been used implicitly [Jaku00, Aube99, Scha96b, Shad96]. Figure 1.2 shows an example for a shaft in 2D. The shaft apex lies in the center of the represented scene part. It is defined by a direction, an apex angle and a minimum distance to the object (see figure 1.2). Shafts address the fact that errors introduced by the impostor are much less apparent when increasing the view distance. Consequently, the impostor can be displayed for a much larger view space region compared to box-shaped view cells.

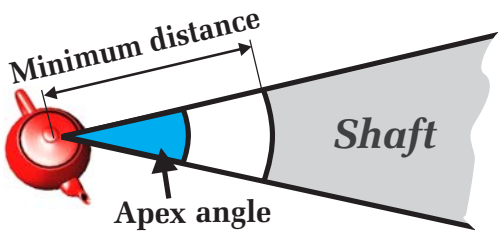

Figure 1.2: A shaft-shaped view cell.

If an impostor is no longer valid, another representation (impostor or original object) is displayed. This results in more or less noticeable *popping artifacts*, depending on how well the impostor resembles the original scene parts at the moment the representation is switched.

1.4 Problems and Challenges

Several rendering systems using impostors for research purposes have been presented. But until today, they were hardly used in commercial applications. This is caused by a number of problems and challenges introduced by adopting impostors into a rendering system. These problems are described in this section.

1.4.1 Image Quality vs. Efficiency

A consequence of the limited impostor validity described in Section 1.3 is that for typical applications, numerous impostors are required to provide sufficient rendering acceleration for every view in a scene. Depending on whether static or dynamic impostors are used, this results either in very high memory requirements or frequent impostor updates during runtime.

High Memory Requirements for Static Impostors

An impostor representation requires a relatively large amount of memory. This is because a large amount of appearance samples are typically stored. In contrast, the impostor geometry usually requires only little memory. Numerous impostors for a whole scene let the required memory grow of an enormous pace. In typical cases, the impostors do not fit into graphics memory nor into main memory. Recent research [Alia99c, Deco99] reported on several gigabytes even for mid-range

scenes, which would not even fit on a DVD. Furthermore, the impostors have to be successively loaded from harddisk into main memory and into graphics memory. This operation must be finished for every impostor before it is needed for display. So-called *prefetching strategies* try to load the most probably needed impostors in advance, often using predictions about future user movement. However, in case of sudden viewpoint changes, the limited memory bandwidth might lead to missing impostors at the display time. Restricting the user velocity is one solution to this problem, but this is not always tolerable. Furthermore, the widely varying memory bandwidths of different hardware limit the portability of such a rendering system. These facts reduce the practical usability of impostors significantly.

Another related problem is the time needed to generate an impostor. More involved impostor techniques have been presented that provide a larger region the impostors are valid for. While this reduces the required memory for a whole scene, such techniques in turn need very long preprocessing times that might be unacceptable for some applications. In recent work [Alia99c, Wils03], several hours of preprocessing even for moderately sized scenes are reported. This makes impostors not useful for applications where they could be generated on the fly, for instance, when loading a level of a computer game.

Frequent Dynamic Impostor Updates

For dynamically generated impostors, the rendering resources are shared for model visualization and impostor generation. An important point is that the rendering system must not be overwhelmed by the impostor generation process. Therefore, impostor techniques with short generation times are desired. However, because such simple impostor techniques are not valid very long, this in turn causes numerous impostor updates, which decreases the efficiency of this approach. In case of sudden viewpoint changes, too many impostors might have to be updated, resulting in a frame rate drop. Restricting the user velocity in order to avoid such cases is not always an option. This fact makes the efficiency of dynamic impostor techniques highly sensitive to the target application and to the performance of the target rendering system.

The issues above lead to similar demands for static and dynamic impostors: a large view region every impostor is valid for, fast generation, and (especially for static impostors) low memory requirements. Furthermore, a high image quality without artifacts is desirable. Avoiding gaps due to disocclusions is a special challenge in this connection. Today, there is still no technique that provides all these features at the same time. Instead, a tradeoff must be found between the

contrary demands. Previous approaches that use impostors in order to accelerate the rendering of typical scenes often accept a loss of image quality to keep the immense amount of memory to a tolerable (but still not desirable) level. This typically results in visible image gaps and popping artifacts.

In summary, an impostor technique that provides a large view region without showing artifacts, that is fast to generate and requires only little memory seems to be one key issue for a more efficient impostor use.

1.4.2 Efficient Impostor Usage

A rendering system dealing with impostors has to decide for every output view which scene parts to display with impostors and where to rely on a geometric representation. We call this task *impostor placement*. The goal of an impostor placement is to make optimal use of the impostors. This means, a reasonable image quality with only little memory requirements should be provided while at the same time achieving a high rendering acceleration for every output view.

Placing impostors manually is hardly an option, even for moderately sized scenes. Instead, previous approaches used impostors either for complex objects like for instance trees [Jaku00], or the scene was partitioned into a set of view cells, each associated with a separate set of impostors. Practically all approaches address the fact that distant scene parts are amenable for being replaced by impostors because a lot of geometry covers only few pixels in the output image. This favors a high rendering acceleration with low memory requirements and large view cells with only few noticeable image artifacts. Furthermore, a priori knowledge about visibility was successfully used for architectural scenes [Raff98b] and urban environments [Sill97] to further lower the memory needed for impostors.

However, using impostors per object might lead to situations where too many impostors have to be displayed so that the rendering system is still overloaded. While a placement per view cell avoids such situations, this strategy leads to many similar impostors for representing distant scene parts for adjacent view cells, which constitutes a waste of memory. Furthermore, impostors should not be placed indiscriminately but only were needed for rendering acceleration in order to reduce memory requirements. These issues have not been addressed in a satisfying way in recent work and there still exists no general impostor placement strategy that is useful for arbitrary scenes and addresses all aspects of the impostor placement problem.

1.5 Contributions

This dissertation contributes to the field of computer graphics by addressing the challenges stated in Section 1.4. New techniques and algorithms are presented for making impostors useful for a wide range of scenes and applications.

1.5.1 Efficient Impostor Techniques

In Chapter 3 we present a new *layered impostor technique* that combines the following desirable features:

- It provides a guarantee that all scene parts visible from a particular view cell are represented with adequate resolution. This eliminates artifacts like image gaps.
- Invisible scene parts are efficiently excluded with practically no additional computational effort and without relying on any knowledge about the geometric model structure. This further reduces the required impostor memory.
- The impostor generation is very fast.
- The memory requirements and geometric complexity of the impostors are fairly low for distant scene parts. The technique is also efficient for large view cells.
- Since the impostors only consist of few textured polygons, it naturally supports graphics hardware for very fast impostor display.

In Chapter 3 the individual steps of the layered impostor generation algorithm are explained and respective results are discussed. Parts of this work have been presented at the Graphics Interface 2002 [Jesc02b]. In Chapter 5 this impostor technique will be used for accelerating the rendering of a larger scene. This will demonstrate the potential of the technique for an application that is of practical interest.

Chapter 4 presents a *textured depth mesh* impostor technique that allows efficiently representing scene parts near the view cell. Features like the elimination of image artifacts, low memory requirements and geometric complexity as well as fast display using graphics hardware are equally supported by this technique. The cost for the compact representation is mainly the longer impostor generation time. This work has been presented at the Eurographics Workshop on Rendering 2002 [Jesc02a].

We believe that the combination of the desirable features mentioned above is beneficial for a wider use of impostors. Especially the elimination of image artifacts for a large viewing region with only very little computational effort is an important step to make impostors beneficial for many applications.

1.5.2 Automatic Impostor Placement

Chapter 5 introduces a new impostor placement approach that automatically decides for every output view which scene parts should be represented as impostors and which are rendered using the original representation. The aim is to guarantee a minimum frame rate and image quality, while the amount of memory required for all impostors is minimized. The contribution of the algorithm is that it addresses all aspects of the impostor placement problem so that it provides a reasonable impostor placement in practically all scenes. The individual aspects are:

- Impostors are only generated for views where they are actually needed for sufficiently fast rendering. This is done in a general way so that the algorithm is not tied a particular rendering system, type of scene or impostor technique.
- It breaks the rigid correlation between impostors and view cells as was applied by all previous approaches. This avoids the generation of similar impostors for distant scene parts for adjacent view cells, which greatly reduces the required impostor memory.
- It seamlessly integrates into common rendering systems: additional techniques like visibility culling and geometric simplification are simultaneously used in a way so that best use is made of all available approaches.

The fact that all of these issues are addressed leads to better placements than achieved before. The results will show that in combination with the new layered impostor technique, the impostors needed for mid-range scenes can completely fit into graphics hardware memory. Because prefetching tasks are then no further an issue, impostors become useful for a number of new applications, for instance, computer games. This work was submitted to the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2005.

1.6 Models used in this Thesis

The “Vienna” model (Figure 1.3) is publicly available at the Vienna University of Technology (`www.cg.tuwien.ac.at/research/vr/urbanmodels/`).

We enhanced the model with numerous street objects in order to obtain a higher

Figure 1.3: The Vienna model, consisting of 10.4 Million textured polygons.

scene realism, so that the final model consists of 5287 objects with 10.4 Million polygons in total. Although visibility culling was available [Wonk00] and geometric simplification techniques were used for some objects like trees, the model cannot be rendered in real time with current graphics hardware. Note that geometric simplification techniques are not efficiently applicable for this model, because it consists of numerous individual textured objects. In contrast, impostors can be efficiently applied for accelerating the rendering of this model, as will be shown in Chapter 5.

The second model is the "UNC Power plant" (Figure 1.4), which is freely available at the University of North Carolina at Chapel Hill (`www.cs.unc.edu/~geom/Powerplant/`). For this model, neither visibility culling nor geometric simplification techniques were used. The original model consists of 12.7 Million polygons that are arranged in 14 individual large sections. Because of this, view-frustum culling could not be applied efficiently. Therefore, we split the model into several parts using an octree. Note that because no scene parts are instantiated, the model already requires approximately 700 MB of main

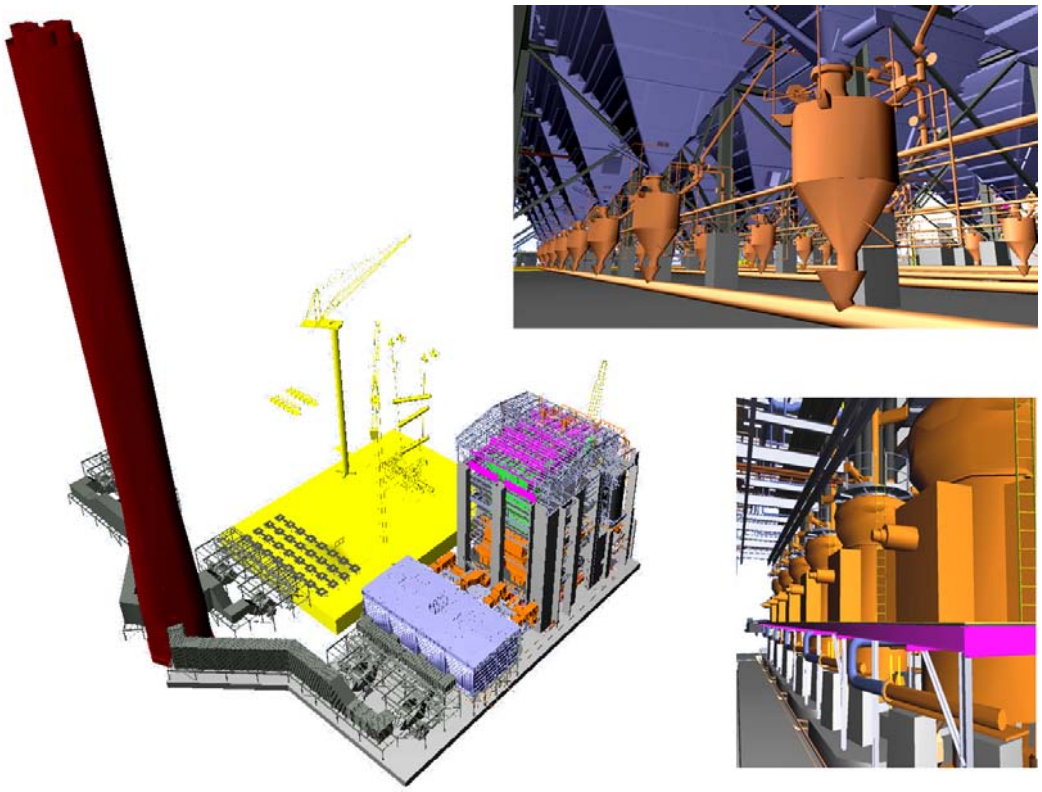

Figure 1.4: The UNC Power Plant model, consisting of 12.7 Million colored polygons.

memory, which makes the application of acceleration techniques hard without causing memory swapping.

Chapter 2

Related Work

Impostors consolidate two major research areas in computer graphics: image-based representations and real-time rendering acceleration. The following sections provide an overview of these two topics. Afterwards, a more in-depth analysis of previous work on impostors is given.

2.1 Image-Based Scene Descriptions

The basic principle of image-based modeling and rendering is to describe a scene with a collection of images instead of polygonal geometry. Here, the term *image* means a set of appearance samples (mainly color values), typically arranged in a regular grid. A general advantage of images over geometry-based representations is that their complexity hardly depends on the geometric complexity of the represented scene.

The first image-based appearance description in the form of textures applied to object surfaces was proposed by Blinn and Newell in 1976 [Blin76]. The time it takes to render a texture depends on the number of pixels the texture covers on the screen and not on the complexity of the represented detail. This feature allows fast rendering of numerous small details, which would be hard to do using a purely polygonal description. Furthermore, minification and magnification filtering for high-quality image reconstruction of pictorial information is well established and a standard feature in current rendering hardware. High-quality filtering for geometry-based representations in order to avoid staircase artifacts is nowadays done with full-screen oversampling in graphics hardware at the cost of increased fill rate requirements.

Describing a complete scene with images can be intuitively understood using the *plenoptic function* [Adel91]. This function describes the radiant energy received at a viewpoint x in view direction Θ and for wavelength Λ at time t:

$$\mu = Plenoptic(x, \Theta, \Lambda, t)$$

The output μ of this function can be any photometric unit useful for describing the color intensity values of images. Storing the complete plenoptic function for a nontrivial scene is hardly possible. Instead, images can be interpreted as subsets of the plenoptic function. Deriving new views from existing ones is then a function reconstruction problem. McMillan and Bishop [McMi95] were the first to realize the importance of the plenoptic function for image-based rendering in computer graphics.

Images from real-world objects can easily be acquired by taking photographs. As an early attempt of using images for 3D scene exploration, Lippman [Lipp80] presented the "Aspen Movie Map System" in 1980. It consisted of static panoramic images in an interactive slide show presenting the city of Aspen. The view position could be changed at fixed intervals and in order to change the view direction, the excerpt from a panoramic image was shifted. At that time this technique was the only way to provide an interactive walkthrough of a city. Videodisc technology was used to store the huge amount of data.

In the Quicktime VR System developed by Chen [Chen95], cylindrical panoramic images are obtained from a fixed viewpoint. They are stored on harddisk and reprojected for display. User movement is restricted to rotation around the viewpoint, zooming, and discrete changes of the viewpoint. Despite the quite limited functionality, this principle provides a certain degree of immersion and is very popular because modern digital cameras provide a fairly low effort for image acquisition.

Chen and Williams [Chen93] presented an approach that interpolates between samples of different reference viewpoints in order to generate an output image. This technique provides reasonable image quality for viewpoints along the line between two reference images. McMillan and Bishop [McMi95] describe a system called *plenoptic modeling* for acquiring cylindrical images and synthesizing new views from them. By taking into account the relative camera positions, they allow exact view reconstruction for all interpolated output images. All these techniques can also be seen as special cases of *view morphing* [Wolb94], where the main difference is in the definition of corresponding samples.

If multiple color and depth values are stored for every pixel position, the image is called a *layered depth image* (LDI) [Shad98, Max96]. An LDI can be generated by either merging multiple images using image warping (which also works for

real-world scenes from registered photographs) or by using ray casting in a synthetic scene and storing the intersections of every ray with the scene. Because it is in general desirable to adapt an LDI's level of detail (i.e., its resolution) to different output image resolutions, Chang presented the *LDI tree* [Chan99], where LDI samples are stored using an octree-like data structure. Popescu et al. [Pope00] presented a special hardware, called the "warp engine," for displaying depth images.

Another image-based primitive called *light field* was introduced in 1996 by Levoy and Hanrahan [Levo96]. Simultaneously, Gortler introduced the same concept, calling it *lumigraph* [Gort96]. A light field describes a relatively large subset of the plenoptic function as a 4D function. If wavelength and time are not considered and the domain of the light field is a convex region in space (either "inside-out" or "outside-in looking") without obstacles, four dimensions are sufficient for a complete description of the plenoptic function. Two planes with a regular grid of sampling positions capture a light field: a camera is positioned at the sampling positions on the *entry plane* and records samples on the *exit plane*. The storage costs of a light field are high. Therefore, compression methods like vector quantization and entropy encoding are applied [Levo96]. Chai et al. [Chai00] propose "plenoptic sampling," which gives a minimum sampling density of images needed for the anti-aliased reconstruction of a light field if the minimum and maximum scene depth is known. However, since occlusions are not taken into account, image gaps are not avoided, so that the practical usability of the approach is restricted. In general, the huge amount of data still limits the use of light fields for scenes with larger extents.

So-called *surface light fields* [Wood00] encode the directionally dependent radiance for points on an object surface. Consequently, only the radiance per fixed sample position is encoded, whereas the geometry is given separately. Because parallax effects and occlusions are not encoded in the light field but represented by the geometry, a much better image quality compared to a standard light field can be achieved.

The image-based scene representations mentioned so far do not support *relighting*, i.e., light source modifications. Changing lighting conditions (such as moving shadows, specular highlights and anisotropic effects like metallic reflections) requires the whole representation to be rebuilt. To overcome this problem, Wong [Wong97] extends light fields using the *bidirectional reflectance distribution function* (BRDF) [Nico77]. This function encodes the relation between the incoming and outgoing light for every sample point, so that arbitrary lighting conditions can be reproduced at the cost of even higher memory requirements compared to light fields. The *bidirectional texture function* (BTF) [Disc98] is a discretized BRDF on an object surface. It is used for complex object surface descrip-

tions because it provides view- and illumination-dependent appearance changes and can be encoded so that it can be rendered with standard graphics hardware. For small surfaces it requires a tolerable amount of memory and can also be tiled or instantiated just like common textures [Suyk03, Meye01, Mese03].

2.2 Real-Time Rendering Acceleration

The process of generating images from a geometric scene description is carried out in the *rendering pipeline*. The typical rendering pipeline consists of three major stages:

- **Scene traversal:** The application manages a database in main memory and—in case of out-of-core rendering—also on harddisk. During the scene traversal, the geometry that is necessary to render an output image is extracted from the database and sent to the graphics hardware.
- **Geometry processing:** In order to generate an image of the scene, a transformation from world coordinates into device coordinates is performed for every graphical primitive. Clipping on the view frustum, lighting, texture coordinate calculation etc. are also performed in this step.
- **Rasterization:** Finally, every geometric primitive is rasterized, and color values for the covered pixels are stored in the frame buffer. A z-buffer test [Catm75] resolves the visibility between individual primitives. Additional operations in this stage include per-pixel lighting, texturing, blending, stencil operations etc.

In traditional rendering architectures, every primitive is processed separately. This means the computational effort grows with the number of primitives. Current hardware is able to render several million primitives per second, depending on the graphics driver, the geometry characteristics and hardware behavior. In the past, the main bottleneck in rendering systems was the sheer number of polygons sent down the pipeline. Therefore, many proposed rendering acceleration techniques aim on reducing the number of primitives already in the traversal stage. However, in order to accelerate the rendering process today, all aspects of the rendering hardware must be taken into account as will be described in Section 2.2.1. Rendering acceleration algorithms can be performed either in a preprocess or directly at runtime. The following issues must be considered when comparing the efficiency of different rendering acceleration techniques:

- **Acceleration:** Different approaches provide different orders of rendering acceleration. For instance, some algorithms provide rendering times sublinear in the number of primitives, whereas others provide only small increases of the frame rate under certain conditions. The additional effort introduced by the acceleration technique must always be balanced for the actual application.
- **Generality:** Some algorithms make assumptions about the input data. Other approaches exploit coherence between consecutive frames [Gröl93], allowing only limited changes of the output image over successive frames. Another point is how well the algorithm scales with extremely large scenes consisting of several million primitives. Whether such assumptions and/or concessions can be made mainly depends on the application.
- **Additional computational effort:** Acceleration algorithms may require unacceptably long preprocessing times. For online algorithms, the additional computational effort at runtime might or might not be worth the obtained acceleration.
- **Memory requirements:** The memory requirements for an acceleration data structure must be kept low on harddisk as well as in main memory and/or in graphics memory.
- **Image quality:** Reduced image quality for faster rendering is not tolerable for many applications. Some approaches progressively enhance the image quality at runtime if free resources are available.

Most acceleration algorithms allow trading off some of these issues to a certain degree. For instance, in case of impostor rendering, image quality and/or processing time can be traded for impostor memory.

We divide rendering acceleration techniques into 4 main categories: elimination of bottlenecks in the rendering pipeline, visibility calculations, level-of-detail techniques, point-based representations, and approaches that address frame to frame coherence using image-based techniques. The following sections describe these main strategies in more detail. Although in principle impostors also address frame to frame coherence, they are discussed in more detail in a separate section.

2.2.1 Eliminating Hardware Bottlenecks

Today, the performance of graphics hardware is growing at an enormous pace. On the other hand, the complexity of the graphics driver and hardware behavior

leaves much space for optimization in many cases. For instance, the geometry format (indexed geometry, display lists, etc.) can influence the output frame rate up to an order of magnitude because graphics drivers do not support all formats in an efficient way. Further important issues in this connection are hardware state changes (textures, shaders etc.), vertex caching performance and shader program complexity.

Three main bottlenecks in the rendering process can be identified. If the CPU cannot serve the GPU fast enough, the rendering is called *CPU bound*. In the graphics hardware, either the vertex transformation stage and/or the rasterization stage are the slowest stage of the pipeline. The small cache between transformation and rasterization stage means that only a certain triangle size lets both units work in parallel for a longer time. This causes shifting bottlenecks even during the rendering of a single object [Wimm03].

These facts show that several partly interacting aspects must be considered when optimizing the hardware rendering pipeline. Unfortunately, a *hard guarantee* for the maximum rendering time an output image needs cannot be given with current graphics hardware architectures. However, a *soft* rendering time limit can be met for a fixed 3D model and target rendering system by estimating the rendering time per frame as described by Wimmer and Wonka [Wimm03]. In Chapter 5, such an estimation will be used for the impostor placement algorithm, guaranteeing a maximum rendering time for each frame in a scene by using impostors.

2.2.2 Visibility Culling

Visibility culling algorithms calculate an estimation of invisible primitives to avoid sending (i.e. *cull*) them down the rendering pipeline. As a simple example, hierarchical view-frustum culling [Clar76] provides very effective visibility culling for many cases. View frustum culling tests a bounding volume (a box or a sphere) of an object against the view frustum. If the bounding volume is completely outside the view frustum, the object is culled. If this is done for nodes of a scene graph in a top-down fashion, large scene portions can be culled very efficiently.

Backface culling [Suth74] removes polygons whose normal does not face the viewer. Today this operation is performed in graphics hardware. If polygons with similar normals are grouped in a hierarchical way [Kuma96], larger scene portions can be culled before they are sent to the graphics hardware. Backface culling is used today in almost every real-time rendering system.

Occlusion culling culls scene parts (so-called *occludees*) that are *occluded* by closer ones (so-called *occluders*). This operation can be performed during runtime

or in a preprocess.

If the occlusion is computed at runtime, the problem reduces to a *from-point visibility* problem that can be solved in image space. In 1993, Greene et al. [Gree93] presented an online-occlusion culling algorithm called the *hierarchical z-buffer*. The scene is hierarchically organized using an octree, and the z-buffer is hierarchically partitioned using a quadtree (the so-called *z-pyramid*). The scene hierarchy is rendered in a front-to-back fashion, allowing efficient culling of invisible scene hierarchy nodes using the z-pyramid. Although the approach is conceptually very general and efficient, the practical usage of this approach is still not feasible with today's graphics hardware, since a lot of depth information would have to be read back from the z-buffer at every frame. Unfortunately, this is still a time-consuming operation. However, the idea has spawned a number of derived approaches, low resolution software implementations for coarse culling and fragment block culling in current graphics hardware. The *hierarchical occlusion map* presented by Zhang et al. [Zhan97] stores opacity values instead of depth values. Relying on a careful selection of occluders, only little information has to be read back from the graphics hardware. Because of this it achieves better acceleration compared to the hierarchical z-buffer, although it is conceptually less powerful.

On modern graphics hardware, so-called *occlusion queries* can be performed parallel to the rendering process. These allow the calculation of visibility for an object in image space, dependent on the content of the depth buffer. A good strategy for drawing good occluders first and then testing occludees is the key for an effective acceleration, in order to avoid the inherent latencies of the queries. This was shown in recent research by Bittner et al. [Bitt04].

Other online occlusion culling algorithms work in object space. Hudson et al. [Huds97] test a bounding volume scene hierarchy against shadow frusta of few large occluders in the foreground. Coorg and Teller [Coor97] also perform hierarchical tests using supporting and separating planes, while Bittner et al. [Bitt98] construct a shadow volume BSP-tree.

Calculating occlusion in a preprocess has the advantage of requiring only marginal online computation. To this end, the view space is partitioned into view cells, and for every cell *from-region visibility* is calculated. The result is a set of primitives that are likely to be visible, called the *potentially visible set* (PVS) [Aire90]. During runtime, only the PVS of the actual view cell is rendered.

For certain types of scenes, occlusion culling can be very effectively applied. For instance, architectural models provide excellent potential for occlusion culling because large scene parts are hidden by walls. This fact is utilized by Airey et al. [Aire90] as well as Teller and Sequin [Tell91]. View cells coincide with rooms, and *portals* are open connections between the cells, like for instance doors and

windows. Visibility between cells (so-called *cell-to-cell visibility*) is computed in a preprocess. Luebke and Georges [Lueb95] propose an efficient online method that accumulates portal intersections in image space until no intersecting region is left. The image-space bounding box of every object is then tested against the portal intersection of its cell and culled if invisible.

Visibility culling can also be efficiently used in urban scenes with 2.5-dimensional characteristic. Nearby building facades serve as excellent occluders, as was shown by Wonka et al. [Wonk00]. Occluder shadows are rendered into an orthographic map parallel to the ground. After reading back the map (which can be a bottleneck of this approach), depth values are used to decide on the visibility of every object. This occlusion-culling algorithm can also be performed at runtime [Wonk99] or in parallel to the rendering pipeline by another rendering system [Wonk01]. It will also be used in Chapter 5 in this thesis.

The calculation of effective occlusion culling for *arbitrary models* is quite difficult. Many algorithms assume large single primitives or large *water-tight* objects to serve as good occluders [Scha00]. Durand et al. [Dura00] propose *extended projections* as an extension of the hierarchical z-buffer algorithm for from-region visibility culling. Visibility is computed in image space using several occlusion maps for different scene parts. The intersection of all possible projections of an occluder in image space when seen from any point in a view cell is defined as its extended projection, while for an occludee, it is the union. Graphics hardware can be used for fast computation of the projections. The occlusion-culling technique for the layered impostor technique presented in Chapter 3 is conceptually related to this technique.

All from-region visibility algorithms mentioned above are *conservative* in the sense that they calculate a superset of the geometry actually visible from a view cell. For some scenes, a conservative solution does not provide enough occlusion and computing an exact visible set is by far too complex. Vegetation models are an example for such scenes. For those cases, Andujar et al. [Andú00] compute *approximate* visibility by allowing a certain amount of visibility error with respect to the output image. The approximative set of primitives is called *hardly visible set*.

To conclude, there is a huge amount of literature on visibility-culling techniques. Recent work of Bittner [Bitt02] as well as Nirenstein et al. [Nire02] even propose *exact* from region visibility algorithms with reasonable computational effort. Further examples as well as in-depth discussions on proposed visibility algorithms can be found in the theses of Durand [Dura99], Wonka [Wonk02] or Bittner [Bitt02].

2.2.3 Geometric Simplification

Geometric level-of-detail (LOD) techniques address the fact that small details are hardly noticeable from a distance, so less geometric primitives can be used per object in order to generate a similar output image. Clark [Clar76] was the first who made use of this fundamental issue in 1976.

In traditional geometry-based LOD techniques, several independent representations with different complexity and approximation quality are used for a polygonal model. We give a brief overview on LOD generation methods and LOD selections at runtime.

Much work has been published in recent years treating the generation of different level-of-detail representations. The proposed methods differ mainly in the simplification operations and the used error metrics. Algorithms working on manifold surfaces (so-called *mesh simplification algorithms*) use knowledge about topology for the simplification process. Local operations are performed on the mesh while keeping the introduced error as small as possible. Vertex removal and edge collapsing are common examples for such operations. Proposed error metrics include energy functions [Hopp93], the quadric error metric [Garl97] and simplification envelopes [Cohe96], to name but a few. A broad overview of these techniques is given in numerous tutorials, for instance [E. P97].

During runtime, an appropriate representation is chosen for model display. Various level-of-detail selection algorithms have been published that trade image quality for rendering acceleration.

One possibility for LOD selection is to set a desired image quality and minimize the number of primitives for display. Unfortunately, it seems to be practically impossible to include all relevant human perception issues for such a selection. In many cases, a simple distance-based metric is used: the farther the object is away, the less polygons are used for its representation. Schaufler and Stürzliner [Scha95a] and Luebke and Erikson [Lueb97] independently developed methods for using a hierarchy of representations to provide the desired image quality. A pixel-based screen-space error threshold was used in order to select the desired level of detail. Furthermore, these methods allow very high compression of arbitrary geometric scenes at the cost of significantly decreased image quality in case of a higher compression ratio.

Another approach is to meet a primitive budget and maximize the image quality. Funkhouser and Sequin [Funk93] presented a predictive algorithm for maximizing the image quality while maintaining a desired frame rate. This is done using a cost-benefit heuristic: the cost for an object basically describes the time it needs to render, while the benefit describes the importance of an object for an

output image. The resulting optimization problem is a special variant of the well-known knapsack problem [Maso99].

One important issue for LODs are smooth transitions when switching between different representations in order to avoid popping artifacts. Blending provides smooth switching, but both representations have to be drawn during the blending phase, which increases the number of primitives to be drawn [Möll02]. If geometric correspondences are established between different levels of detail, a *geomorphing* operation [Hopp98a, Alia96b, Alia98b] can be applied to overcome this problem. However, geometric morphing often looks unnatural on solid objects.

So-called *progressive meshes* [Hopp96], on the other hand, automatically avoid popping artifacts because no separate representations are generated, but the objects are either represented as wavelets [Loun97], using subdivision connectivity [Eck95], or a sequence of edge collapses. The method of Garland is most widely used today because of its satisfying results for many applications and its free availability on the internet. Improvements of progressive meshes allow an adaption of the mesh following special criteria: *view-dependent simplification* methods [Hopp97, Xia96] simplify a mesh with respect to the current viewpoint. This means, visually important parts like object silhouettes are tesselated finer than regions outside the view frustum or those facing backwards from the viewer. This approach is mainly useful for terrain rendering [Hopp98b] because of the high CPU overhead of maintaining the simplification data structure. Another criterium is the preservation of attributes such as material indices, color values or texture coordinates. Cohen et al. [Cohe98a] use textures and bump-maps for representing details for lower geometrical levels of detail. In this method, a parameterization of the model is necessary, which is only available for objects with known topology. Further information on this research is provided in the thesis of Cohen [Cohe98b].

Although much research has been done in the field of LODs, a general efficient simplification method which works on arbitrary geometry while at the same time preserving appearances is still not available. More specifically, an unsolved problem is the preservation of textural information when simplifying multiple objects simultaneously. The problem is that a common parameterization is not available for this case. The problem is exacerbated if different objects use different vertex and pixel shaders.

2.2.4 Point-based Representations

Polygonal model descriptions are powerful and efficient representations for 3D models. Different tasks like visibility calculations and collision detection were

largely solved with this representation. On the other hand, arbitrary distant scene parts covering only few pixels with millions of polygons cannot be displayed efficiently even today. For such cases, point-based representations have been a hot discussion topic in the scientific community in recent years.

In fact, points constitute a convenient representation for several classes of complex objects. By completely leaving topology information aside, relatively low storage requirements and fast rendering become possible. This is especially true for "soft" objects like explosions, fire, smoke etc., which were rendered using particle systems in the 80's by Reeves [Reev83]. Another example is vegetation, as was shown in the work of Reeves and Blau [Reev85], Weber and Penn [Webe95], and more recently by Deussen et al. [Deus02]. Max and Ohsaki [Max95] rendered trees from precomputed images with z-values, which can also be interpreted as a point-based representation but also as an LDI. This shows the close connection between point-based and image-based computer graphics. While both rendering techniques are based on a set of appearance samples of a model, the main difference is the format in which the samples are stored and processed.

When using points for representing a surface, one challenge is the adequate sampling, filtering and reconstruction of the surface in order to generate an output image. The aim is to support a properly filtered surface appearance for minified views and no holes for magnified views. The first work in this direction was done by Levoy and Whitted in 1985 [Levo85] for the special case of continuous and differentiable surfaces. They also addressed filtering issues for that case. In 1998 Grossman and Dally [Gros98] presented a point sample rendering approach for geometric models. They give a theoretical condition for the adequate sampling of an object and show that it can hardly be achieved in practice. Based on this insight, they present a sampling approach using orthographic cameras. Rendering is done using an incremental block-warping method with visibility computation based on a hierarchical pull-push algorithm.

Pfister et al. [Pfis00] obtained high-quality point-based rendering with *surfels*: points with shape and shading attributes that locally approximate a surface. The geometric object is sampled into three orthogonal LDIs. This structure is called *layered depth cube* (LDC). They generate a hierarchy of LDCs with precomputed texture information. The higher output image quality compared to conventional graphics hardware rasterization comes at the cost of usually huge storage requirements for the surfels and a high computational effort for the image generation process.

Points also allow a simple and natural generation of different levels of detail independently from the represented geometry. Catmull [Catm74] proposed al-

ready in 1974 that any geometric simplification may finally lead to points. The QSplat rendering system by Rusinkiewicz and Levoy [Rusi00] uses a bounding-sphere hierarchy for this. Similarly, an earlier approach by Chamberlain et al. [Cham96] already used a spatial hierarchy of colored boxes for representing far-away scene parts.

Recent research has focussed on high image quality [Zwic01], fast and hardware-assisted point data traversal [Dach03] and splatting [Coco02] for fast display, as well as memory-efficient point representations [Bots02]. This last work discusses further important characteristics of points. Points perform well compared to polygons in terms of memory consumption if many small geometric details have to be represented, as is the case for statues or trees. However, in the case of lower geometric complexity but rich color detail, textured polygons need by far less memory. Furthermore, until today there is no solution for occlusion culling for point-rendered geometry except in the special case of terrains, as was described by Stamminger and Drettakis [Stam01].

A different approach for the rendering acceleration of geometric models using points was presented by Wand et al. [Wand01]. Points are dynamically generated at runtime for the nodes of a hierarchy in a random fashion. The point sampling density of every node is chosen so that the resulting image contains no holes. Every node contains information that guides the algorithm for the efficient view-dependent choice of points. Unfortunately, visible scene parts may not be sampled sufficiently, leading to gaps in some objects. While this is hardly noticeable in vegetation environments, artifacts become visible in urban scenes, resulting in mixed fore- and background appearances. However, scenes of extreme complexity (up to 10^{14} triangles were reported) can be rendered at interactive frame rates, also by addressing frame-to-frame coherence. Wand and Straßer [Wand02] impressively show that random point sampling can also be used for the interactive display of dynamic scenes with over 90,000 moving objects.

2.2.5 Addressing Frame to Frame Coherence Using IBR

In a conventional rendering pipeline, every output image is rendered from scratch. In contrast, the idea of rendering acceleration techniques that are described in this section is to redisplay previously rendered output images in order to save rendering time. Frame-to-frame coherence [Gröl93] is explored so that the rendering speed can be decoupled from scene complexity to a certain degree. The main decisions to be made are which scene parts to update and which kind of image-based representation to use for them.

Reagan and Pose [Rega94] presented a hardware architecture called *virtual*

address recalculation pipeline that already shows many aspects of image-based rendering acceleration. In that approach, the scene is partitioned into depth layers based on the distance to the viewer. Every layer has its own video memory. Because changes occur more often for near scene parts, the layers are updated at different frame rates (the so-called *priority rendering*). The approach decouples the output frame rate from the rendering speed. The acceleration was more than one order of magnitude compared to the traditional rendering pipeline at that time.

Lengyel and Snyder [Leng97] enhance the concept of partitioning the output image into *coherent image layers*. The image layers are generated with respect to object perception in the fore- and background, different object movements, and efficient usage of texture memory. Rendering resources are then adaptively distributed so that fast-moving foreground objects get more rendering resources than a hardly changing background. The rendering system was implemented on Talisman [Torb96], a rendering hardware prototype that directly supports the rendering with such coherent image layers.

Mark et al. [Mark97] present another method called *post-rendering 3D image warping*. Only every n-th output image is rendered from geometry. Images in between are generated by composing two to three reference images rendered from previous and predicted future viewpoints. This ensures that most visible scene parts are present in the reference images so as to avoide image gaps. Another approach in this spirit is the work of Simmons and Sequin [Simm00]. Here the graphics hardware is used to reproject already cached radiance values by using a triangulated unit sphere centered at the viewpoint.

Chen et al. [Chen99] present a method that smartly incorporates geometric levels of detail with image-based techniques. A high-quality rendered image is mapped as a texture to a simplified mesh obtained from a conventional mesh simplification approach. The high-quality image is updated if a user-defined image quality is not met anymore. The authors show results of this technique for a terrain.

Wimmer et al. [Wimm99] divide the scene into a near and a far field. While the near field is rendered using conventional graphics hardware rendering, the far field is rendered using ray casting. They argue that with increasing distance, more geometric primitives cover an output pixel, so that ray casting performs better than conventional rendering. A radiance cache in the form of a panoramic image was used to keep the number of cast rays low if the viewer moves.

As a very simple but effective approach, Bishop et al. [Bish94] presented *frameless rendering*. Here, pixels are displayed instantly and in a random order. This allows displaying fluent animations if the output images cannot be rendered sufficiently fast, at the cost of reduced image quality due to varying times for the

update of every pixel.

In all methods described above, images are generated at runtime. A problem arises if too many scene parts must be updated between two consecutive output images. For that case, the rendering speed may drop significantly. If it is desired to keep the rendering speed high, a common way is to temporarily decrease the image quality.

2.3 Impostors

The border between impostors and the techniques discussed in Section 2.2.5 is not very sharp. For instance, the work of Mark et al. [Mark97] can also be seen as a "full-screen impostorization". However, the work described in this section is closer related to this thesis. We distinguish between the *impostor technique* that defines the impostor representation, and the *impostor placement* that defines the strategy for replacing scene parts with impostors in every output image.

2.3.1 Impostor Techniques

Impostors have to fulfil several demands like fast generation and display, low memory requirements and good image quality for a large viewing region (see Section 1.3). Several impostor techniques have been presented in recent work, each with the emphasis on different demands. Especially the problem of insufficient image quality (examples for this have been given in Section 1.3) has been addressed with combining different amounts of impostor geometry and appearance information. The following subsections describe previous impostor techniques in more detail.

Planar Impostors

In 1995, Maciel and Shirley [Maci95] introduced so-called *planar impostors*. They basically consist of simple planar geometry (for instance, a quadrilateral) with an alpha texture applied to it. Schaufler [Scha95b] generates such planar impostors dynamically at runtime. This is possible because the generation of a planar impostor is computationally not expensive. Displaying it is also very fast because of its very low geometric complexity.

On the other hand, *parallax movements* are not accounted for, so that the impostor needs to be frequently updated. In order to quantify the introduced error if the viewpoint is moved, Schaufler [Scha95b] presents a fundamental criterion

for the validity of planar impostors: as long as the so-called *error angle* between a point of the original object and its impostor representation falls below the angle of one pixel in the output image, the impostor is valid. Aliaga [Alia96a] (and also Shade et al. [Shad96]) mention to compute the error angle for every texel. However, this operation is usually much too costly. Schaufler [Scha95b] instead estimates the error angle for the cases that the viewer moves *sidewards* and *towards* the object by using a specific oriented bounding box of the object. This metric was also used later by Harris and Lastra [Harr01] for rendering clouds. In further related work [Aube99, Jaku00, Alia96a, Shad96], a simpler error estimation was presented, based on the angle the observer views an impostor in relation to the view position the impostor was generated. Furthermore, Aubel [Aube00] presented a special-purpose metric designed for impostors for moving humans, which is based on the distance between different body parts.

Another problem are *discontinuities* on the border between geometry and the impostor. Aliaga [Alia98b] proposes to distort the geometry the same way the impostor does it. In order to reduce popping artifacts that occur when switching from impostor to geometry and vice versa, Aliaga furthermore proposes to smoothly warp the represented scene part again. Of course, the output image is by no means correct, but the impostors are reported to fit smoother into the scene. Ebbesmeyer [Ebbe98] and Jakulin [Jaku00] achieve this by blending between different representations.

Furthermore, the *visibility* between the scene and a flat impostor must be solved satisfactorily. Therefore, Schaufler [Scha97] introduced the so-called *Nailboards*, which store per-pixel depth information in order to calculate the visibility with a two-pass rendering algorithm. Note that Nailboards use depth information only for resolving visibility and not for supporting parallax effects. Aubel et al. [Aube99] instead split the object into multiple small non-overlapping impostors. However, this leads to disturbing discontinuities at the impostor borders [Aube00]. Therefore, they mention to store all overlapping scene parts into one single impostor (they call it *factorized impostor*), at the cost of having to update the impostor frequently. As a second solution, they propose to solve visibility when the impostor is generated and only store visible pixels (they call it *depth corrected impostors*). However, this means that if any occluding scene part or the camera moves, the impostor has to be updated, which might be very costly. Harris and Lastra [Harr01] render clouds as impostors generated using particle systems. For resolving visibility of a cloud with an object, they split it into a foreground and a background part and render an impostor for each of them. In contrast to Aubel et al. [Aube00] (who rendered solid objects), this method works sufficiently correct, due to the fuzzy appearance of clouds.

Layered Impostors

In 1998, Schaufler [Scha98a] presented *layered impostors*, which are able to simultaneously reproduce parallax movements, solve the visibility satisfactorily correct and integrate seamlessly into the scene. The idea of layered impostors is to use multiple image layers at different distances from the viewer, each representing a certain depth range. Schaufler shows how the parallax error decreases with increasing numbers of layers. In order to capture as many potentially visible scene parts as possible, Schaufler [Scha98b] uses a view frustum centered *behind* the object for the impostor generation process. In order to avoid image gaps between the layers, slightly overlapping depth regions are rendered into each layer. However, no guarantee can be given for avoiding image gaps, as will be seen in Chapter 3. Compared to planar impostors consisting of only one image layer, layered impostors provide a larger valid viewing region. Jakulin [Jaku00] shows that this technique can be used for complex objects with impressive image quality. However, the memory requirements for the image layers are quite high. In Chapter 3 we will present a new variant of layered impostors that guarantee both, a representation without any image gaps and very low memory requirements.

Video-based Representations

Wilson et al. [Wils00, Wils01] stored planar impostors of adjacent view cells in an MPEG2 video stream. The MPEG video compression standard provides relatively high compression rates for the representation. Although video encoding provides fairly good compression rates resulting in low memory costs, high-frequency scene content causes aliasing artifacts and results in lower compression rates. Unfortunately, aliasing artifacts occur often when rendering distant scene parts. Another problem is that since video-based impostors actually constitute flat images, a correct visibility calculation is not possible for dynamic scene content.

Textured Depth Meshes (TDM)

The basic concept of *textured depth meshes* [Dars97, Sill97] is to triangulate a rendered image with respect to depth discontinuities (like object silhouettes) based on the z-buffer. Planar image regions are represented using individual textured polygons, whereas a tradeoff between approximation accuracy and mesh complexity must be found for non-planar regions. TDMs provide good parallax movement reconstruction and visibility is also solved satisfactorily. They need only little additional geometry and storage compared to planar impostors.

Several methods have been proposed for TDM generation. Darsa et al. [Dars97] use a Delauney triangulation based on the Voronoi diagram of irregular sparse samples generated by a ray tracer. In that work, every triangle is assumed to have only one color, so that color and depth must be approximated simultaneously. In later work, Darsa et al. [Dars96, Dars98] obtain the triangulation only from the depth component of the samples. Color is applied to the depth mesh using textures. Sillion et al. [Sill97] extract silhouettes and depth discrepancy lines from the z-buffer, thus creating several image regions. For triangulating these regions, equally spaced points are inserted, followed by a constrained Delauney triangulation. Aliaga et al. [Alia99a] instead build a very dense regular mesh with image resolution which is then simplified using a fast greedy algorithm that merges planar regions. Afterwards, an accurate but also computationally expensive mesh simplification algorithm [Garl97] is applied. Wilson and Manocha [Wils03] use first the approach of Garland and Heckbert and afterwards the view-dependent simplification algorithm of Luebke and Erikson [Lueb97].

In order to reduce the required storage space on harddisk, Decoret et al. [Deco99] suggest to generate the mesh geometry in a preprocess (because this is the most time-consuming operation) but the textural information on the fly.

Textured depth meshes suffer from disocclusion artifacts, typically visible as *rubber-sheet effects*. They are caused by distorted meshes that cover geometry not represented in the TDM. In order to avoid such artifacts, all potentially visible scene parts should be captured at a reasonable sampling density. Therefore, Darsa et al. [Dars96, Dars98] use a measurement for every triangle that indicates the quality of the sampling. If the quality is not sufficiently high, triangles from multiple TDMs generated near the actual viewer position are displayed. Similarly, Wilson et al. [Wils03] sample the geometry incrementally. This means, the visual error is estimated for a TDM, and the sampling process is repeated for a new viewpoint that is assumed to provide the missing information. This process stops if the sampling density is sufficiently high. Afterwards, redundant information is removed and textured depth meshes are constructed from the remaining data. During runtime, multiple TDMs are rendered in order to minimize visual artifacts. Although the algorithm may provide satisfying image quality in many situations, no real guarantee is given that the error in image space is smaller than a certain value, and many TDMs have to be rendered for complex views.

Another method for avoiding disocclusion artifacts was presented by Decoret et al. [Deco99], who generate multiple meshes at different distances from the viewer. *Visibility events* are introduced in order to quantify depth discrepancy and thereby disocclusion artifacts. During the mesh generation process, objects are treated successively. Every object is inserted into an existing mesh if the depth discrepancy is low enough. If an object fits into multiple meshes, the one with

least wasted texture space is used. If the object fits in no mesh, a new mesh is generated. Furthermore, if rendering resources are available at runtime, meshes are dynamically updated, which further increases the accuracy of the representation.

In Chapter 4, we present an approach for generating TDMs that guarantees that no rubber sheet effects become visible in the first place while at the same time keeping the mesh complexity and texture memory reasonably low.

Per-Pixel Depth Information

Depth images contain per pixel depth information for displaying images for new viewpoints. The transformation for projecting samples to a new image is called *3D image warping* [McMi97] and is a special case of general image warping [Wolb94] in 3D space. The warp can be performed in *forward* direction by projecting every pixel into the output image. In contrast, searching the correct representant for every output pixel in the input image is called *backward warping*. The latter can be seen as a kind of ray tracing and is only useful if few pixels have to be displayed because of the often costly search operations. For avoiding holes in the output image in case of magnified viewing, depth images use the same techniques as point-based representations, like for instance point splatting or the triangulation to fine meshes. Both methods were already discussed in the context of image-based rendering [McMi97]. In fact, a depth image can also be seen as a point cloud with a regular grid structure.

The per-pixel depth information allows a correct reproduction of parallax movements and makes it possible to resolve visibility with the geometric scene. However, although the impostor display can be done in graphics hardware today, warping depth images is a relatively costly operation compared to polygon-based representations because many per-sample transformations are performed. Popescu et al. [Pope98] shows how image warping can be parallelized, which is unfortunately not supported by common hardware.

Depth images also suffer from disocclusion artifacts in the output image. In order to reduce them, multiple images can be warped to fill image gaps [Raff98a, Mark97]. Another possibility is to use layered depth images that contain multiple samples for every sampling position [Shad98, Pope98, Alia99c] (see Section 2.2.5). More recently, a very high-quality impostor technique was presented by Wimmer et al. [Wimm01], so-called *point-based impostors*. First, a set of points is obtained by ray casting from three viewpoints from within the view cell and removing redundant information. Wimmer provides a sampling algorithm that tries to make sure that no holes become visible. They report on about two points required per screen pixel for the impostor. For every point, view-dependent ap-

pearance is applied using Monte-Carlo ray tracing, thus generating a small point-light field. This light field is encoded into a texture so that the rendering can be done using graphics hardware. The result are high-quality anti-aliased impostors without image gaps. However, the reported preprocessing times are relatively high. Furthermore, it might be argued that the high-quality impostors may stand out in the output display if the rest of the scene is rendered using OpenGL rendering. Therefore, the impostor technique is perfectly suited for applications that need very high-quality rendering without aliasing artifacts.

Billboard Clouds

Decoret et al. [Deco03] recently presented an impostor technique called *billboard clouds*. The basic principle is to represent an object by a set of alpha-textured planes. In contrast to planar impostors, the position and orientation of the planes is optimized for approximating the object geometry so that a given error tolerance in object space (or in image space [Deco02]) is met. The main advantage of this view-independent impostor technique is that the viewer is not restricted to a bounded view cell, but only to a minimum distance to the impostor. Another advantage of view independent-techniques is that they can also be used directly for standard shadow mapping algorithms. Relighting can also be applied to the billboard clouds, as was shown by Decoret et al. [Deco03]. The technique can be used for arbitrary geometry and shares concepts of image-based as well as geometry-based simplification. Current implementations of the technique show very long preprocessing times and image artifacts, especially on curved surfaces. Furthermore, high memory requirements as well as relatively slow rendering are caused by large planes containing many transparent texels. However, the basic idea is promising and it seems like many of the problems can be overcome in further research.

2.3.2 Impostor Placement Strategies

The basic aim for an efficient impostor placement is the efficient use of memory for static impostors and keeping the computational effort for the impostor update as low as possible for dynamic impostors. The decision in what way impostors are placed into a scene mainly depends on the scene characteristics: a priori scene information is typically utilized for an efficient impostor placement. In the following subsections, we discuss previous impostor placement strategies for different types of scenes.

Per-Object Impostor Placement

In this type of impostor placement, whether a scene part is rendering from geometry or using an impostor is decided separately for every object. This decision is typically based on the size of the object and/or the distance between object and viewer. The impostors efficiently represent complex objects from a certain distance, because in this case they cover only few pixels on the screen and parallax effects are rather small so that the update frequency is low.

When using precalculated per-object impostors, view-independent techniques [Deco03] have the advantage that they represent the object from all sides, which can save much memory. Maciel and Shirley [Maci95] also described some view-independent impostors that were designed for special objects like textured boxes for box-shaped objects. For view-dependent impostors, multiple impostors are necessary for a complete object representation, each representing a certain angular range. At runtime, the one generated from the reference viewpoint nearest to the actual viewpoint is displayed. Depending on the application, the reference viewpoints can be distributed on a sphere [Alia99b] (a so-called *image sampling sphere*) or on a circle [Jaku00] in case of ground-level movement. Such static impostors are especially useful if the represented object is very complex and instantiated in the scene. Jakulin [Jaku00] shows this for the case of trees that are recorded from six viewpoints using a layered impostor technique [Scha98b]. This results in four megabytes per tree, so that impostors for numerous trees can be stored in graphics hardware. By instantiating the trees, a highly complex forest scene can be efficiently represented using impostors without the need of enormous amounts of impostor memory.

In contrast, the efficiency of dynamic impostors [Scha95b] depends on how the speed gain obtained from the impostor display makes up for the additional effort for impostor updates. Harris and Lastra [Harr01] represent clouds with dynamic impostors. Because the clouds are internally represented as particle system, impostors are also efficient for nearby clouds because they are much faster to render than the particles. Furthermore, because clouds do not contain high-frequency appearances like sharp edges, popping artifacts and blocky pixels are not a big problem. In contrast, the impostors are even used in order to avoid artifacts that would occur when directly rendering the particles.

Impostor Placement in Architectural Models

For architectural models, cell and portal approaches provide efficient visibility culling (see Section 2.2.2). When impostors are placed in portals so that they represent the geometry of neighboring cells, only the current cell has to be rendered

from geometry. If the viewer comes near a portal, the adjacent cell is also rendered from geometry in order to avoid image artifacts. Portals are a convenient place for using impostors because the complex geometry behind a portal covers only few pixels on the screen. Furthermore, because the overall number of portals in a model is quite limited, precalculated impostors often fit completely into main memory or even on graphics hardware. Aliaga and Lastra [Alia97b] use planar impostors placed in the portals. In order to obtain a reasonable image quality, multiple impostors are generated using reference viewpoints distributed on a semicircle in front of the portal. In contrast, Rafferty et al. [Raff98a] warp depth images in order to reduce the number of impostors needed for a convincing representation. In order to avoid disocclusion artifacts, two depth images are warped simultaneously to the output image. Popescu et al. [Pope98] instead use a layered depth image (LDI) for every portal. For fast LDI display, they use a parallel warping algorithm and clip invisible samples using a hierarchical data structure for the LDI. In contrast, Simmons and Sequin [Sequ01] use textured depth meshes with three TDMs per portal, each recorded from another viewpoint. Rubber sheets are avoided by composing the TDMs during rendering. Rafferty et al. [Raff98b] give a comparison of the approaches from Aliaga, Rafferty and Popescu. Furthermore, while online generation of portal impostors is also possible, Rafferty et al. [Raff98a] reported “animation hickups” due to the too time-consuming impostor generation process parallel to the model display.

Placing Impostors in Distant Scene Parts

This impostor placement is done by partitioning the view space in a set of view cells and for every cell, the scene is split into a *near field* and a *far field*. While the near field is rendered using geometry, the far field is displayed using impostors. Distant scene parts often show complex geometry that covers only few pixels on the screen. Such scene parts allow high rendering acceleration and at the same time low memory requirements for impostors. Furthermore, such impostors are valid for a large view region because of only few apparent parallax movements due to the large distance to the viewer.

Aliaga et al. [Alia97a] describe a simple method called *textured box culling*, were the far field is represented by a cube consisting of 6 textured quads. This results in visible popping artifacts when the view cell is changed. In later work, Aliaga et al. [Alia98a] use textured depth meshes similar to Darsa et al. [Dars98]. Wilson et al. [Wils00] instead use video-based impostors (see Section 2.3.1).

In contrast to these approaches, Sillion et al. [Sill97] place textured depth meshes in an urban environment at the end of street segments. They assume that facades close to the viewer completely occlude the scene behind. This greatly

reduces the required impostor memory, because visibility is calculated before impostors are generated (similar to portal impostors).

Another interesting application is the *guarantee* of a frame rate through the use of far-field impostors. A common way to obtain this is to shrink the near field until the number of contained primitives falls below a certain threshold [Wils01, Wils03]. Aliaga et al.[Alia99a] instead balance the image quality error caused by a TDM representation with the image error caused by geometric LOD simplification in the near field by adapting the distance of the border between near and far field.

In 1999, Aliaga et al. [Alia99c] describe a far-field impostor placement that uses impostors (i.e., LDIs) only where they are necessary so that a prescribed primitive budget is not exceeded, depending on the view position and -direction. They subdivide the view space using a grid of points adapted to the local model complexity. For every grid point and view direction, an optimization algorithm selects a model subset which is represented using an LDI so that the primitive budget is met. The model subset is chosen using a cost-benefit heuristic aimed at low memory requirements. This means that small, distant and complex model subsets are preferred to large, nearby and less complex ones.

However, all placement algorithms discussed above share some limitations: first, the placement is always done on a per view-cell basis. Consequently, for distant scene parts, many similar impostors are generated for every cell, which is of course highly inefficient. Second, the algorithms are aimed to limit the number of primitives, which is hardly coincident with the frame rate as was discussed in Section 2.2.1. Furthermore, additional acceleration techniques to reduce the required impostor memory, like visibility calculations, were only used in one previous approach [Sill97]. The impostor placement approach presented in Chapter 5 overcomes these drawbacks and achieves much better placements.

Hierarchical Image Cache

Schaufler and Stürzlinger [Scha96b] as well as Shade et al. [Shad96] concurrently presented rendering systems using dynamic impostors, called *hierarchical image cache*. In these approaches, impostor placement is done based on the nodes of a hierarchy of the input model, with nearby objects clustered first. During runtime, impostors are dynamically generated for the leaf nodes of the hierarchy. Impostors for intermediates nodes are generated from the impostors of their children. The impostor update is based on the image-space error metric of Schaufler [Scha95b]. In order to render an output image, the tree is traversed, and the first node containing a feasible impostor is displayed and subtree traversal is discarded. Consequently, with increasing distance to the viewer, impostors are displayed for ever

higher hierarchy nodes, thus greatly accelerating the rendering process.

The hierarchical image cache relies on coherent output images. A problem may arise due to numerous impostor updates if the viewer moves very fast or the view is rotated so that many previously invisible nodes have to be generated. Furthermore, the nodes close to the viewer need to be updated frequently. Very close nodes are rendered from geometry due to the inefficiency of impostors in close range. Shade et al. calculate the lifetime of an impostor based on the distance to the viewer, and an impostor is only generated if its lifetime justifies the generation cost. Furthermore, they mention the predictive generation of impostors in order to avoid sudden drops of the output frame rate if many impostors need to be instantly updated. Using visibility calculations for reducing the number of impostor updates might be an interesting extension to that approach. However, the user movement needs to be restricted to avoid too many impostor updates for a frame. This is the main cost introduced by the use of dynamic impostors.

Impostor Placement using Funkhouser's Adaptive Display Algorithm

Maciel and Shirley [Maci95] enhanced the work of Funkhouser and Sequin [Funk93] by introducing *static* impostors into the predictive level-of-detail selection algorithm (see Section 2.2.3). They replace nodes of an input model hierarchy with impostors, taking advantage of the fact that impostors can be applied to arbitrary scene parts. This allows displaying many objects faster and with higher image quality than graphics hardware capabilities would allow for a geometric representation. However, huge texture memory requirements were also experienced, and they also mentioned prefetching impostors dynamically from harddisk.

In contrast, Schaufler [Scha96a] shows how *dynamic* impostors can be incorporated into that framework. In addition to just displaying an object with an appropriate level of detail, an impostor is generated and displayed using a certain level of detail if the object is suitable for an impostor representation. If impostors are displayed instead of original objects, more frame time is left for generating higher quality impostors from better LOD selections. This means that the image quality is improved progressively over time if the user does not move. In contrast, in the original framework of Funkhouser and Sequin, the LOD selection is always the same if the output image does not change, so that the image quality cannot be improved over time.

2.4 Summary and Conclusion

In this chapter, an overview of impostor-related research was given. Impostors are a form of image-based rendering (Section 2.1). Common problems include high memory requirements, image artifacts caused by disocclusions and a strong reliance on coherence in image space.

Section 2.2 gave a brief overview over other rendering acceleration techniques related to this thesis. Although visibility culling often provides dramatic frame rate increases, it cannot provide any guarantee about the output frame rate, because the visible geometry might still exceed the capabilities of the rendering system. In such cases, additional acceleration techniques are required. Theoretically, visibility culling together with geometric simplification techniques could solve the problem. However, this latter technique does not work for arbitrary geometry, i.e., disconnected textured objects cannot be merged efficiently while preserving their textures. One option is to use point-based representations. While points work very well for instantiated geometry as was shown for plants in ecosystems, high memory requirements prevent points generated in a preprocess from being used for arbitrary scenes. In this context, the idea of generating the point samples at runtime from internal representations [Wand01] is promising. This allows a simple tradeoff between image quality and frame rate.

The image-based rendering acceleration techniques presented in Section 2.2.5 strongly rely on coherence in image space. If this is not provided, they perform poorly. In general, it is not possible to give any frame rate guarantee for such techniques without limiting the user velocity, or significantly lowering the image quality. This fact is also true for the hierarchical image cache, where simple planar impostors are used for fast online generation. Generation in a preprocess, on the other hand, entails large impostor databases due to the number of required impostors. In order to reduce the number of impostors, more complex impostor representations with support for parallax movements are used. However, such more sophisticated impostor methods still struggle with problems like huge memory requirements and/or long preprocessing times. Another problem is image quality: visibility gaps often occur due to insufficient scene information. Up to now there is no general approach for *guaranteeing* the prevention of such artifacts with reasonable effort. In Chapter 3 and Chapter 4 we present solutions to this problem.

Impostor placement algorithms have been presented that try to give a guarantee for a minimum output frame rate for every view in a scene. This was done by bounding the number of primitives visible in every output image, but today, this measure is hardly related to the frame time (see Section 2.2.1). Furthermore, all approaches share problems like long preprocessing times, very high memory

requirements and uncertain image quality. This comes due to the fact that not all aspects of the impostor placement problem were addressed, and additional rendering acceleration techniques have not been used in order to reduce impostor memory. In Chapter 5, we present an impostor placement approach that takes all aspects of the problem into account. We will show that for moderately complex scenes, a sensible impostor placement together with new efficient impostor techniques can provide both, guaranteed frame rates and low memory requirements so that the whole impostor database can usually be stored in graphics memory. In this case, it is not necessary to restrict the user movement speed, because there are not memory prefetching issues.

Chapter 3

Memory-Efficient Layered Impostors without Image Artifacts

3.1 Introduction

Several impostor techniques for accelerating the rendering of complex distant scene parts have been proposed in previous work (see Section 2.3.1). However, there is still no technique that simultaneously addresses all problems stated in Section 1.4. More specifically, the problem of generating an impostor that is guaranteed to show no artifacts for a large viewing region and needs only a small amount of memory in reasonable time has not been solved in a satisfying way.

In this chapter, a new impostor technique for representing distant scene parts is described which addresses all these issues simultaneously. It is based on partitioning the relevant scene part into multiple image layers with varying depth [Scha98b, Lacr94, Meye98, Rega94]. We introduce a special sampling method for the impostor generation process which guarantees that all visible scene parts are acquired without having any knowledge about the scene structure. In combination with a new layer arrangement, this guarantees the absence of image gaps in the final impostor. We will present a number of algorithms for efficiently combining the acquired information with impostor geometry in order to obtain a compact representation in a very short time. We will also show the suitability of the impostor technique for a number of different applications.

3.2 Overview

The layered impostor generation process requires as input

- the scene part to be represented,
- the view cell (shape, size and position),
- the output image resolution and the field of view used for impostor display.

Before the impostor generation starts, the camera for recording the impostor is set to a fixed position within the view cell, the so-called *reference viewpoint*. The camera frustum is defined so that it tightly encloses the scene part to be represented (Figure 3.1 (a)). Using this setup, a layered impostor is generated by

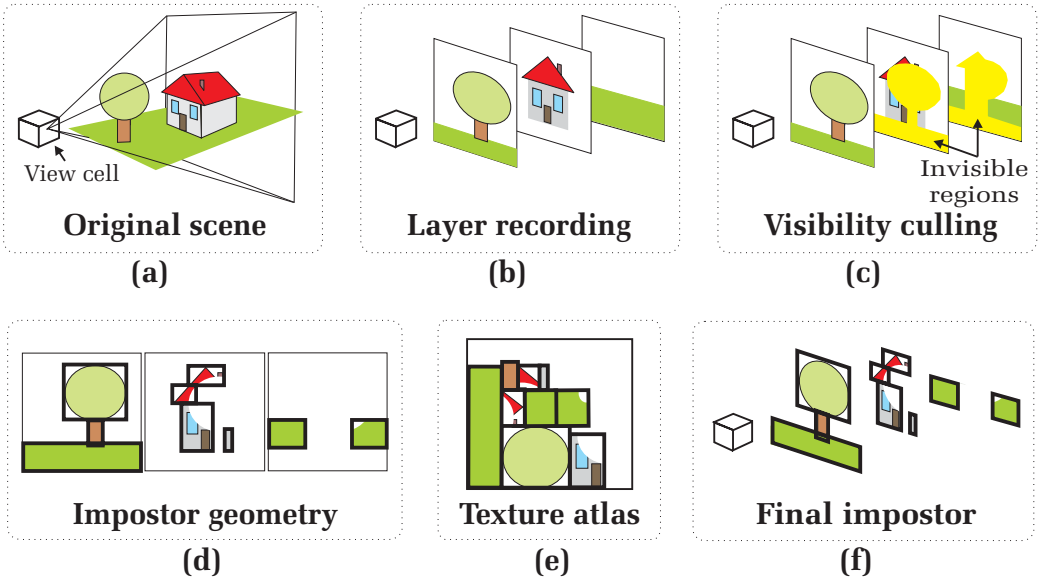

Figure 3.1: Main steps for layered impostor generation.

applying the following steps (see Figure 3.1):

1. **Layer setup calculations:** The positions and depth ranges of the individual impostor layers are calculated (see Section 3.3.1) so that in combination with a special rendering technique for recording a layer, the final impostor will show no image gaps.
2. **Layer recording:** Render the geometry for every layer into a texture (Figure 3.1 (b)) with the near and far clipping plane set so as to define the respective depth range. Impostor texels that contain no information are set transparent via the stencil-buffer functionality of current graphics hardware. Furthermore, all available features of current graphics hardware like shading effects are automatically applied. A special rendering technique guarantees that no scene information is missed (see Section 3.3.2).

3. **Visibility culling:** Exclude scene parts that are not visible from the view cell (Figure 3.1 (c)), which helps reducing texture storage requirements of the impostors. The special layering allows doing this in a very fast and effective way (see Section 3.4).

4. **Impostor geometry generation:** Partition every layer into a set of textured polygons (i.e., quadrilaterals) that are "well filled" (Figure 3.1 (d)). The exclusion of transparent texture areas reduces the impostor memory requirements to a tolerable level, while the additional polygons required place no burden on the graphics hardware (see Section 3.5).

5. **Texture atlas generation:** Combine the textures of all polygons into a large texture atlas (Figure 3.1 (e)), because a separate rendering of all small textures is not efficient due to numerous hardware texture switches (see Section 3.6).

Finally, the impostor polygons are inserted into the scene so that every texel appears in the output image exactly at the position it was recorded (Figure 3.1 (f)).

The advantages over previous impostor techniques (see Section 2.3.1) are an artifact-free representation (which is provided by step 1 and 2) and very low memory requirements (provided by steps 3 and 4). This is achieved while providing a very fast impostor generation. Also note that in order to save memory during the preprocess, steps 2 to 4 can be performed successively if the layers are processed in front to back order.

3.3 Scene Layering with Prevention of Image Artifacts

In this section, we describe a method for preventing image gaps in a layered impostor representation. The basic idea consists of two issues:

- **Appropriate layer spacing:** The distance between two adjacent impostor layers is chosen so that they do not move more than one texel against each other when seen from within the view cell. We call this property the *one-texel layer spacing* (see Section 3.3.1).

- **Complete scene recording:** The rasterization process used for impostor layer recording is modified so that no scene parts are missed (see Section 3.3.2). In addition, scene parts that are present in two adjacent layers, the transition between the layers is drawn using identical texel positions.

In summary, because every texel exposes not more than the texel behind it and all layer transitions are represented with identical texels, it is not possible to look "through" a transition from the view cell. For this reason we can guarantee that no scene part shows any image gaps, even if it is represented by multiple layers.

3.3.1 Layer Placement Calculation

This section describes the calculation of the position and the depth range of every impostor layer. Two issues have to be taken into consideration in this connection. First, the one-texel layer spacing must be fulfilled in order to guarantee an artifact-free representation. Therefore, the following subsection gives a scheme for calculating the positions of the impostor layers so that this requirement is always fulfilled. The second issue are parallax errors introduced by the impostor which must be quantified and limited. This issue is important for calculating the depth range of every layer, as will be presented further below.

One-Texel Layer Spacing

The goal is to place the layers close enough so that for every view within the view cell, each texel uncovers at most the texel behind it in the following layer. Without loss of generality, for all further illustrations the object is assumed to be centered in front of the view cell. Furthermore, the reference viewpoint is positioned at a vertical line through the view-cell center as is shown in Figure 3.2 (left and middle). This is the best horizontal position because all errors introduced by the impostor are equally distributed to the left and right side. The optimal vertical position will be discussed further below. For this symmetric setup, all following considerations can be reduced to one side of the view cell (we choose the left side for explanation).

Figure 3.2 ((a) and (b)) shows for every texel in a layer a line indicating where the texel in the following layer is completely uncovered when moving to the left. This means an image gap will occur behind a texel when viewed from a position to the left of this line. The one-texel layer spacing has to be calculated so that no such line crosses the view cell.

First we will show that all these lines meet in a point at the same height as the reference viewpoint, because this reduces the consideration to only the outermost lines. Assume a given reference viewpoint P and two impostor layers with distances a and $a + c$ as shown in Figure 3.2 (c). The size of a texel in the layer with distance a is defined as p. For every texel, a triangle is formed by the line that indicates when the texel behind it is completely uncovered, and a line from P to

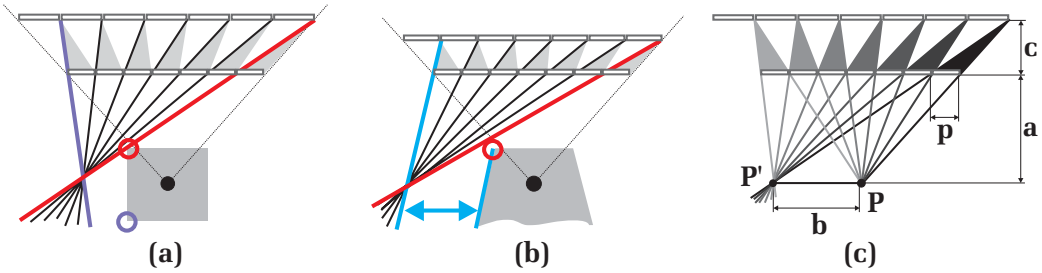

Figure 3.2: Movements of one texel when seen from within a rectangular (a) and shaft-shaped (b) view cell. The lines indicate for every texel when the texel behind it is completely uncovered when moving to the left. (c) shows the setup for the derivation of texel movements between two layers.

the right border of that texel. This triangle is shown in different colors for every texel in Figure 3.2 (c). From similar triangles we get:

$$\frac{c}{a+c} = \frac{p}{b}.$$

Because p, c and a are constant, b is also constant, i.e., independent from the actual texel position in the layer. Consequently, all lines meet the single point P' as can be observed in Figure 3.2 (c).

This in turn means that the line of the leftmost and the rightmost texel define the right boundary of *all* lines (see Figure 3.2). In order to avoid image gaps, it is sufficient to ensure that these two lines do not cross the view cell. For the following considerations we will call them *left boundary line* and *right boundary line*, respectively.

The layer calculation assumes the following input parameters (see Figure 3.3 (left)):

- Either a rectangular view cell with width n and depth m, or a shaft-shaped view cell with an apex angle β. Furthermore, the distance s from the reference viewpoint to the view-cell border.
- The distance t_0 between the object and the view cell, and the width w of the object. The frustum used for the impostor generation is assumed to tightly fit the bounding box, as is depicted in Figure 3.3 (left).
- The impostor texture resolution res, which is chosen so that the impostor texture is not magnified when seen from any viewpoint within the view cell.

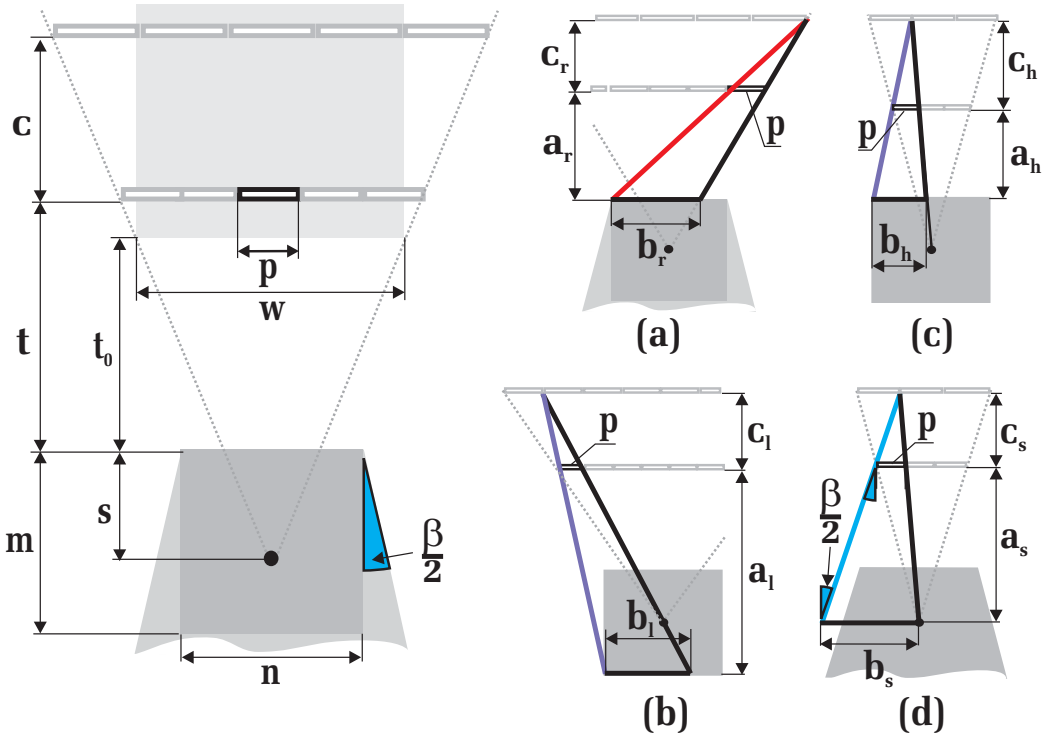

Figure 3.3: Left: setup for the layer spacing calculation based on the maximum allowable texel movement. Right: (a) for the right boundary line (identical for rectangular and shaft shaped view cell), (b) and (c) for the left boundary line for a rectangular view cell, (d) for the left boundary line for a shaft shaped view cell.

The resolution can be calculated using the method of Schaufler [Scha95b]. The size of a texel p at distance t (see Figure 3.3 (left)), is then defined as:

$$p = \frac{w(t+s)}{res(t_0+s)}.$$

Given a layer distance t, the distance c from that layer to the following one is calculated so that the boundary lines just touch the view cell. c is calculated independently for the left and right boundary line, and the smaller value is used. This is done for all cases by using similar triangles:

$$\frac{c}{a+c} = \frac{p}{b}, \tag{3.1}$$

Figure 3.3 ((a)-(d)) shows the respective configurations for a and b for the left and right boundary line, which are now discussed.

For the right boundary (see Figure 3.3, (a)) the definition is

$$\begin{aligned} a_r &= t, \\ b_r &= \frac{n}{2} + \frac{sw}{2(t_0+s)}. \end{aligned}$$

Substituting a and b in Equation 3.1 results in the distance c_r:

$$c_r = \frac{t}{\frac{res}{2(t+s)}\left(\frac{n(t_0+s)}{w} + s\right) - 1}. \tag{3.2}$$

This equation holds for the right boundary of rectangular as well as of shaft-shaped view cells.

For the left boundary line, the layer computation is first considered for a rectangular view cell. A distinction must be made depending on whether the left boundary of the impostor layer ends to the left or to the right of the left boundary of the view cell. This indicates whether the left boundary line touches the lower (see Figure 3.3 (b)) or upper (see Figure 3.3, (c)) view-cell corner.

The definition of a and b for the lower view-cell corner (Figure 3.3, (b)) is

$$\begin{aligned} a_l &= t + m, \\ b_l &= \frac{n}{2} + \frac{(m-s)\left(\frac{w}{2} - \frac{w}{res}\right)}{t_0+s}. \end{aligned}$$

Equation 3.1 results in the distance c_l:

$$c_l = \frac{t+m}{\frac{res}{2(t+s)}\left(\frac{n(t_0+s)}{w} + (m-s)\left(1 - \frac{2}{res}\right)\right) - 1}. \tag{3.3}$$

The definition for the upper view-cell corner is analogous (Figure 3.3, (c))

$$\begin{aligned} a_h &= t, \\ b_h &= \frac{n}{2} - \frac{s\left(\frac{w}{2} - \frac{w}{res}\right)}{t_0+s}, \end{aligned}$$

with Equation 3.1 resulting in the following distance c_h:

$$c_h = \frac{t}{\frac{res}{2(t+s)}\left(\frac{n(t_0+s)}{w} + s\left(\frac{2}{res} - 1\right)\right) - 1}. \tag{3.4}$$

For shaft-shaped view cells, in order to ensure that the view cell is not intersected by the left boundary line, the line must be *parallel* to the left border of the shaft (see Figure 3.3, (d)). a and b are defined as follows for this case:

$$\begin{aligned} a_s &= t + s, \\ b_s &= \frac{(t+s)w}{2(t_0+s)} + (t+s)\tan\left(\frac{\beta}{2}\right). \end{aligned}$$

Equation 3.1 results in c_s:

$$c_s = \frac{t+s}{res\left(\frac{1}{2} + \frac{t_0+s}{w}\tan\left(\frac{\beta}{2}\right)\right) - 1}. \tag{3.5}$$

Up to now, the layer placement calculation was presented for one dimension. The extension to the general 2D case is quite simple because all texel movements are independently for the x and y direction. This means that c must be calculated for *both* directions and the smaller value is used in order to ensure that no gaps occur in any direction.

An interesting question is the choice of an optimal reference viewpoint, i.e., the choice of s. The best choice maximizes the inter-layer spacing for the left *and* right boundary line, because the smaller value has to be used. Figure 3.4 (a-c) shows an example how the position of the reference viewpoint affects the inter-layer spacing. The spacing c_r for the right boundary line always decreases with

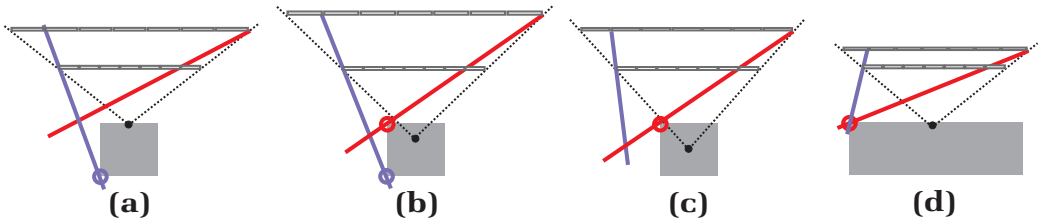

Figure 3.4: Layer spacing for different reference viewpoints: (a) too close to the object, (b) optimal placement, (c) too far from the object. (d) If the impostor width is smaller than the view-cell width, the optimal viewpoint is always nearest to the object.

increasing s. This can be shown using the derivation $c_r'(s)$ of Equation 3.2 which is

$$c_r'(s) = \frac{2reswt(n(t_0-t) - wt)}{(res(n(t_0+s) + ws) - 2w(t+s))^2}.$$

The denominator is always positive and the numerator is always negative, because $t > t_0$ holds. Consequently, c_r decreases monotonically with increasing s. Therefore, the best value for s for the right boundary line is 0. Note that placing the

reference viewpoint in front of the view cell (i.e., $s < 0$) is not possible, because distant impostor layers would appear magnified in the output image. Fortunately, if $s = 0$, Equation 3.4 always results in exactly the same value for c_h as Equation 3.2 does for c_r. This means, as long as the left boundary of every layer does not exceed the left boundary of a rectangular view cell, the optimal viewpoint is $s = 0$ as is shown in Figure 3.4 (d). This is also the best viewpoint considering the sampling of the impostor texture: if the viewer moves to the view-cell border nearest to the impostor (where $s = 0$), the texels in every impostor layer are equally-sized on the screen, thus making best use of texture memory. In contrast, if the reference viewpoint is placed further apart from the object (i.e., $s > 0$), distant layers are never viewed at full resolution which is a (minor) waste of memory. For all other cases (for the lower rectangular view-cell corner and a shaft-shaped view cell), no general optimum can be found. This is because the optimal reference viewpoint is different for each layer (as it depends on the layer distance t). However, changing the position of the reference viewpoint for different layers is not compatible with the artifact-free layer recording technique (see Section 3.3.2). However, in practice, a value for rectangular view cells of $s = \frac{m}{2}$ and for shaft-shaped view cells of $s = \frac{n}{2}$ was found to provide satisfying results.

Parallax Errors

The layering technique provides a reproduction of parallax movements within the represented scene parts. However, because every texel represents a certain depth range, parallax errors still occur within every layer. This error is quantified with the *parallax angle* α, which is the angle between the *true* 3D position of the point and its projection to the impostor [Scha98b]. The goal is to limit α for the whole impostor, typically to the angle of a pixel in the output image. In order to achieve this for layered impostors, the depth range for every layer has to be set up appropriately.

For the following considerations, we distinguish between two cases: first, we derive the depth ranges *between* two adjacent layers, i.e., where to place the border between two impostor layers. Afterwards, we determine the depth ranges of the outermost layers, i.e., in front of the first (nearest) layer and behind the last (most distant) layer.

Concerning the depth ranges between two layers, the one-texel layer spacing (see Section 3.3.1) already guarantees that the parallax errors that occur between adjacent impostor layers are always smaller than α. This is because the relative movement of a texel against the texel at the same position in the previous and following layer is limited to less than a texel. Given the fact that every texel is always visible with an angle smaller than α, no parallax error larger than a texel

can occur. A result of this consideration is that the choice where to place the depth border (i.e. the clipping plane) between two adjacent impostor layers is arbitrary.

However, to keep the introduced parallax errors as small as possible, parallax movements should be distributed *equally* between adjacent layers. This means that for the viewpoint were the maximum parallax angle occurs, it should be equally distributed to the closer and more distant layer, for every "view direction" from that point. Figure 3.5 (a) shows a configuration where the border is placed so that this is *not* achieved: the parallax angle that occurs for the closer layer is much larger than the one for the distant layer. In contrast, Figure 3.5 (b) shows a configuration where the border is placed so that this is achieved, independently from the view direction. Note that it is *not* desired to equally distribute the parallax angle for a particular texel position.

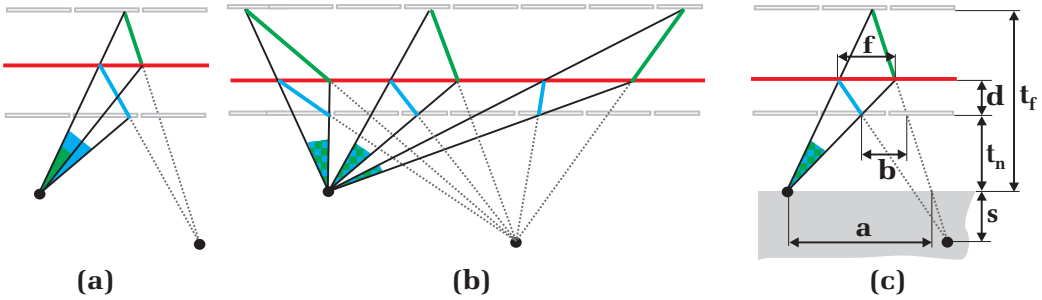

Figure 3.5: (a): different parallax angle in two adjacent layers. (b): equal parallax angle in two layers, independently from the view direction. (c): calculation scheme for optimally placing the border.

We assume that the largest parallax angle occurs if the viewer is located somewhere on the view cell border closest to the impostor (as Figure 3.5 (c) shows), because the impostor appears in maximum magnification for this configuration. Given two impostor layers with distances t_n and t_f to the view cell, the distance d from the near layer to the border between the layers is calculated using three pairs of similar triangles (see Figure 3.5 (c)):

$$\begin{aligned} \frac{t_f}{a} &= \frac{t_f - t_n - d}{f}, \\ \frac{t_n + d + s}{f} &= \frac{t_n + s}{b}, \\ \frac{d}{b} &= \frac{t_n + d}{a}. \end{aligned}$$

Solving the equation system for d (by eliminating a and b, which automatically

eliminates f) results in

$$d = \frac{-t_n^2 - t_n s + \sqrt{t_f t_n (t_f + s)(t_n + s)}}{t_n + t_f + s}. \quad (3.6)$$

Note that d only depends on t_n, t_f and s. This means that the parallax angle is equally distributed between the layers for *all* view directions of any view point on the view cell border, as is shown in Figure 3.5 (b).

The final task is the derivation of the depth range in front of the nearest and behind the farthest impostor layer. We assume a plane at a given distance, which is either the near border of the scene part to be represented for calculating the first impostor layer, or the last impostor layer for calculating to which distance that layer can represent the scene. The goal is to choose the depth range so that the parallax angle never exceeds α when seen from the view cell. Therefore, the setup for which the *largest parallax angle* occurs must be found. Figure 3.6 (left) shows

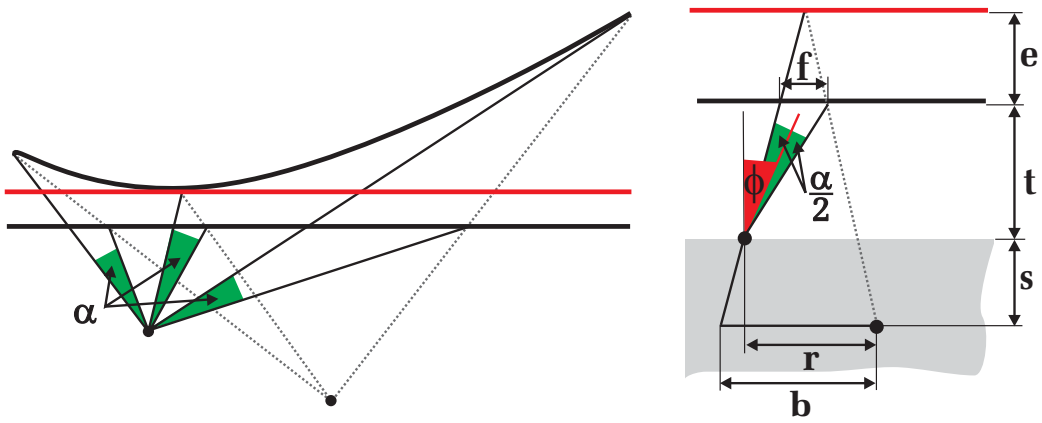

Figure 3.6: Left: points that are represented with an angle of α with respect to a layer form a curve. Right: setup for calculating the maximum allowable depth range.

for every point on some plane a corresponding point which is represented with an error angle of α when seen from a new viewpoint. All such "α-points" form a curve, as the figure shows. If the depth range is set to the minimum distance of all α-points to the plane (the red line in the figure), the parallax error will never exceed α for the whole depth range.

We again assume that the maximum parallax angle occurs for a viewpoint on the view cell border close to the impostor as Figure 3.6 (right) shows. The depth range e is described in dependence of an angle ϕ, which can be interpreted as the view direction from the new viewpoint. Given the plane at distance t to the view

cell and a viewpoint of r left to the reference viewpoint (see Figure 3.6 (right)), for an arbitrary ϕ we get the depth range e by similar triangles:

$$\begin{aligned} \frac{e}{f} &= \frac{e+t+s}{b}, \text{ where} \\ f &= t\tan\left(\phi+\frac{\alpha}{2}\right) - t\tan\left(\phi-\frac{\alpha}{2}\right), \\ b &= r + s\tan\left(\phi-\frac{\alpha}{2}\right). \end{aligned}$$

This results in

$$e = \frac{2tsin(\alpha)(t+s)}{r(\cos(2\phi)+\cos(\alpha)) + s(\sin(2\phi)-\sin(\alpha)) - 2t\sin(\alpha)}. \quad (3.7)$$

The minimum value for e is obtained by differentiation with respect to ϕ:

$$e'(\phi) = \frac{4t\sin(\alpha)(t+s)(r\sin(2\phi) - s\cos(2\phi)}{(r(\cos(2\phi)+\cos(\alpha)) + s(\sin(2\phi)-\sin(\alpha)) - 2t\sin(\alpha))^2},$$

and solving $e'(\phi) = 0$. This results in

$$\phi_{min} = \frac{1}{2}\arctan\left(\frac{s}{r}\right).$$

With the second derivative, it can be shown that this is a minimum for e, as Figure 3.6 (left) shows. Note that ϕ_{min} only depends on the position of the new viewpoint relative to the reference viewpoint, i.e., it is independent from the distance between the impostor and the view cell, and from the pixel angle α. In order to obtain the minimum depth range e_{min}, ϕ_{min} is inserted in Equation 3.7, resulting in

$$e_{min} = \frac{2t\sin(\alpha)(t+s)}{r\cos(\alpha) - (2t+s)\sin(\alpha) + \sqrt{r^2+s^2}}.$$

Note that e_{min} is decreasing with increasing r. This means that r has to be set to the maximum possible value in order to guarantee a parallax angle smaller than α for every viewpoint in view cell. This maximum value is obtained in the corner of the view cell, which is intuitive because this is the largest distance to the reference viewpoint.

An interesting characteristic is that starting from a certain distance, the parallax movements in the whole scene part farther away than this distance are smaller than α, so that the whole scene part can be represented by a single layer, regardless of its extent. This is the case if $f \geq b$ (see Figure 3.6 (right)), because the two

lines that enclose f (starting in the reference viewpoint and in the new viewpoint) do not meet each other.

All considerations up to this point work in a 2D plane. In contrast to texel movements, parallax movements cannot be treated independently for the x and y direction for the general 3D case. However, the viewpoint where the largest parallax effect occurs is still the view cell corner near the impostor, but now in 3D space. The calculations presented above can still be done in 2D, but in a plane which is defined by the reference viewpoint and the two diagonal view-cell corners near the impostor. Consequently, the value for r must be changed to mean the distance to a view cell corner in 3D space (see Figure 3.7). Using this setup, the calculations remain as described above.

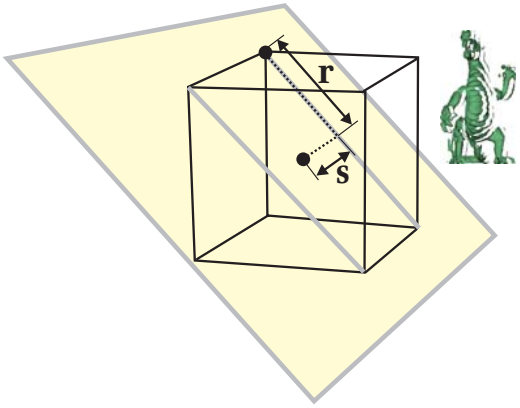

Figure 3.7: Parametrization for the 3D case for calculating the depth ranges concerning parallax angles.

3.3.2 A Rasterization Method for Guaranteed Layer Connectivity

Because every layer is recorded separately, scene parts hidden from the reference viewpoint are included in the representation (in contrast to Schaufler [Scha98b] who used only a single image). However, image cracks still appear. The problem is basically caused by the specification of polygon rasterization in current graphics hardware: a pixel is only drawn during polygon rasterization if its *center* is covered by the polygon. One effect of this definition is that surfaces viewed from an acute angle may not be rendered at all if they fall between two adjacent pixel centers. In this case, information is missing which can lead to large image gaps. Furthermore, small gaps appear between adjacent layers representing a

single primitive (i.e., polygon) because of the following fact: the intersection of the primitive with the clipping plane separating the two layers forms a *clipping edge*. The definition of rasterization entails that each texel of the clipping edge is rasterized *either* in one *or* in the other layer, as is shown in Figure 3.8, left. This instantly leads to gaps between adjacent layers if the viewpoint is moved.

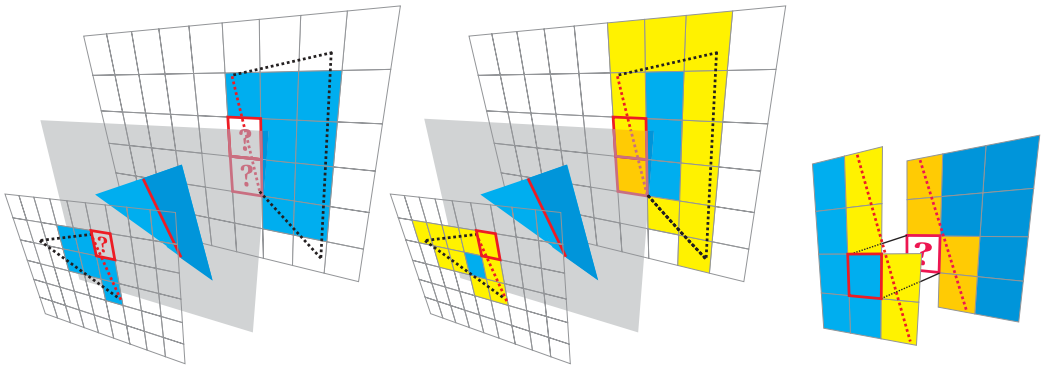

Figure 3.8: Left: the clipping edge of the polygon (drawn in red) is not represented in both adjacent layers. Middle: after separately drawing the outline of the clipped polygon (yellow texels), a common representation of the clipping edge is generated in both layers. Right: holes at each clipping edge step are filled manually to guarantee an artifact free representation.

The elimination of such image cracks is obtained in the following way: texels are drawn even if they are covered only partly by a primitive, especially if the texel center is not covered. This ensures that *all* scene parts are acquired (regardless whether they cover a texel center or not) and that *all* clipping edges are present in *both* involved layers.

The way to ensure this is to *manually* clip each polygon to the corresponding near and far layer border (Sutherland-Hodgeman clipping can be used, for example), and draw all polygon outlines explicitly, for example using the OpenGL edge primitive (see Figure 3.8, middle). The outlines have to be drawn in a predefined direction (e.g., from left to right) to ensure that identical texels are rasterized for the clipping edges in every layer. In order to ensure that all line endpoints are drawn as well, the polygon vertices are rasterized separately as points.

Although this removes most of the image cracks, sporadic ones might still appear if the viewer moves in *diagonal* direction within the view cell. This is caused by the rasterization of the clipping lines (an “eight-connected” line) as is shown in Figure 3.8 (right). Such cases can be manually identified by considering each 2x2 texel block in both layers: while the diagonal texels in one direction are

present in *both* layers (Figure 3.8 (right) shows the upper-left to lower-right case), the texels in the orthogonal direction are only present *either* in one *or* in the other layer. The gap can be closed by copying the texel color from the closer layer to the more distant layer (thus forming a “four-connected” line).

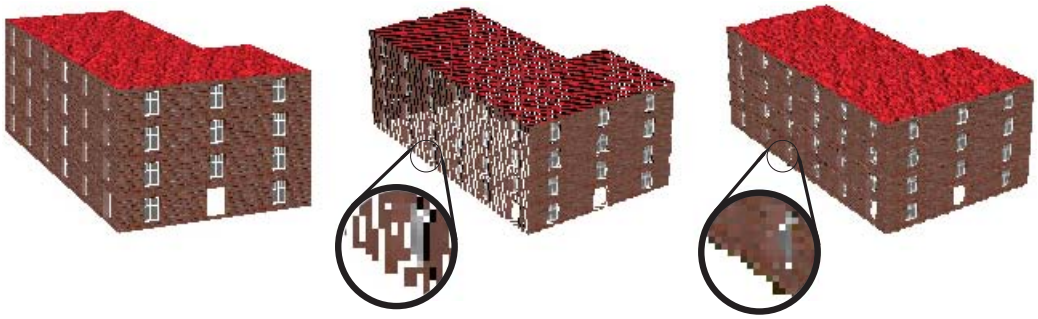

Figure 3.9: Left: original object seen from the reference viewpoint. Middle: impostor recorded using layers widely overlapping in depth. Note that image cracks are not avoided. Right: the same impostor generated using the new method with all image cracks eliminated.

The result is that for every object, all borders between adjacent layers are drawn using identical texel positions. In combination with the one-texel layer spacing (see Section 3.3.1) the representation shows no cracks or image gaps as can be observed in Figure 3.9 (right). Note that no previous approach (see Section 2.3.1) provides this desirable property. For instance, Schaufler [Scha98b] proposed to let the depth ranges of adjacent layers overlap. However, this gives no guarantee that image gaps do not occur, as Figure 3.9 (middle) shows. For this example, even a very large depth overlap of half a layer does not remedy the problem. Another approach by Meyer and Neyret [Meye98] tries to solve the problem in image space by estimating object contours in every layer and filling the inside of each such contour. However, because the algorithm does not exploit information about the original scene parts, it is unclear how well the result resembles the original scene.

3.3.3 Discussion on the Number of Layers

The number of impostor layers is important for the efficiency of the layered impostor technique since it strongly correlates to the required impostor memory as well as the geometric complexity of an impostor (see Section 3.7). Therefore, this section describes the basic factors that influence this value.

The number of required impostor layers depends on two main factors:

- The parallax movements within the scene part, which depend on the size of the view cell and the distance between the view cell and the represented scene part. Because parallax movements are “mimicked” by the impostor layers, the number of required layers increases proportionally with the amount of parallax movements.
- The size of an impostor texel, defined by the output image resolution and field of view. Because each layer is not allowed to move more than one texel (see Section 3.3.1), the number of layers grows with decreasing texel size.

In order to show the influence of parallax movements, the number of layers is calculated for the model of a dragon (about 20 m high and consisting of 108,500 polygons) for varying distances between the model and a cubic view cell with a sidelength of 10m. Figure 3.10 shows some views from a point outside the view cell in order to show the number of layers. Diagram 3.11 (left) shows the

Figure 3.10: Example for layered impostors for different object distances to the view cell. From left to right: 120, 60, 30, 15, 7, 1 layers generated for 42, 59, 96, 142, 209 and 563 m distance.

number of layers, obtained in each test for varying output resolutions (discussed further below). It can be seen that the number of layers falls hyperbolically with increasing distance between the model and the view cell. This was expected since parallax movements increase analogously. In a second test, the view-cell size was varied for three fixed distances. It can be observed that the number of layers grows roughly linearly with the view-cell size for all distances. A result from these tests is that the distance between the scene part and the view cell in combination with the view-cell size has a major impact on the efficiency of the resulting impostor: while for distant objects and/or small view-cell sizes only few layers are needed to cover large scene portions, near objects and/or large view-cell sizes result in very high number of layers. The distance has a greater impact than the view-cell size.

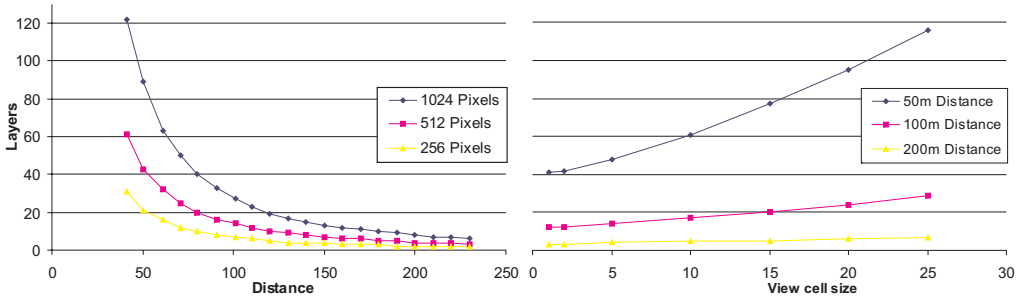

Figure 3.11: Number of layers required for the dragon model in dependence of the distance between the view cell and the model and for different output image resolutions (left), as well as for some fixed distances but varying view-cell sizes (right).

In the first test described above (Figure 3.11, left), the output resolution was varied between 256 and 1024 pixels. It can be seen that the number of layers increases roughly linearly with the number of pixels. This can also be derived from the Equations 3.2 to 3.4.

Note that all trends discussed here also hold for shaft-shaped view cells because of their similar parallax and texel movement characteristics.

3.4 Occlusion Culling Within the Impostor

In order to reduce the amount of texture memory required for an impostor, it is desirable to eliminate those texels from the impostor that never become visible because they are always occluded by texels of closer layers. A very fast and effective occlusion culling algorithm is presented here in the spirit of the extended projections method, introduced by Durand et al. [Dura00] (see Section 2.2.2).

The algorithm is based on the following consideration: the one-texel layer spacing (see Section 3.3.1) ensures that every texel exposes no more than the texel behind it when seen from within the view cell. A consequence is that every opaque 3x3 texel block completely occludes the center texel of the same 3x3 texel block in the following (more distant) layer.

Beginning with the layer closest to the view cell, the visibility information is propagated through the whole impostor. For this, the following operation is repeated for every pair of adjacent layers (see Figure 3.12, (a) shows the initial configuration for the 2D case): mark texels in the more distant layer as occluded if they are centered behind an opaque 3x3 texel block in the closer layer (see

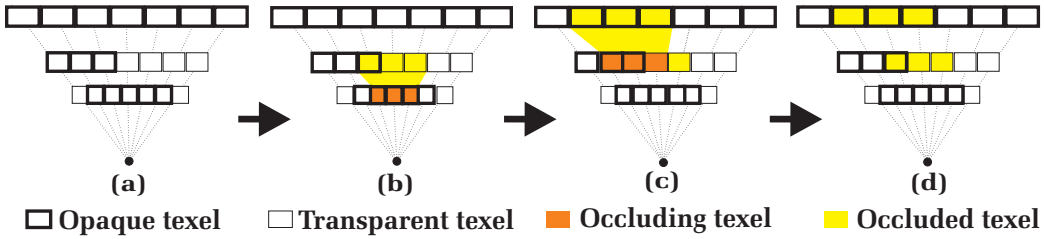

Figure 3.12: Occlusion culling algorithm for layered impostors. (a): initial configuration, (b): occlusions in the second layer caused by the first layer. (c): occlusions in the third layer caused by the first *and* second layer. (d): occluded texels are drawn in yellow.

Figures 3.12 (b) and (c)). Texels in the closer layer already marked as hidden in a previous iteration must be interpreted as *opaque* in this iteration (see Figure 3.12 (c)). Occluded texels are "deleted" by setting them to be transparent. Figure 3.12 (d) shows all occluded texels for the example, and Figure 3.13 shows two results obtained with this occlusion culling algorithm.

Figure 3.13: Examples for visibility culling within impostors. Culled texels are drawn in yellow. Left: house example. Note that even small occlusions visible in the windows and door frames are identified. Right: facade partly occluded by trees.

The occluded texels identified by the algorithm are covered by the *umbra* of the content in previous layers [Wonk00]. Furthermore, connected occluders and connected occluder umbrae are automatically *fused* during layer processing, as can also be observed in Figure 3.12. These desirable features ensure that many occluded parts in each layer are identified.

3.4.1 Improved Culling Addressing Penumbra Overlapping

The basic occlusion culling algorithm can be improved significantly at almost no additional computational costs. The approach described in this section is

conceptually related to the visibility-culling approach presented by Wonka et al. [Wonk00] (see Section 2.2.2) for the special case of *overlapping penumbrae*. Figure 3.14 (left) shows an example where overlapping penumbrae provide occlusion (red area). This type of occlusion is not detected by the basic occlusion-culling algorithm described in Section 3.4.

Figure 3.14: Left: example for occlusions caused by umbrae (yellow areas) and overlapping penumbrae (red area). Right: approximate occlusion (yellow areas) using the intersection of several umbrae obtained from a view-cell decomposition.

Wonka et al. approximate the occlusion provided by penumbra overlapping by decomposing the view cell and calculating the umbra separately for each part. The *intersection* of all such umbrae provides a solution that contains an approximation of the correct penumbra overlapping. The finer the view cell is partitioned, the more accurate is the approximation. Figure 3.14 (right) shows an example for only two view-cell parts, providing a considerably better solution.

This principle can also be applied to layered impostors using the boundary view positions. Figure 3.15 shows the algorithm for the 2D case. When the viewer moves from the reference viewpoint to the leftmost position, every opaque texel exposes the texel behind it. On the other hand, if the left neighbor is also opaque, the texel behind stays occluded by the neighbor. Using this fact, the occlusion culling algorithm described in Section 3.4 can be applied to obtain the umbra for the left view-cell part. The only difference is that texels are marked as occluded if they are behind opaque texels with a left opaque neighbor, instead of a left *and* a right opaque neighbor. This means that texels are marked as invisible if they are lying in the umbra of the left view cell part (i.e. they are completely invisible from the left view cell part), as is shown in Figure 3.15 (b). The texels lying in the umbra of the right view-cell part are obtained similarly to the left case, producing the result shown in Figure 3.15 (c). Finally, texels occluded for *both* the left and right view-cell parts are occluded for the whole view cell (Figure 3.15 (d)). The result also contains occlusions caused by overlapping penumbrae where the algorithm described in Section 3.4 fails (compare to 3.15 (a)).

The extension to the general 3D case is quite simple: instead of a left and

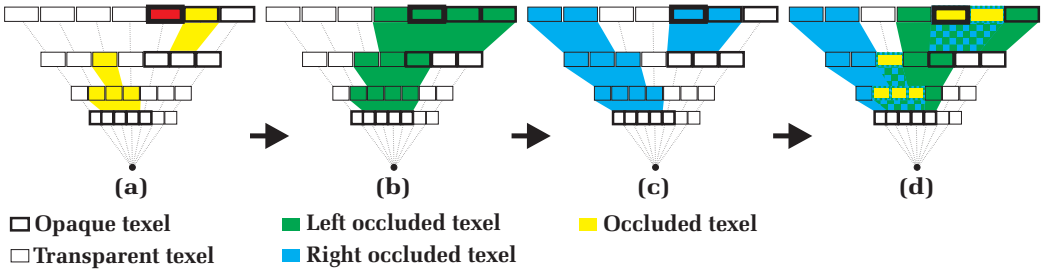

Figure 3.15: Improved occlusion culling accuracy with overlapping penumbrae. (a): result of the algorithm described in Section 3.4. (b): umbrae for the left view-cell part. (c): umbrae for the right view-cell part. (d): the intersection of the left and right umbrae defines the invisible texels. Note the additional occluded texel compared to (a).

right view-cell part, the cell is decomposed into four parts, each bounded by the reference viewpoint and a corner of the view cell. This results in a left-upper, right-upper, left-lower and right-lower part. For example, for the left-upper part, a texel is occluded if the closer layer contains an opaque texel at the same position as well as on the next higher, left, and higher-left position. Texel occlusions for the other parts are determined analogously. Finally, texels occluded in every part are marked to be occluded for the whole view cell.

The algorithm described above considers the umbrae intersections of only two (and for the 3D case four) view-cell parts. Extending the approach to more view-cell parts may improve the result in some situations. However, since the algorithm is based on the one-texel layer spacing, visibility information is only available for the reference viewpoint and the corners of the view cell. While it is theoretically possible to calculate umbra occlusions for arbitrary view-cell parts, it is not expected that the additional occlusion accuracy will justify the additional computational cost.

The occlusion-culling algorithms described above provide conservative results in the sense that all visible texels are classified as visible, but some invisible texels might no be found. However, because all types of occlusion are addressed to a certain amount, the algorithm provides sufficiently accurate results with only very little computational effort, as will also be shown in Section 3.7.3. The visibility culling process is very fast because it is performed in image space with exploiting the special layering scheme, in contrast to previous methods [Wimm01, Wils03] that use a sampling process from the view cell. On the other hand, this means that parts of the impostor that are occluded from scene parts not included in the impostor cannot be identified using the new method. However, if this is desired,

other visibility algorithms like the extended projection approach [Dura00] can be used to identify the invisible texels in the first layer. Afterwards, visibility culling for the following layers can be done using the method described above.

3.5 Memory-Efficient Layer Encoding

Every impostor has to be combined with geometry when it is placed into the scene. For layered impostors, this is done separately for each layer using quadrilaterals (in short, *quads*), with the content of the impostor layer applied as a texture. A straightforward approach would combine a single quad with every layer. Such a method leads to large memory requirements, as will be shown further below.

Every layer contains many transparent texels that take up unnecessary space. This can be exploited to significantly reduce the impostor memory requirements: for every layer, a set of quads is generated, each tightly enclosing some group of opaque texels, so that the whole set of quads contains all opaque texels in the layer. The goal is to tightly enclose the content of each layer while at the same time keeping the number of quads reasonably low for fast impostor display. While it would be possible to limit the number of quads and then minimize the texture memory, we focus on finding a good tradeoff between both demands.

In order to get a reasonable solution in a short time, a greedy *box-growing* algorithm is applied. First, the impostor texture is transformed to a regular grid, with each grid cell containing several texels. A grid cell containing at least one opaque texel is marked as *filled*, otherwise it is *empty*. A smaller grid size gives preference to fewer wasted texels, but also to a larger number of quads. Second, a filled cell is chosen as a *seed cell*. This region is then grown at steps of one cell in axis-parallel and diagonal directions, but is constrained to maintain a rectangular shape. The side with the best ratio between filled and empty cells (called *fill ratio*) is selected in every step. The growing process stops if either no side is adjacent to any filled cell, or the fill ratio of the whole region falls below a user-defined threshold. Smaller threshold values favor fewer wasted texels but also a larger number of quads. A value of 0.8 has been found to work well in practice.

If the region cannot be grown further, a quad is created, and the respective area in the impostor layer is applied as texture. The quad is then "deleted" from the impostor layer by setting all covered grid cells to empty, and the respective area in the impostor layer is set to transparent. Finally, a new seed cell is selected and the algorithm is repeated as above until no filled cell is left.

Figure 3.16 shows some examples obtained with this approach for different grid cell sizes for the dragon model (also see Section 3.3.3). Table 3.1 shows

statistics for the respective tests. The texture resolution of the impostor was

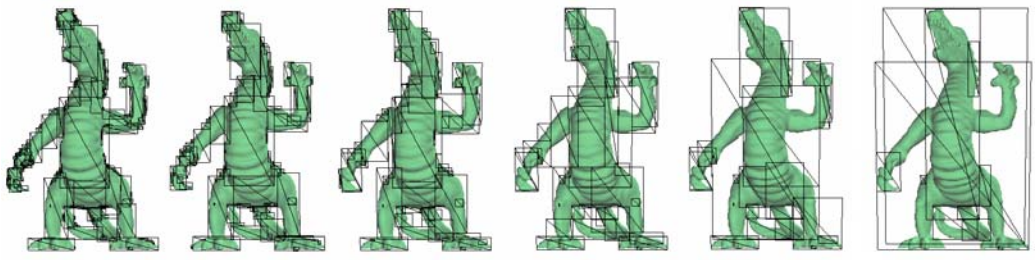

Figure 3.16: Different amounts of geometry for a layered impostor of the dragon model. From left to right: 340, 114, 71, 39, 25, 11 quads. The respective parameters used for the generation are given in Table 3.1.

Grid cell size (texels)	1	2	4	8	16	32	64	128	256	700
Number of quads	539	340	207	114	71	39	25	11	11	6
Geometry memory (kB)	43	27	17	9	6	3	2	1	1	1
Texture memory (kB)	414	423	447	490	555	719	996	1697	1833	2131
Overall memory (kB)	457	450	464	499	561	721	998	1698	1834	2132

Table 3.1: Statistics for layered impostors with different complex geometry for the dragon model.

397x677 pixels, and 6 layers were generated. Encoding each layer using a single quad would result in approximately 6.15 MB of texture memory. Note that in the last test in Table 3.1, a single tightly enclosing quad was generated for every layer, as was proposed by Schaufler [Scha98b]. This results in slightly more than 2 MB texture memory. As was expected, with decreasing grid cell size the required texture memory decreases too, but the number of quads increases. Note that a very high number of quads might in turn increase the overall memory needed for the impostor due to the memory required for the quad vertices, as is the case for a grid cell size of one texel in Table 3.1. Also note that general statements about the optimal relation between quad number and texture memory are not possible: if the number of impostor quads is critical for the rendering acceleration, a small number is desired at the cost of more necessary texture memory. Otherwise, a high number of quads can be used for reduced impostor memory.

In theoretical research, similar problems have been studied in the context of multi dimensional data partitioning [Muth99] and in computer graphics, for instance in the context of accelerated volume rendering [Li03]. Another option would be to use *hierarchical split and merge* algorithms [Horo76]. However, practical tests [Preu04] have shown that such approaches are not well suited for

the application at hand because they are slower and generate more quads for the same desired texture memory compared to the approach presented above. Better results may be obtained using more involved algorithms [Beck91] mainly at the expense of much longer preprocessing times.

3.6 Efficient Graphics-Hardware Treatment Using Texture Atlases

The number of impostor polygons generated with the algorithm described in Section 3.5 is usually quite large. Storing every texture separately is very inefficient because of the general overhead involved in handling a large number of textures. The main point is that texture switches, like most state changes, are a costly operation on current graphics hardware. Furthermore, older graphics hardware only supports textures with resolutions of powers of two. Enlarging every single texture would be very memory inefficient.

These observations motivate the use of *texture atlases* which comprise many small textures [Mail93, Cign98]. Decisions must taken as to the number and sizes of the atlases, the atlas each texture should be placed in and the position within the atlas. Of course, every atlas should be "well filled" to increase texture memory as little as possible. A fast greedy algorithm is presented here that solves these three problems in reasonable time. The algorithm is aimed at generating only few atlases which at the same time achieve a good coverage with opaque texels and can be sent directly to graphics hardware.

First, all textures are rotated so that the larger side is oriented vertically. Then they are sorted into a list by decreasing height and—for equal heights—by decreasing width. The height of a new atlas is determined using the height of the first texture in the list, enlarged to the next power of two. The width of the atlas is calculated by the summed area of all textures in the list, divided by the atlas height. This value is then also enlarged to the next power of two.

The problem of placing textures into the new atlas is equivalent to the *2D rectangle packing problem*, a special version of the classic NP-complete *bin packing problem* [Bake78]. In literature, many placement strategies for 2D rectangle packing [Lodi99] have been proposed. However, the performance of the strategy depends on the actual data set [Preu04]. The approach used here is similar to the well known computer game "Tetris": beginning with the first texture in the sorted list, each texture is placed in the atlas at the lowest possible vertical position (if it fits). After being placed, the texture is removed from the list.

When all textures are tested and possibly placed, the algorithm resumes with generating the next atlas. This is repeated until no texture remains.

Figure 3.17 shows the texture memory overhead produced by the packing algorithm for several larger impostor databases for the Vienna model and the UNC Power Plant model. It can be observed that the overhead is reduced with increas-

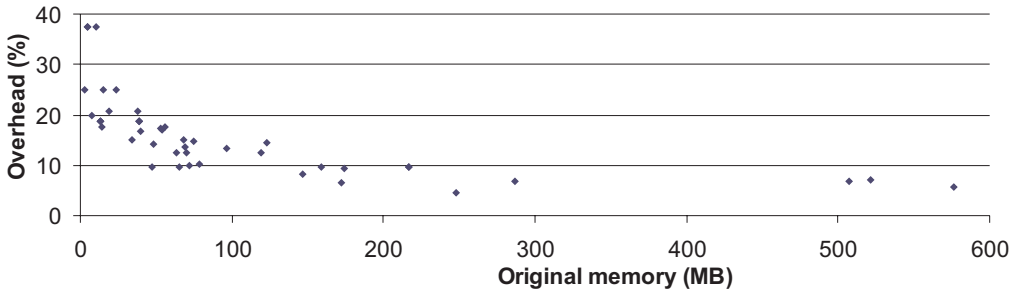

Figure 3.17: Memory overhead produced by the texture atlas packing algorithm.

ing size of the data set. This is because texture atlases with sizes of powers of two were generated, which induces more relative waste of memory for smaller data sets. However, in most cases the overhead is between 5 and 25 percent, which should be acceptable for most applications.

Note also that for storing the atlases on harddisk, compression techniques such as PNG or JPEG compression can be used to further reduce the amount of required memory. Furthermore, textures can also be compressed automatically in modern graphics hardware. However, for lossy compression methods it must be ensured that the alpha channel is not changed, as this would result in undesirable artifacts.

3.7 Results

The algorithms to generate the layered impostor representation presented in this chapter have been implemented in C++ using the OpenGL API. All statistics were obtained on a PC with an Intel Pentium4 3.2 GHz processor, 1 GB of main memory and an NVIDIA GeForce Quadro FX 3000 graphics board.

The image quality, memory requirements and generation time are the most interesting parameters and are discussed in the following sections. The rendering acceleration provided by layered impostors depends on the actual rendering bottleneck (see Section 2.2) and is discussed in detail in Chapter 5.

3.7.1 Image Quality

One advantage of the layered impostor technique presented in this thesis is that it *guarantees* that no image artifacts like image gaps caused by disocclusions occur. Figure 3.18 shows an example for the dragon model. Note that small image gaps become visible only outside the view cell (leftmost and rightmost image), while there are no problems for viewpoints in the view cell (e.g., behind the claw of the dragon). Furthermore, all parallax effects are always reproduced sufficiently with-

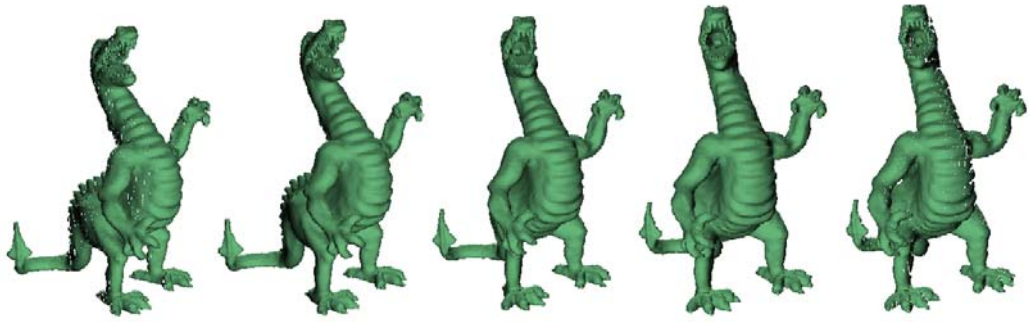

Figure 3.18: Within the view cell, no image gaps or cracks occur and parallax movements are correctly represented. The leftmost an rightmost image show views with a viewpoint slightly outside the view cell, where the guarantee does not hold anymore.

out having to rely on the input model structure. Previous impostor techniques (see Section 2.3.1) try to achieve these desirable features with massive oversampling of the input model or with displaying multiple impostors for an output image. However, drawbacks of those methods are high memory requirements and long preprocessing times, and/or a much less efficient impostor display while still not guaranteeing the absence of image artifacts.

Since the impostor layers are recorded using OpenGL rendering, the reconstruction quality is limited to the sampling provided by OpenGL. Also note that all layers are recorded at the screen resolution, independent of their distance to the view cell. This reduces temporal aliasing (i.e., flickering). While the impostor quality can easily be improved, for instance by applying oversampling during the layer recording process, a representation with higher quality than the rest of the (normally rendered) scene would be noticeable.

The explicit drawing of polygon outlines for avoiding image gaps, as was described in Section 3.3.2 results in a noticeably "bloated" representation for some objects. Figure 3.19 shows an example of a tree: the branches in the impostor (bottom image row) are more apparent than in the original rendered image (top

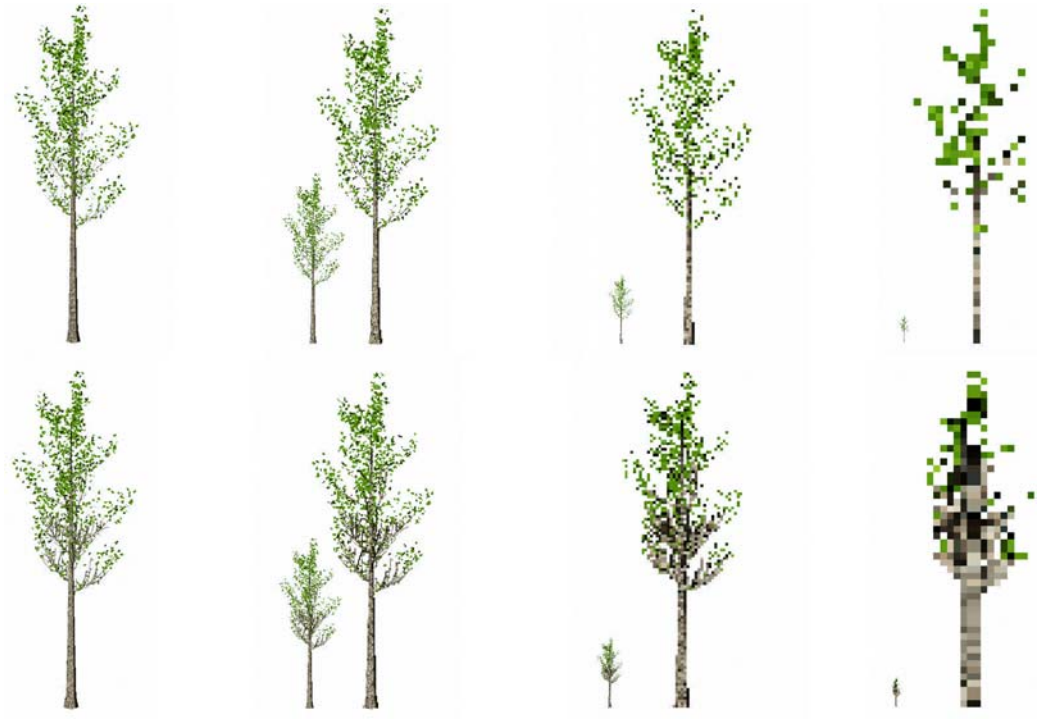

Figure 3.19: "Bloated" impostor representation for a tree: the top row shows the original rendered tree, the bottom row the respective impostor representation.

image row). This can be avoided if the "fuzzy" structure of a scene part is known (as is the case for trees and other ecological objects). Then drawing the outline can be skipped, because small gaps will be masked by the fuzzy appearance of the scene parts.

3.7.2 Memory Requirements

Memory requirements are discussed for some setups that are of practical interest. Although no general formula can be given how memory changes for different setups, general trends can be shown.

Figure 3.20 shows the memory requirements for the dragon model for a view cell placed in different distances to the model. It can be observed that the memory falls more than $\frac{1}{d^2}$ with increasing distance d, because the number of pixels on screen covered by the object already behaves like $\frac{1}{d^2}$ with increasing distance, and the number of impostor layers also decreases due to reduced parallax movements.

A second observation is that the required memory grows between 3 and 4

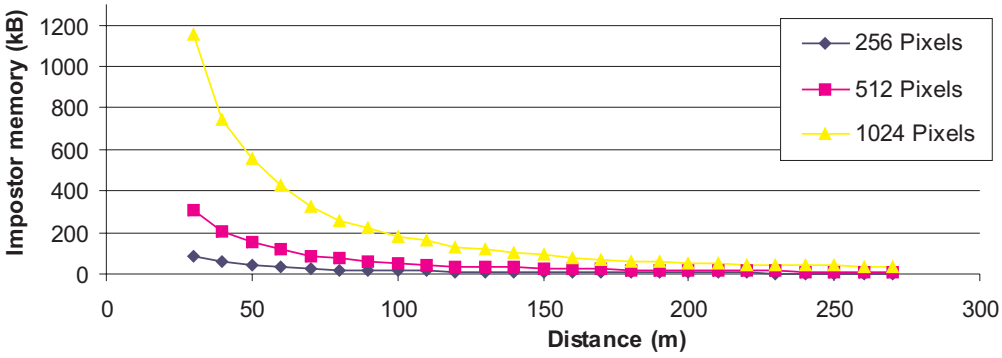

Figure 3.20: Memory requirements for layered impostors for different output resolutions in dependence of the distance between view cell and object.

times when doubling the output image resolution. On the one hand, the number of pixels covered by the object grows quadratically with the output resolution, and the number of necessary layers also increases due to stronger parallax movements. On the other hand, the algorithm that generates the geometry for every layer (see Section 3.1) provides tighter fits of the content of each layer for higher resolutions, since the grid cell size it operates on is fixed.

As was stated before, it is desirable for the impostor technique to support large view cells in order to reduce the overall impostor memory for a scene. A comparison between shaft-shaped view cells and rectangular view cells is presented here: using the test setup described above, a rectangular view cell placed 80 m away from the model was extruded, thus forming a shaft that encloses all rectangular view cells placed farther away. For this shaft, an impostor was generated for 512x512 pixels output image resolution, resulting in 390 kB of impostor memory. On the other hand, the accumulated memory of all rectangular view cells enclosed by the shaft is more than 536 kB, still providing a smaller view space than one shaft. This shows the general advantage of using shaft-shaped view cells, since they provide a good ratio between supported view space and memory requirements.

Note that when using a simple planar impostor technique [Scha95b], for an impostor that is valid for the shaft described above, 9 planar impostors (3 each in vertical and horizonal direction) are necessary resulting in 1250.5 kB of required memory. This shows how memory requirements can be greatly reduced with the layering technique. Note that the advantage even grows if the apex angle of the supported shaft is increased: while for the layered impostors, only the number of needed layers increases, many additional planar impostors will have to be gener-

ated, thus significantly increasing memory.

3.7.3 Generation Time

The generation time of a layered impostor depends on three main factors:

- the time needed for rendering every layer,
- the time needed for reading back the rendered textures from graphics hardware into main memory, and
- postprocessing the layer, including occlusion culling and impostor geometry generation.

The time needed to generate an impostor grows linearly with the number of layers, because all operations have to be calculated per layer.

On our test machine, examples for the rendering time needed to render a layer range from 2.36 ms for a single building, over 12.6 ms for the dragon model, and up to 41 ms for the whole Vienna model and 164 ms for the UNC Power Plant model. Note that these times include the additional operations needed for an artifact free representation (see Section 3.3.2). Note also that our implementation leaves space for further optimization. For instance, the primitives can be sorted according to the layers they cover, and for each layer, only the contained primitives could be sent to graphics hardware.

The time needed to read back the frame buffer and the time for the layer processing step mainly depend on the size of the rendered impostor texture. The time to read back the frame buffer varies for the test system between 0.1 ms for an impostor covering 32x32 pixels on screen, 3.2 ms for 256x256 pixels and 50 ms for a very large area of 1024x1024 pixels. The time needed for postprocessing every layer takes between one third and the whole of the time needed for reading back the frame buffer, independently of the actual model and texture size.

Given the fact that impostors are in practice used preferably for small distant scene parts that are very complex, it can be said that the impostor generation time mainly depends on the rendering of the layers, but not on the generation steps applied afterwards. Furthermore, the overall generation time can be called fairly low compared to other methods. Especially higher quality impostors that provide comparable image quality and memory requirements like for instance textured depth meshes (see Section 2.3.1) require substantial postprocessing time for generating the impostor geometry.

3.8 Applications

The layered impostor technique presented in this thesis can be used for rendering acceleration in several ways. Aside from the use for rectangular or shaft-shaped view cells, layered impostors can also be used in architectural models as portal impostors. Additional applications for the technique include *layered environment map impostors* and view-independent impostors.

3.8.1 Layered Environment Map Impostors

In this application, the impostor layers are arranged in a concentric way around a cubic view cell, forming "cubic" environment maps. This means that the impostors are used to represent the far field (see Section 2.3.2). Figure 3.21 shows an example for the Vienna model, where impostors were placed in four orthogonal directions parallel to the ground plane. Note especially that most of the invisible scene parts within the impostor have been automatically eliminated by the occlusion culling algorithm.

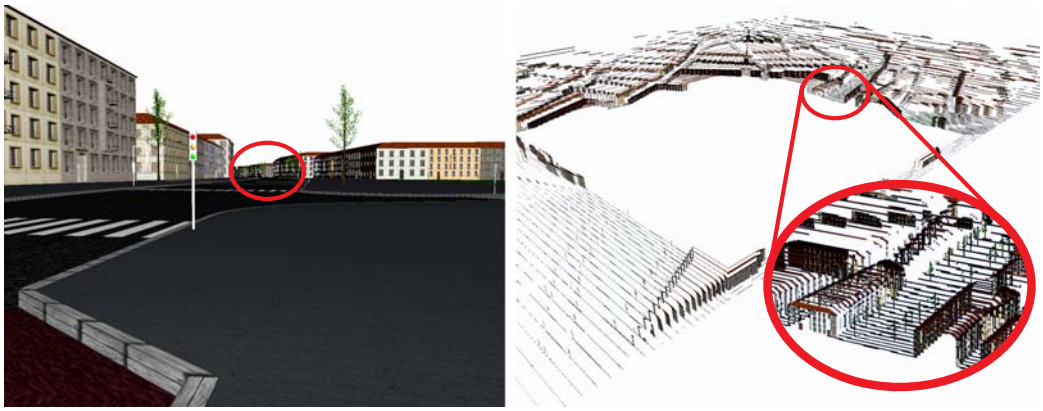

Figure 3.21: Layered environment map impostor.

3.8.2 Layered View-Independent Impostors

The main advantage of view-independent impostors is that they cast and receive shadows, as is necessary in current computer games. Layered view-independent impostors consist of multiple impostor representations for the same

object, so that one representation is used for each possible view direction (similar to Jakulin [Jaku00]). The layering technique reduces the number of required representations compared to planar impostors [Alia99b]. Figure 3.22 shows an example where the impostor is generated only from three orthogonal view directions. The distance from the first (nearest) layer to the second one is calculated using the equations presented in Section 3.3.1 and all layers are uniformly spaced based on this distance. This allows using the impostor also for the opposite view direction it was generated for.

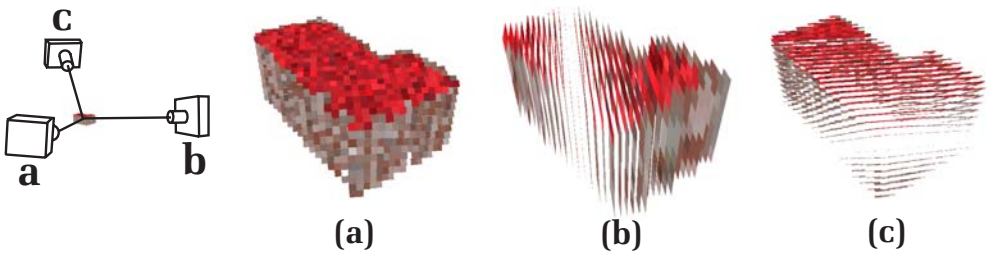

Figure 3.22: View-independent layered impostor recorded from three orthogonal directions.

Although conceptually possible, the high number of layers required for each view direction makes this approach less efficient compared to the billboard cloud approach [Deco03]. For instance, the example in Figure 3.22 results in 214 impostor polygons and 90kB of impostor texture memory. This is quite a lot, given the fact that the house covers only 30x30 pixels on the screen. The only advantages over the billboard cloud technique are much shorter generation times and the complete elimination of image artifacts. However, both of these problems are expected to be overcome during further investigation of the billboard cloud technique.

3.9 Discussion

In this chapter, a new layered impostor technique was presented. It combines the following desirable features:

- **Artifact-free representation:** One main advantage compared to previous approaches is that the new technique achieves the elimination of all the artifacts usually associated with impostors, such as disocclusion artifacts. This is achieved by a special layer placement in combination with a special

layer recording method that ensures that every scene part is present in each layer it covers.

- **Low memory requirements:** The layered impostor technique needs very few memory because invisible scene parts are removed from the layers and (even more importantly) transparent regions in every layer are excluded from the final representation.

- **Fast generation:** The algorithms for excluding invisible scene parts and generating the impostor geometry operate in image space, which is very fast with current CPUs. Therefore, the whole impostor generation process is mainly defined by the rendering of the scene part to be represented into the layers.

- **Fast display:** The method naturally supports conventional graphics hardware for fast impostor display. Since no complex online calculations are necessary, optimal runtime efficiency is achieved. The use of texture atlases reduces the number of texture switches which further accelerates the impostor display.

These features make the new technique very useful for applications such as computer games where high-quality impostors were not often used before. On the other hand, layered impostors for scene parts near the view cell are less efficient due to the prohibitively high number of required layers. Chapter 4 will present a method that overcomes this drawback.

Chapter 4

Textured Depth Meshes for Near Scene Parts

4.1 Introduction

The layered impostor technique presented in Chapter 3 offers numerous desirable features. On the other hand, care has to be taken about the number of required impostor layers because the preprocessing time and the memory requirements grow with this value. This reduces the efficiency of the technique especially for nearby objects, where it requires a very high number of impostor layers.

In order to overcome this drawback, this chapter extends the technique by introducing a method for connecting the layers, thus forming a textured depth mesh (TDM) [Alia99a, Sill97]. The goal is to preserve the desirable features of the layered impostor technique, like the applicability for arbitrary geometry, an artifact-free representation and fast rendering. Simultaneously, the required texture memory and impostor complexity should be significantly reduced, so that impostors for objects near the view cell can be generated and displayed efficiently.

4.2 Overview

For the TDM generation process, the scene part to be represented is layered in the same way as was described in Chapter 3. For a better illustration, all texel layers together are interpreted as a *voxel grid*. While for standard layered impostors, every voxel layer is encoded separately as a set of quads, for TDMs, successive voxel layers are *connected* and the resulting mesh simplified. A result is that the complexity of the final representation no longer relies on the number of layers.

Given a scene part and a view cell, as well as the output display resolution and field of view, the algorithm for generating the TDMs consists of the following steps (see Figure 4.1):

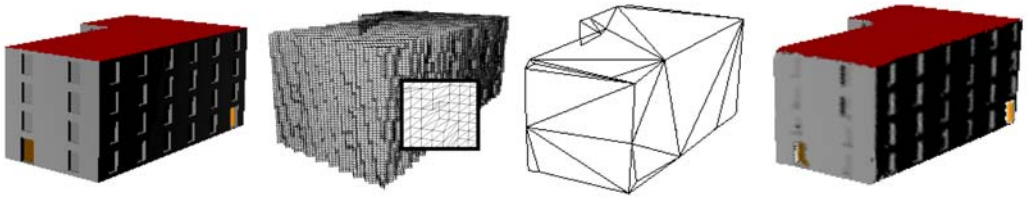

Figure 4.1: Overview of our method for constructing textured depth meshes (from left to right): the original object (1880 polygons), the initial mesh constructed from the voxel grid (61584 polygons), the simplified mesh (35 polygons), final textured mesh. Note that all images are close-up views.

1. The layer spacing is calculated and the voxel grid is generated by rendering the scene into layers using the steps described in Chapter 3. Afterwards, the recorded grid is transformed so that all voxels are sized equally (see Section 4.3).

2. An initial complex yet very regular mesh that covers the whole voxel grid is created (see Section 4.4).

3. The complex mesh is simplified (see Section 4.5). A special error bound guarantees that the shape of the simplified mesh does not change more than a user-defined amount in image space.

4. The pictorial information (i.e., the textures) for the simplified mesh is extracted from the voxel grid (see Section 4.6).

4.3 Voxel Grid Generation

The voxel grid is generated exactly as described for the layered impostors in Chapter 3, including the layer placement calculation. The difference for TDMs is that every texel is interpreted as a voxel (defined by the texel boundaries and the depth range of the layer), and the color of the texel defines the color of its voxel. Figure 4.2 (left) shows the arrangement of a voxel grid for a particular view cell.

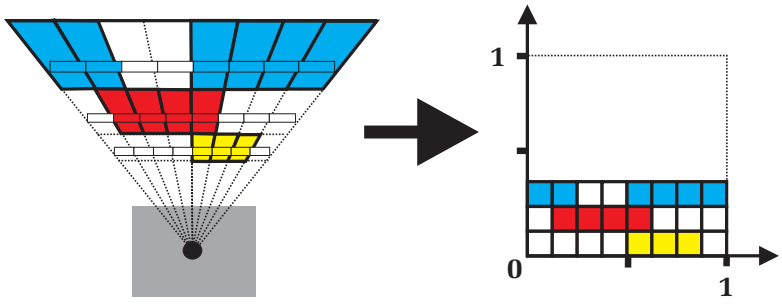

Figure 4.2: The voxel grid (left) is transformed into a normalized, uniform voxel grid (right).

The voxel grid layers are non-uniformly spaced according to texel movements and parallax error as was described in Section 3.3. For further treatment, the voxel grid is transformed into a normalized, uniform voxel grid, i.e., a grid with cubic voxels and a maximum total sidelength of 1 (see Figure 4.2 (right)). This is obtained by defining the voxel position in the grid using the 2D texel coordinates and the layer number.

The reason for this transformation is twofold: on the one hand, it allows storing all voxels in a compact hash table-based memory structure for efficient lookups. Furthermore, this organization is important for the simplification and texture generation process, since it influences the error metric used during the mesh-simplification step later on. More specifically, it causes the projected errors for views within the view cell to be equivalent to the simple geometric distance between vertices.

4.4 Initial Mesh Generation

The first step for creating the TDM representation is to produce an intermediate mesh that covers all voxels in the voxel grid. The idea is to connect all neighboring voxels using a single mesh. The advantage of having a connected mesh is that it can be simplified so that the complexity of the final mesh is independent from the number of layers. While the intermediate mesh can be highly complex, it is easy to create and of regular structure, and therefore highly amenable to simplification (see Section 4.5). Note that for disconnected voxel sets, multiple meshes have to created. Here, an important challenge is to create not more meshes than strictly needed.

The basic idea is to apply a meshing operator after rendering every voxel layer.

This operator connects all adjacent voxels in the current layer and closes connections to the previous (less distant) one. The detection of these connections is guided by knowledge about the one-texel layer spacing between adjacent layers (see Section 3.3).

For the sake of clarity, the process is described first for a single voxel layer. An obvious approach would be to simply connect the centers of adjacent voxels as shown in Figure 4.3 (left). However, this leaves voxels with only one neighbor unaccounted for. To overcome this problem, four vertices are used for every voxel, thus allowing to connect all adjacent voxels (Figure 4.3 right).

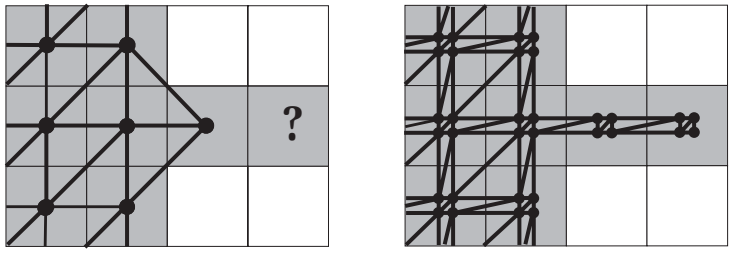

Figure 4.3: Covering voxels with only one neighbor is not possible using a single vertex (left). At least two connecting vertices are necessary (right).

This configuration creates more polygons, but allows covering all connected voxels with a mesh. The illustrations in Figure 4.4 show the *previous* (closer) and the *current* (more distant) layer as seen from the view cell. Transparent voxels are shown as white, while gray ones are opaque. In order to generate a mesh that only consists of polygons which are visible from the view cell, 3 rule sets together with simple meshing operations (Figure 4.4 (a)-(c)) are applied for every voxel in the grid.

The first rule set (Figure 4.4 (a)) establishes connections in the current layer if any of the involved voxels in the previous layer is transparent. From left to right and top to bottom, the current voxel is first connected to itself, then to the right, to the bottom, and to all 3 neighboring voxels, if the voxels shown in gray are opaque in the current layer, but at least one of them is transparent in the previous layer.

With the second rule set (Figure 4.4 (b)), the current voxel is *sealed off* towards each of the 4 sides if both the voxel itself and the corresponding voxel in the previous layer are opaque. In addition, for a particular side, the voxel is only sealed off if at least one of the voxels in the previous layer adjacent to this side is transparent, as the figure shows.

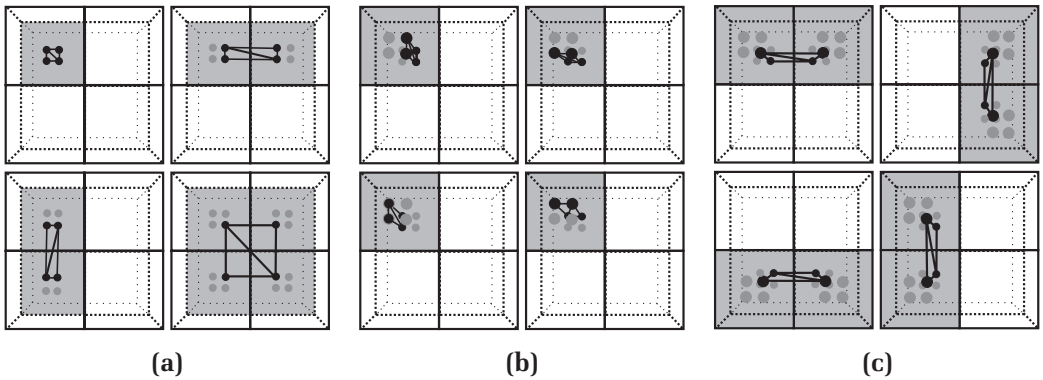

Figure 4.4: 12 initial mesh generation rules. (a): in the current layer, (b): for sealing off each voxel towards each side. (c): for sealing off voxel pairs towards each side.

The third rule set connects 2x2 voxel blocks (2 in the current, 2 in the previous layer) of opaque voxels if any of the 2 facing voxels in the previous layer is transparent, as shown in Figure 4.4 (c).

These rules account for all cases and form all connections that are valid and needed in order to cover neighboring voxels with a mesh.

Note that the total number of polygons generated by this algorithm could be reduced by a factor of 2 to 3 by adaptively introducing extra vertices only where needed. However, this would also complicate the mesh-generation process and should be considered only in memory-constrained situations. Also note that the number of polygons generated is output sensitive due to the removal of invisible parts of the scene, and therefore the mesh is not expected to grow beyond a size that can easily be handled in main memory.

In order to record and maintain the mesh connectivity, when a new polygon is created by one of the steps described above, it is tested whether the polygon can be connected directly to any of the existing meshes. If only one candidate mesh is found, the polygon is added and the algorithm proceeds. If it can be connected to more than one mesh, however, all candidate meshes are merged using this new polygon as a connection. If the polygon cannot be connected to any existing mesh, it is used to start a new mesh.

The method described in this section does not produce more individual meshes than strictly necessary due to occlusions and disjoint parts in the voxel grid. When rendered, the set of resulting meshes covers all the voxels from the original representation. Note that while at the borders, the resulting meshes may be slightly smaller than the set of voxels they cover, this is actually correct because due to rasterization, the voxels are usually slightly larger than the objects they represent.

4.5 Mesh Simplification

The meshes produced by the algorithm presented in the previous section provide a good starting point for the application of a mesh simplification algorithm. The aim is to produce a set of meshes with significantly lower complexity, but which still cover all of the voxels touched by the initial meshes. Although many of the simplification approaches presented in recent years [E. P97] could be applied, the specific structure of the problem suggests the development of a new method.

The approach presented here is conceptually related to simplification envelopes [Cohe96] by its use of a surrounding volume bounding the simplification process. Individual simplifications are based on edge contractions (using the *QSlim* simplification software developed by Michael Garland [Garl97]). The algorithm proceeds by inserting all edges into a priority queue sorted by edge length, and iteratively removing the shortest edge from the queue. This edge is checked against the voxel grid (see below), and if it can be contracted, all edges affected by the contraction are resorted or reinserted into the queue according to their new length. If a contraction fails, the edge will only be reconsidered if it is reinserted into the queue when a neighboring edge is contracted. The process continues removing the shortest edge from the priority queue until the queue is empty.

Always choosing the shortest edge for contraction favors a uniform simplification and prescribes a useful order even for finely tesselated coplanar regions. Using other error metrics for ordering the priority queue would fail in such cases and lead to slithery triangles because the simplification order would be more or less random. Also note that an edge connecting two layers has the same length as an edge between two pixel centers in the warped voxel grid the algorithm is operating in (see Section 4.3). This is exactly what we are looking for, because the length in the warped grid corresponds approximately to the perceived length after projection to the screen. In this way, edges that appear to have the same length on screen are treated equally, regardless of their length in world space. This practically results in a view-dependent simplification [Lueb97].

A crucial part of the algorithm is the test whether an edge can be contracted. This test basically checks whether the resulting mesh still covers all relevant voxels: at first, while creating the initial mesh, each polygon stores identifiers for all voxels it covers (this is trivial to compute due to the mesh structure). Now, assuming a contraction between the vertices v_1 and v_2 should be performed so that v_1 remains, the necessary steps are:

1. Collect the voxels covered by all polygons that include v_2 by simply accumulating the voxels stored for every involved polygon.

2. Contract v_2 to v_1 and check if the voxel set computed in the previous step is also covered by the resulting polygons (using a polygon-box intersection test). If not all voxels are covered, undo the contraction, and either exchange v_1 and v_2 and repeat step 1, or, if v_1 and v_2 were just exchanged, exit. In the latter case, the contraction failed.

   The size of the box representing a voxel for testing against a polygon can be used as a simple criterion to trade off rendering speed against accuracy: increasing the size of a voxel results in more simplified but less accurate meshes. Here the warped voxel grid calculated in Section 4.3 plays an important role again, as it provides a one-to-one correspondence of screen-space pixel error to the factor used to increase voxels. In practice a factor of about 1.5 to 2 has proven to result in dramatically simplified (see Section 4.8) but still fairly accurate meshes.

3. If all relevant voxels are covered by the new polygons, store for each new polygon the set of voxels that are covered by it. This information has already been computed as a by-product of step 2.

See Figure 4.5 for an example in 3D space for a forbidden (a) and an allowed (b) contraction.

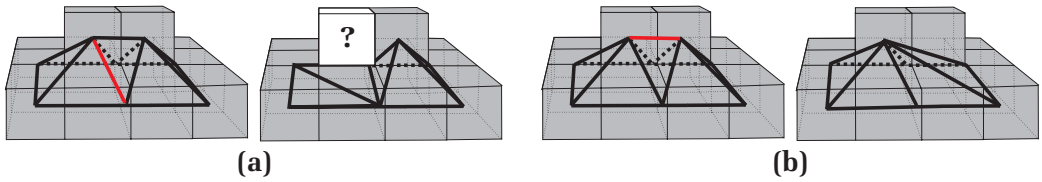

Figure 4.5: An example for a forbidden (a) and an allowed (b) contraction in 3D space.

While this test is adequate for the *interior* of the mesh, special treatment is required in order to maintain the shape of the mesh *boundary* for avoiding image cracks. Therefore, the mesh boundary must neither be shrunken nor enlarged.

Contracting a boundary vertex towards the interior of the mesh might be allowed when the resulting mesh still covers all relevant voxels—however, this will create a crack in the representation because the mesh has been shrunken. See Figure 4.6 (middle) for an example. Therefore, contractions involving boundary vertices are only allowed if the contraction is applied *towards* a boundary vertex (Figure 4.6 (right)).

Considering mesh enlargement, contracting a boundary edge may incorrectly cause empty voxels to be covered as is shown in Figure 4.7. This can be prevented

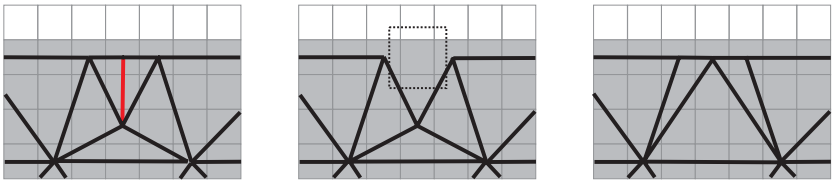

Figure 4.6: Left: the initial configuration with the red edge to be contracted. Middle: this contraction away from a boundary vertex is forbidden because it would create a gap, although all (enlarged) voxels are still covered. Right: contracting towards the boundary vertex produces a correct mesh.

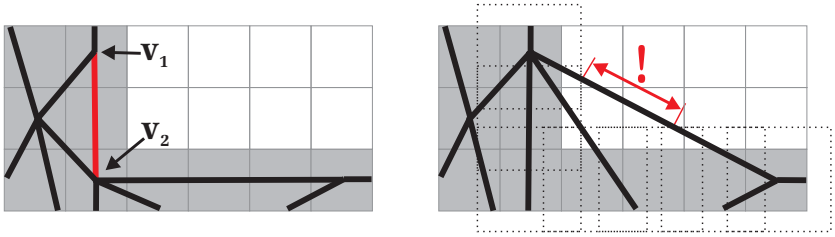

Figure 4.7: A contraction is not allowed if an empty edge interval would result. The dotted boxes indicate the enlarged voxels for intersection testing with the polygon.

by calculating the intersection intervals of the new boundary edge with all relevant voxels (using an edge-cube intersection test) and test whether the union of these intervals completely covers the edge. If a non-covered edge interval remains, the contraction is rejected.

The result of the mesh simplification step are a much coarser meshes which still cover all voxels that were covered by the initial meshes. The final step for generating the TDM is to apply textures for the simplified meshes.

4.6 TDM Texture Generation

Color information is assigned to the polygons of the simplified representation via texture maps. This can be done efficiently by sampling the voxel grid for each polygon separately: first, generate a texture so that the texel size approximately corresponds to the side length of a voxel, and the extents of the texture are slightly larger than the polygon in order to ensure that all texels needed for rasterization are present even when bilinear texture filtering is used. Second, this polygon texture is sampled in the voxel grid by averaging for every texel center the color of the n closest voxels, weighted according to their distance. For texels lying

completely inside the polygon, it is sufficient to consider only the voxels stored with the polygon in the mesh data structure. For texels at the border of or outside the polygon, the whole voxel grid has to be searched. However, this last step can be done efficiently using the hash table-based data structure for the voxel grid. Applying oversampling (i.e., setting $n > 1$) results in smoother, but also somewhat more blurry textures.

When all textures are generated, they can be stored using a *texture atlas* (see Section 3.6) in order to minimize texture switching overhead during rendering and to make efficient use of texture memory.

4.7 Placement of the TDM into the Scene

The final step in generating a TDM is to place it into the scene. This is done by reprojecting the vertices of the simplified and texture-mapped mesh to their correct positions. Since no new vertices were introduced during the simplification, for each vertex its exact position within the voxel grid and therefore within the scene is already known.

As previously noted by Darsa et al. [Dars97], perspective correction for the textures of the TDM should be disabled during rasterization, since perspective is already recorded in the texture.

4.8 Results

The algorithms for generating the TDM representation presented in this chapter have been implemented in C++ using the OpenGL API. All statistics were obtained on a PC with an Intel Pentium4 3.2 GHz processor, 1 GB of main memory and an NVIDIA GeForce Quadro FX 3000 graphics board.

In order to discuss the main characteristics of the method, the model of a single building was used and TDMs were generated for a large cubic view cell with 100 m sidelength, an output resolution of 512x512 pixels, a field of view of 45° and varying distances between the view cell and the model. Figure 4.8 shows how the complexity of the resulting TDMs increases with decreasing distance between the object and the view cell, as can also be seen in the respective statistics in Table 4.1. This characteristic is similar to the layered impostor approach, because visible parallax movements that have to be modelled with the mesh increase with decreasing distance.

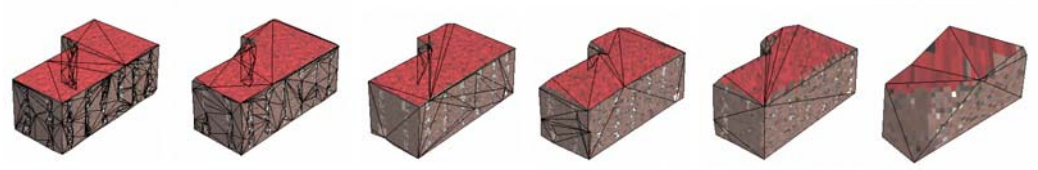

Figure 4.8: TDMs for the building model for varying distances between the model and the view cell. Note how the mesh approximates the geometry of doors and windows for nearby view cells.

Dist. view cell to model (m)	90	110	140	180	250	500	1000
# layers	110	81	56	34	19	5	1
# of meshes	4	2	1	1	1	1	1
# polygons in initial meshes	210,104	139,588	83,928	47,738	23,176	5,430	1,078
# polygons for TDMs	418	203	65	35	17	8	3
Memory for TDMs (kB)	154.2	89.7	49.7	26.4	12.3	3.46	0.64
Mesh generation time (s)	8.6	5.5	2.22	1.2	0.54	0.14	0.03
Mesh simplification time (s)	74.3	51	19.1	6.15	2.62	0.52	0.07
Mesh resampling time (s)	6.25	5.5	3.2	0.56	0.05	0.012	0.006
Overall time (s)	89.2	62	24.5	7.9	3.2	0.67	0.101
# polygons for layered. i.	1176	700	372	178	82	16	2
Memory for layered i. (kB)	329	198	107	57.4	25.5	4.5	0.67
Generation time for l.i. (s)	0.57	0.356	0.2	0.11	0.06	0.02	0.01

Table 4.1: Statistics for textured depth meshes and layered impostors for the house model for varying distances between model and view cell.

Since one main motivation for the TDM is to decouple the impostor complexity from the number of layers, Table 4.1 also shows a direct comparison to results obtained with the layered impostor approach presented in Chapter 3. As was expected, the TDM technique requires less memory and a smaller number of polygons if many layers are generated (compare to Table 4.1): up to 250 m distance, it needs less than half of the memory. Given the fact that layered impostors already constitute a memory-efficient technique, this difference is quite significant. For larger distances (and thus a smaller number of layers), the advantage of connecting the layers with a mesh becomes less important, so that the memory requirements for the two techniques are almost equal.

The behavior of the number of required polygons is quite analogous: for small distances ($\leq$ 250 m), the mesh contains in many cases less than one third of the polygons required for layered impostors. For close distances ($\leq$ 90 m), layered impostors need almost as many polygons as the original object (1880 polygons), while the TDMs still provide a much lower complexity. With increasing distance,

this advantage vanishes due to the same reasons as mentioned above. Also note that although the number of polygons in the intermediate mesh may seem high, this is mainly because of the number of pixels on screen and not because of the complexity of the original scene. It is unlikely that polygon counts will grow much beyond the values shown in Table 4.1.

On the downside, the generation time for a TDM is between 10 times and 150 times longer than for the layered impostor technique. The main time-consuming steps are the generation of the complex mesh, the simplification step and the texture generation step. The implementation of the first step is still unoptimized, so that this time could be significantly improved. The simplification step can also be accelerated by applying a fast pre-simplification approach as was proposed by Aliaga et al. [Alia99a]. However, because it is not expected that the generation time will be anywhere as fast as for layered impostors, it seems that TDMs are best suited for objects near the view cell. For those cases, the additional preprocessing time is justified by the savings in memory.

In order to demonstrate the generality and the wide variety of possible usages of TDMs, two additional test scenes with different characteristics are shown: first, a model of the Aztec city of Tenochtitlan (which is freely available on `www.3dcafe.com`) as a wide urban environment, and second, a single, relatively small car model. The resulting model statistics are shown in Table 4.2.

Model	Tenochtitlan	Car
# polygons original model	158944	1188
Model size (m)	450x450x50	4.7x1.8x1.5
Size of cubic view cell (m)	42	63
Distance view cell to model (m)	487	73
# generated meshes	12	2
# polygons in initial meshes	637462	83481
# polygons for final TDMs	946	220
Generation time (s)	72.6	3.9

Table 4.2: Model statistics for textured depth meshes of two very different models.

The urban model includes large occluded scene parts, which are automatically excluded by the algorithm (see Figure 4.9). On the other hand, the method guarantees that all visible scene parts are represented, so that no rubber-sheet effects or image gaps appear. While degenerate meshes can occur under rare circumstances, the output is usually of high quality: the mesh follows the shape of the objects to be represented quite precisely, even when relaxing the parameter for mesh simplification, as can be seen for the car model in Figure 4.10. Note that a factor of 2 was used to enlarge the voxel size during simplification for both test scenes.

Temporal aliasing (visible as flickering) is also greatly reduced in the TDM representation due to the proper scene sampling. The effect of oversampling the impostor textures can be observed in Figure 4.9, where the result looks smooth, but also a bit blurred.

The rendering acceleration provided by TDMs mainly depends on the rendering bottlenecks of the target rendering system. Savings in rendering time stem not only from the significant reduction in polygon count, but also from the simplicity of the representation: no state changes are necessary when rendering TDMs, allowing graphics hardware to be utilized to its fullest potential. As an impressive example, a TDM for a part of the Vienna model covering 600x400 pixels on the screen reduces the polygon count from 10.4 Million to only 81. The rendering time decreased from 766 ms to only 0.2 ms. However, the actual rendering acceleration depends on the original scene part, the final TDM and the rendering system, so that general statements are not possible.

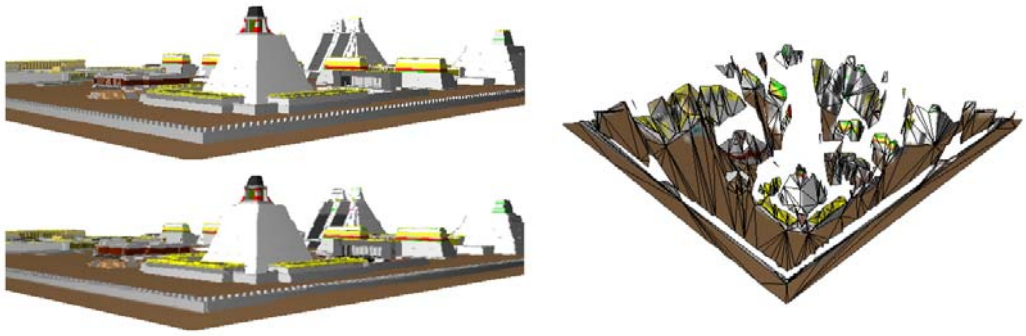

Figure 4.9: Top-left: original Tenochtitlan model. Bottom-left: the TDM representation. Right: the impostor from a bird's-eye view together with the corresponding mesh. Note that no occluded scene parts are present in the TDM.

4.9 Discussion

In this chapter, a novel approach for generating textured depth meshes was introduced. The geometric complexity of the representation is low, even for objects with a relatively small distance to the view cell. The reason for this is that the complexity is largely decoupled from the number of layers that where used to generate the TDM. For example, a continuous surface that requires many layers in a layered impostor can be represented with a single polygon in a TDM. Compared to

Figure 4.10: Left: original car model. Middle: the TDM representation. Right: the mesh of the TDM.

layered impostors (see Chapter 3), this results in reduced impostor complexity and lower memory costs, especially if the number of layers is high.

At the same time, the method retains most of the desirable features of the layered impostor technique. It guarantees a high output image quality by ensuring that the differences between the impostor and the geometry it represents are practically imperceptible. Previous approaches (see Section 2.3.1) try to achieve this by displaying multiple meshes in an output image that were generated from different viewpoints [Dars98, Wils03]. While this makes the impostor display less efficient, image artifacts are reduced but a high image quality is still not guaranteed for every possible view point. In contrast, the new method shows no image gaps or rubber-sheet effects while at the same time providing fast impostor display.

The meshes automatically adapt to the shape of the scene parts with respect to parallax movements. This constitutes a view-dependent simplification targeted at all views starting in the view cell. The metric for this simplification can be kept very simple due to the transformation into a uniform voxel grid. Furthermore, a single parameter is provided for controlling the tradeoff between mesh complexity and accuracy, allowing to adapt the technique for the demand of different applications.

Most invisible scene parts are automatically excluded regardless of scene structure, resulting in a more compact representation compared to previous approaches (see Section 2.3.1). Wilson et al. [Wils03] achieve this by sampling from multiple viewpoints and removing redundant information afterwards, which needs substantial preprocess time.

To summarize, the TDM technique is suitable for effectively accelerating the rendering of objects close to the view cell without noticeable image artifacts. The main drawback is the longer preprocessing time compared to the layered impostor technique. Depending on the application, this is counterbalanced by a more efficient representation.

Chapter 5

Automatic Impostor Placement

5.1 Introduction

An impostor placement defines for every output view those scene parts which should be displayed using impostors. This should be done in a way so that the impostors provide sufficient rendering acceleration and maintain a certain output image quality while at the same time keeping the required impostor memory as low as possible. Several impostor placement algorithms have been presented in the past (see Section 2.3.2). However, previous placement strategies do not provide satisfying results for a broad range of scenes, because the selected impostors need immense amounts of memory even for many small scenes. This has prevented impostors from being used for many applications. The most commonly used alternative is a manual placement (e.g., in games), which is tedious and time consuming.

In this chapter, we present a new automatic impostor placement algorithm that guarantees a user-defined minimum frame rate and a minimum output image quality for every possible view in a scene, while at the same time keeping the memory requirements for all impostors as low as possible. This is achieved by addressing the following main issues:

- Impostors are applied only for those views that cannot be rendered sufficiently fast.
- In contrast to previous approaches, *arbitrary* combinations of viewing regions and scene parts are considered for the placement, which is important for a satisfying solution.

- Additional acceleration techniques like visibility culling and geometric simplification techniques are used in parallel to impostors, thus making best use of all available techniques at the same time.

We will show that by addressing these issues, the memory required for the impostors can be kept very low compared to previous approaches.

5.2 Preceding Considerations and Requirements

In order to design an algorithm that guarantees a minimum frame rate, the time it takes to render a view in the scene must be quantified. Furthermore, the minimum image quality for an impostor must be defined. These two concepts are described in the following subsections. Afterwards, the main observations for a sensible impostor placement are discussed.

5.2.1 Definition of the Rendering Time

In order to ensure that *every* view in a scene can be rendered sufficiently fast, the rendering time must be known. Of course, it is impossible to render every view because the number of views is unlimited. Consequently, it is necessary to have an *estimation* for the time a view or a number of views need to render. Such an estimation must be very general, so that it can be adapted to varying hardware characteristics (see Section 2.2.1) and different scenes. This is provided by the *rendering time estimation* presented by Wimmer and Wonka [Wimm03]. This is a heuristic that predicts the duration of a certain rendering process taking into account system tasks, CPU tasks, GPU tasks and idle CPU and GPU times. For the impostor placement algorithm, the rendering time estimation will be used to get an upper limit for the rendering time for a view in the scene. Furthermore, it is also used to get the maximum time needed to render a particular scene part or its impostor. The rendering time estimation for a scene is obtained by a sampling process: many examples for the time needed to render different combinations of viewpoints and scene parts are evaluated in order to derive a heuristic for the rendering time. Details for this process are given in Section 5.8.1.

Note that previous approaches [Alia99c] only focus on bounding the number of primitives sent to the graphics pipeline for every output view, an approach which is not useful for modern graphics hardware. For example, in current graphics hardware, the number of draw calls is often more important than the number of primitives.

5.2.2 Definition of the Impostor Image Quality

In order to define a minimum output image quality for every output view, we introduce the concept of an *image quality criterion* for an impostor. An impostor is said to meet this image quality criterion if the following requirements are always fulfilled in the viewing region the impostor is defined to be valid (also see Section 1.3):

- The impostor resolution has to meet at least the output image resolution in order to avoid "blocky" artifacts.
- Parallax movements are reproduced with sufficient accuracy.
- No visible image gaps caused by disocclusions occur.
- The border between the impostor and the scene is not visible, which means, artifacts like image cracks are avoided.

These criteria ensure that the impostor does not become apparent as a replacement, and that no popping artifacts occur.

There exist methods for ensuring this criterion for most impostor techniques like for instance for planar impostors, textured depth meshes and point clouds (see Section 2.3.1). For instance, in the case of planar impostors, several impostors have to be built for the view cell in order to fulfill the image quality criterion. This leads to huge memory requirements. For layered impostors, instead, the number of layers can be adapted to meet the criterion using only a single impostor (see Section 3.3).

5.2.3 Observations for a Good Impostor Placement

Since the goal of the impostor placement is to guarantee a minimum frame rate, impostors should only be used for those views that cannot be rendered fast enough. Otherwise, impostor memory would obviously be wasted. Furthermore, in order to reduce impostor memory requirements, it is advisable to apply additional rendering acceleration techniques (if available) before using impostors. For instance, for urban and interior scenes, visibility culling allows reducing the rendering time for many output views significantly. Impostors are then used only were visibility culling fails to provide sufficient acceleration. The following two observations consider the selection of scene parts to be represented as impostors.

Many previous algorithms (see Section 2.3.2) address the fact that distant scene parts are very well suited for impostor representation because complex

geometry (which is favorable for rendering acceleration) covers only few pixels on the screen (which favors low memory requirements for an impostor). A closely related observation concerns multiple objects:

Observation 5.2.1 *If multiple objects are adjacent in object space, a common impostor for* all *objects is likely to require less memory than separate impostors for each object.*

This observation is especially true for distant objects that share many pixels on the screen. A beneficial side-effect of using larger object clusters with increasing distance is that this also reduces the number of rendering calls, and thus improves rendering acceleration on current graphics cards.

The second observation addresses parallax movements (and therefore disocclusions) which have to be represented by the impostor in order to meet the image quality criterion:

Observation 5.2.2 *If the appearance for a scene part hardly changes when seen from within a given viewing region, a single impostor for the whole region is likely to require less impostor memory than splitting the region into smaller view cells and generating an impostor for each such cell.*

This observation can typically be applied where parallax effects increase only marginally when increasing the view cell size, which is mainly the case for distant scene parts. On the other hand, nearby objects often show numerous disocclusions, which might lead to excessively high memory requirements and/or complex impostor geometry. Note that many previous approaches (see Section 2.3.2) generate impostors for a fixed set of view cells, separately for each cell. Because no distinction is made between nearby (apparently changing) and distant (hardly changing) scene parts, many similar impostors are generated for distant scene parts, which constitutes a waste of memory. In Section 5.8.2, we will show the great impact of using this observation for a good impostor placement.

5.3 Formal Problem Definition

In this section, we introduce a number of necessary terms and notations and formally describe the impostor placement problem.

The input model is assumed to consist of a set O of discrete objects o. The space of all possible viewpoints and view directions is called *view space* V. An element $v \in V$ is called a *view* and consists of a 3D *viewpoint* v_{3D} and a 2D *view*

direction v_{2D} (encoded as azimuth and elevation angles), so that $V = V_{3D} \times V_{2D}$. The field of view and image resolution is assumed to be fixed for all views.

Note that rotations around the view vector are ignored in V_{2D}. However, a simple way to incorporate rotations would be to increase the output resolution and field of view so that rotated views are also included.

The *rendering time* t_v is the time necessary to render a view v, and should ultimately stay below the user-defined maximum time t_{max}.

The *problem view space* $V_p \subseteq V$ is one of the fundamental concepts for our approach. It encodes for which *problem views* $v_p \in V_p$ the rendering time exceeds the desired maximum rendering time t_{max} (for a specific target hardware), i.e., where a "problem" occurs.

An *impostor* i can be created given three input parameters: a *view cell* $VC \subseteq V_{3D}$, an *object cluster* $OC \subseteq O$, and an *image quality criterion* IQ.

Each impostor is associated with a set $V_i \subseteq V_p$ of problem views within VC. Note that the impostor is *created* for a 3D view cell VC but *used* for a 5D view space (in short, i *serves* V_i). This distinction is made because an impostor might increase the rendering time for some views in VC, e.g., when most of the objects it represents are actually not visible for a particular view direction. Figure 5.1 illustrates this definition of an impostor.

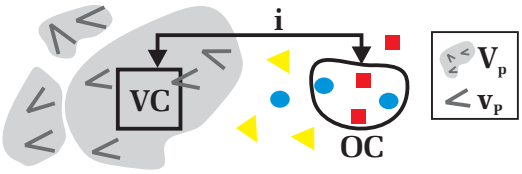

Figure 5.1: An impostor i is *generated* for a 3D view cell and *used* for a 5D problem view space subset.

An *impostor placement* for a view space V and an object set O is a set I of impostors, each representing a set of objects OC_i in the associated problem views V_i. The *impostor placement problem* can be cast as an optimization problem for finding an impostor placement I that

- satisfies the constraint that the rendering time t_v for each view $v \in V$ does not exceed the user-defined maximum frame time t_{max}. Such a placement is called *valid*:

$$\forall v \in V : t_v \leq t_{max} \tag{5.1}$$

  The time t_v is obtained assuming that the original objects are replaced by the impostors. If constraint (5.1) is met for a problem view, we say it is *solved*.

- minimizes the memory required for all impostors. Such a placement is called *optimal*:

$$\sum_{i \in I} m_i \rightarrow \min \tag{5.2}$$

  m_i is the memory needed for impostor i in this equation.

In order to solve the problem, an impostor placement algorithm has to decide on the selection of object clusters for which impostors should be generated, and for every impostor on the size and position of the view cells, as well as the problem views it should serve. Note that minimizing impostor memory typically also reduces the preprocessing time needed to generate the impostors.

The problem in finding a good impostor placement lies mainly in the huge number of valid impostor placements: even for a single impostor, the view cell may have an arbitrary size and position in the non-discrete view space. This fact makes a brute-force enumeration of all possible solutions to find the optimal one infeasible. The goal is therefore to limit the search space while still providing good placements. In the approach presented in this thesis, this process is guided by the observations described in Section 5.2.3, which leads to a reasonably small but well-chosen subset of all possible solutions, so that a good (even though usually suboptimal) impostor placement can be found in reasonable time.

5.4 Algorithm Outline

The impostor placement algorithm consists of four main steps:

- **Object set hierarchy generation:** The object set is subdivided hierarchically, clustering close objects first. This will address Observation 5.2.1. Possible clustering techniques include bounding box hierarchies, octrees and kd-trees. The decision on the depth of the subdivision is a tradeoff: a finer subdivision may result in a better impostor placement, but also increases the preprocessing time.

- **Problem view space approximation:** In order to approximate the problem view space V_p, the 3D positional view space V_{3D} as well as the 2D view direction space V_{2D} are subdivided hierarchically up to a user-defined accuracy. The result is a set of *conservative problem views* (in short, *CPVs*) in the sense that they include one or multiple problem views (see Section 5.5).

- **Impostor candidate generation:** A set of view cells is generated for every node of the object hierarchy, defining a set of *impostor candidates* (see Section 5.6). This addresses Observations 5.2.1 and 5.2.2. The number of candidates is a tradeoff between the quality of the placement and preprocessing time.

- **Optimization:** Finally, an optimization algorithm selects an impostor candidate subset so that constraint (5.1) is met and the optimization criterion (5.2) is approximated (see Section 5.7).

In the runtime system, the current CPV (if applicable) is looked up, and all impostors associated with this CPV are rendered instead of the original scene parts.

5.5 Problem View Space Approximation

The problem view space V_p may have an arbitrary 5D shape. In order to find which views cannot be rendered sufficiently fast, it is approximated using a hierarchical subdivision scheme. The approximation will be conservative in the sense that it will be a superset of V_p. The subdivision is first done along the axes of the 3D view space V_{3D}. For each resulting node, the view direction space V_{2D} is subdivided. The result of this 2D subdivision is used as a termination criterion for the 3D subdivision. This means that a 3D region is only subdivided further if its associated 2D subdivision includes at least one problem view. This allows a fast removal of areas where no problem views exist. Both subdivisions proceed to a user-defined minimal size.

5.5.1 3D View Space

Possible subdivision techniques include octrees, BSP-trees or kD-trees. The minimum region size for the subdivision must be chosen with care: smaller regions lead to better problem view space approximations, but the preprocessing time is also increased. Note that if too many objects intersect a leaf cell (so that the rendering time of those objects already exceeds t_{max} for some view), constraint (5.1) cannot be met, i.e., no valid impostor placement exists. This problem can be overcome in some cases by choosing a smaller minimum region size (see Section 5.9).

If the rendering system provides from-region visibility culling and/or geometric simplification, these techniques can be performed for each node during the 3D view space subdivision. Only the visible objects at the correct level of detail are then passed on to the 2D view direction subdivision.

The availability of such additional acceleration methods reduces the problem view space. Visibility culling for higher nodes in the hierarchy can be accelerated by immediately classifying all objects inside the view cell as visible.

5.5.2 2D View Direction Space

Given a node from the 3D view space hierarchy together with the (visible and simplified) part of the scene, the question is how to get the rendering times for all possible views within that region. This can be answered by extending the concept of *enclosing frusta* [Alia99c]. Figure 5.2 (left) illustrates this concept for a single view direction for the 1D case. The enclosing frustum for a 3D region contains all objects visible from viewpoints in that region with the same view direction. A rendering time estimation [Wimm03] (see Section 5.2.1) applied to the enclosing frustum is then a conservative estimation for the rendering time of every *enclosed view frustum*.

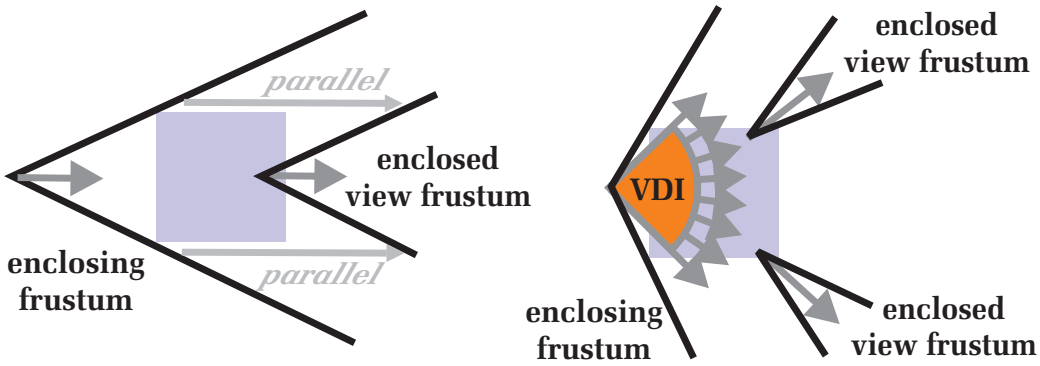

Figure 5.2: Enclosing frustum for a single view direction (left) and for a VDI (right), both for a 3D view region (light blue area).

This concept is extended to a range of view directions, which we call a *view direction interval* (in short, *VDI*). The enclosing frustum of a VDI is chosen so that it encloses all views with a view direction within its range (Figure 5.2 (right)).

The view direction space subdivision starts from a 360° VDI. Figure 5.3 illustrates the subdivision of a 90° VDI, resulting in two 45° VDIs for the 1D case. The figure also shows the resulting enclosing frusta.

For the general 2D case, we start with an initial view direction subdivision into six rectangular regions along every axis of the world coordinate system, thus forming a cube. View directions are then separately subdivided for each side of this cube (using a quadtree) in order to obtain a reasonably uniform subdivision. This is shown in Figure 5.3 (right) for one cube side.

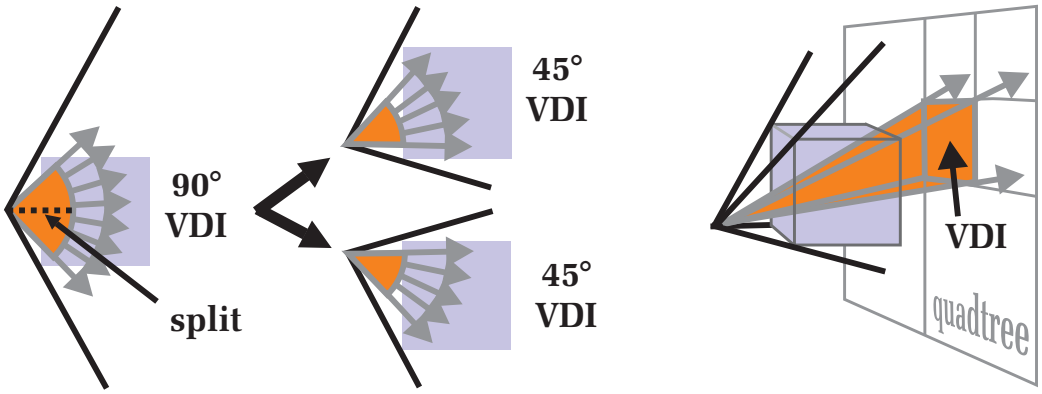

Figure 5.3: A $90°$ VDI (left) is subdivided in two $45°$ VDIs (middle). Right: 2D VDI subdivision using a quadtree.

In every subdivision step, the rendering time of every view within a VDI is again conservatively estimated by applying the rendering time estimation for its enclosing frustum. If the resulting time t_{enc} exceeds t_{max}, the VDI possibly contains a problem view and is subdivided further. This is done recursively until either t_{enc} is lower or equal to t_{max}, or the user-defined minimum VDI size is reached. In the latter case, the VDI is assumed to contain at least one problem view. If the 3D region has also reached its minimum size (see Section 5.5.1), the VDI together with the 3D region is stored as a conservative problem view. This CPV represents all views starting in the associated 3D region with a view direction within the VDI.

Because the 3D and the 2D view space subdivision always terminate at the same level for problem views, the result of the problem view space approximation is a set of equally-sized CPVs. Although this holds only approximately for the leaf nodes of the 5D view space hierarchy because of small distortions introduced in the view direction interval subdivision, these differences are not of practical interest as will become clear later.

5.6 Impostor Candidate Generation

In order to "solve" all conservative problem views generated in the previous step, impostor candidates have to be created so that the optimization algorithm can make a good placement. Every candidate is specified by an object cluster $OC \subseteq O$ and a view cell $VC \subseteq V_{3D}$, so that it serves each CPV with a 3D region enclosed by VC.

The main problem to be solved is the huge number of possible candidates. Therefore, Observations 5.2.1 and 5.2.2 are used for a selective candidate generation:

- The nodes of the object hierarchy are used to define OCs for the candidates. This allows the optimization process to address Observation 5.2.1, because larger nodes can be selected with increasing distance to the view cell.
- For every OC, a set of view cells has to be found so that Observation 5.2.2 can be met by the optimization. This means that with increasing distance between OC and VC, candidates with a larger view-cell size must be provided.

Many impostor approaches have used rectangular view cells [Alia99a, Wils01, Dars97, Jesc02b]. For these techniques, the nodes of the 3D view space hierarchy from the problem view space approximation (see Section 5.5.1) can be used directly so that every combination of a node of the object hierarchy and the 3D view space hierarchy defines a candidate.

Another view-cell shape that is often used implicitly [Jaku00, Aube99, Scha96b, Shad96] is a shaft (see Section 1.3). For each object hierarchy node, a set of shafts in different directions, apex angles, and with different minimum view distances is generated. The advantage of shafts compared to rectangular view cells is that they perfectly address Observation 5.2.2 because the view-cell extent grows with increasing distance to the object. In Section 5.8.2, we will show that shafts provide better results compared to rectangular view cells. Note also that view cells for view-independent impostors (e.g., billboard clouds [Deco03]) are shafts with a 360° apex angle.

Candidates are not considered if they serve no CPV within their view cell, or the combination of VC and OC allows no impostor generation (e.g., if OC intersects VC). This greatly reduces the overall number of candidates, so that only the most promising remain.

5.7 Impostor Placement Optimization

The optimization algorithm calculates an impostor placement from the set IC of impostor candidates generated in Section 5.6. This is done by applying the following steps:

- select a good subset $I \subseteq IC$ to be generated and used as impostors,

- associate with every CPV the set of impostors that serve it. This information is used during runtime for selecting the impostors to display for a particular view.

In order to find an optimal solution, all possible subsets of IC would have to be tested. This is prohibitive even though IC is already of moderate size compared to the original problem. Instead, we adopt a greedy approach: at every choice, the candidate with the best ratio between the rendering acceleration it provides for all served CPVs in relation to its memory cost is selected.

5.7.1 Rendering Acceleration

Figure 5.4 shows for a single viewpoint how the exact *rendering acceleration function* of an impostor i maps to the approximated version we will present below. The *exact rendering acceleration* $\Delta t^i_{V_p}$ of an impostor i is defined as the integral of the rendering acceleration $\Delta t^i_{v_p}$ for every served view v_p:

$$\Delta t^i_{V_p} = \int_{v_p \in V_p} \Delta t^i_{v_p} \, dv. \tag{5.3}$$

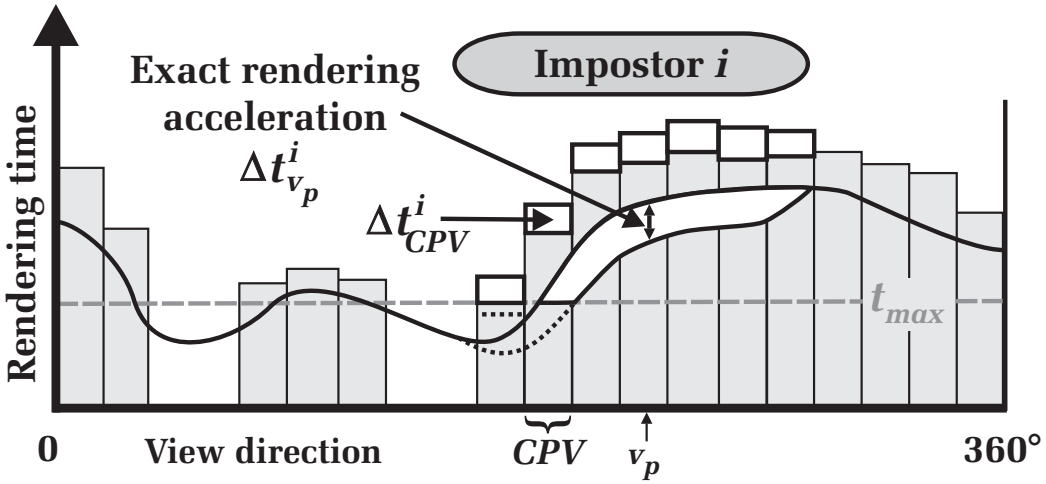

Figure 5.4: The rendering time for a view point for every view direction and the respective CPVs.

Since the exact rendering acceleration os impossible to calculate, we will rather consider an *approximate rendering acceleration* Δt^i_{CPV} of an impostor candidate i, defined for individual CPVs:

$$\Delta t^i_{CPV} = \max(0, t^o_{CPV} - \max(t^i_{CPV}, t_{max})), \tag{5.4}$$

where t^o_{CPV} is the original rendering time of the CPV and t^i_{CPV} is the rendering time of the CPV where impostor i is rendered instead of o. The two maximum-terms in this definition reflect the fact that

- any reduction of the rendering time to less than t_{max} is useless, so candidates with lower rendering acceleration, but also lower memory costs should be preferred in this case, and
- a candidate that takes longer to render than the original geometry will never be selected for p.

In practice, t^i_{CPV} can be approximated by $t^o_{CPV} - (t_o - t_i)$, where t_i and t_o are the times needed to render the impostor and the original object. Both are obtained with the rendering time estimation [Wimm03], while t^o_{CPV} is taken as the rendering time estimation of its enclosing frustum. Note that Δt^i_{CPV} varies for different CPV due to different level-of-detail selections, size on the screen, visibility culling etc. Furthermore, Δt^i_{CPV} is defined to be 0 for any CPV with a 3D region that is not enclosed by the view cell of i.

Finally, the overall rendering acceleration Δt_i, which approximates the integral $\Delta t^i_{V_p}$, can be calculated. Since all CPVs have the same size, Δt_i is defined by the sum:

$$\Delta t_i = \sum_{CPV \in CPVs} \Delta t^i_{CPV} \tag{5.5}$$

This sum can be calculated efficiently by traversing the 3D problem view space hierarchy and exploiting the fact that Δt^i_{CPV} can only be non-zero if the view cell for i encloses the 3D region of CPV.

5.7.2 Candidate Ranking

In order to select the "best" candidate in every greedy choice, each candidate i is ranked according to its *score* s_i. This score corresponds to the ratio of the overall rendering acceleration Δt_i obtained in all CPVs, and its required memory m_i:

$$s_i = \frac{\Delta t_i}{m_i} \tag{5.6}$$

Since the impostor for the candidate to be ranked has not actually been built yet, neither its exact memory requirements m_i nor its rendering time t_i is known exactly. Both have to be estimated based on the object cluster, the view cell, the CPV and the underlying impostor technique (see Section 5.8.1). However, because this estimation is only used for candidate ranking, the accuracy is not a crucial factor for the optimization. The final impostor rendering time is estimated using the generated impostors (see Section 5.7.3).

5.7.3 Greedy Choices

After all impostor candidates i have been ranked with a score s_i, the candidate with the highest score is chosen as an impostor. Such a choice entails the following steps:

- The impostor is generated. This allows a more accurate estimation of Δt^i_{CPV} based on the actual impostor geometry.
- For every CPV served (i.e., for which Δt^i_{CPV} is non-zero), add i to the set of impostors to display. The impostor can now be treated as belonging to the input scene, so that Δt^i_{CPV} can be subtracted from the rendering time t_{CPV} for those CPVs.
- Since a new value t_{CPV} is now available for all CPVs served by i, the scores s_i have to be recalculated for *all* impostor candidates that serve the affected CPV. This operation is accelerated using a lazy recalculation (see Section 5.7.4).

The optimization algorithm proceeds by selecting the next candidate. This is repeated until no candidate is left.

5.7.4 Lazy Recalculation

Recalculating the scores s_i after every step of the greedy optimization might be a very costly operation. Fortunately, no score s_i can increase for any remaining candidate. This is because Δt^i_{CPV} never increases for any CPV if an impostor is added to a CPV. On the other hand, the rendering acceleration Δt^i_{CPV} of an impostor i for a particular CPV might decrease if the view is almost solved, or even become 0 if a problem view is completely solved.

The fact that no candidate score can increase allows the use of a *lazy recalculation scheme*: after every greedy choice, the candidates with the highest scores are recalculated and re-ranked according to their new score. This is repeated until a candidate remains the best choice even after being re-ranked. Candidates with a new score of 0 can be deleted, because all CPV they serve have already been solved.

Lazy recalculation greatly reduces the number of operations needed for the optimization algorithm. Furthermore, for any candidate, only its score, its view-cell geometry and a link to its object hierarchy node have to be stored, as all other required information can be extracted on demand in reasonable time.

5.7.5 Overlapping Impostors

Some impostor candidates partially represent the same geometry for some common CPV. However, displaying multiple impostors for the same geometry in any view is not desired, because image quality problems (z-fighting) might occur. Fortunately, this problem can easily be avoided due to the object set hierarchy used for candidate generation. The hierarchy implies that if two object clusters OC_1 and OC_2 overlap, either OC_1 and OC_2 are identical, or one encloses the other, i.e., $OC_1 \subseteq OC_2$ or $OC_2 \subseteq OC_1$.

Assume an impostor i_1 has already been created for OC_1, and that a candidate i_2 representing OC_2 in some CPV p is the current best candidate. If $OC_2 \subseteq OC_1$, i_2 will not be used for p because i_1 is obviously the better choice. However, if $OC_1 \subseteq OC_2$, i_2 is used and i_1 is removed from p. When calculating the rendering acceleration of i_2, $t_p^{i_1}$ needs to be deducted from t_p instead of $t_p^{OC_1}$, because i_1 is the current representation for OC_1. Note that this consideration constitutes no special case for the algorithm, if already created impostors are treated just as if they *are* the objects they represent.

During the course of the greedy optimization, it is possible that an impostor is removed from all CPVs it is associated with. For that case, the impostor can be deleted and its memory regained.

5.8 Results

5.8.1 Test Setup

The test machine was a PC with an Intel Pentium 4 3.2 GHz and 1 GB memory. The graphics board is an NVIDIA GeForce Quadro FX 3000 with 256 MB of memory. The API was the OpenGL graphics API under the Windows XP operating system.

The automatic impostor placement algorithm was tested on the model of the city of Vienna (see Section 1.6). The object set hierarchy for this model is a bounding volume hierarchy with 9 levels. For all tests, we selected a view space of 1500x1000 m, which covers the whole city. For the problem view space approximation, the 5D view space subdivision was based on a regular binary space partition for the 3D view space and a view direction space subdivision as presented in Section 5.5. The 3D part of the CPVs had a side length of 23m, and the view direction space was subdivided to intervals of 11.25°. Visibility culling was done for the model using a conservative from-region visibility algorithm [Wonk00].

For every node of the object hierarchy, a number of shaft-shaped view cells form the impostor candidates. Shaft apex angles of 11.25°, 22.5°, 45° and 90° were used, and shaft directions in steps of 11.25°. The minimum allowed view distances for every shaft are from 1 to 2^{10} times the object size, doubled in each step. This setup has been found to provide a good tradeoff between a sufficiently high number of candidates and reasonable preprocessing time.

We have chosen the layered impostor technique presented in Chapter 3. Impostors were generated for an output image resolution of 512x512 pixels and 45° field of view. Note that layered impostors always fulfill the image quality criterion described in Chapter 3.

Parameter Estimation

This subsection describes the estimation of the rendering time of object clusters and impostors as well as the estimation of impostor memory requirements. These parameters are needed for the candidate ranking described in Section 5.7.2 and for the greedy optimization described in Section 5.7.3.

The rendering time for an object cluster OC for a particular CPV is obtained using a regression calculation. Therefore, a huge data set of examples of actually measured rendering times is generated as is shown in Figure 5.5. First, any node of

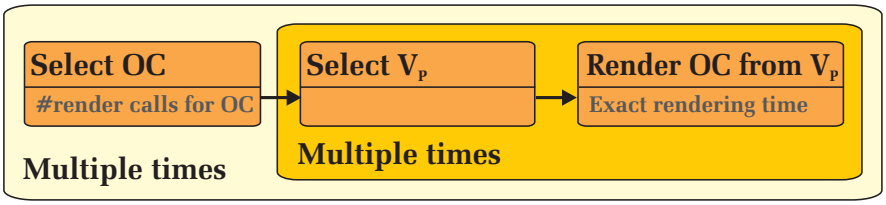

Figure 5.5: Setup for estimating the rendering time of object clusters in the Vienna model. Note that the information provided by every step is shown below every step.

the object hierarchy is stochastically selected to be an OC. For this node, several viewpoints are selected and OC is rendered from each such viewpoint. This is repeated for a huge number of object clusters and for every test set, the variables that are expected to influence the rendering time as well as the measured time needed to render OC are recorded. Afterwards they are used for a least-squares regression calculation.

It turned out that the rendering bottleneck for this model on our test machine was the number of *rendering calls* (a CPU bottleneck [Wimm03]) and not the number of primitives. Note that because collapsing several textured objects is still not efficiently possible using geometric levels of detail, impostors are actually the

only way for accelerating the rendering process for this type of scene without loss in image quality.

The second parameter that has to be estimated is the rendering time for impostors. This is done similarly to object clusters (see Figure 5.6): first, an object

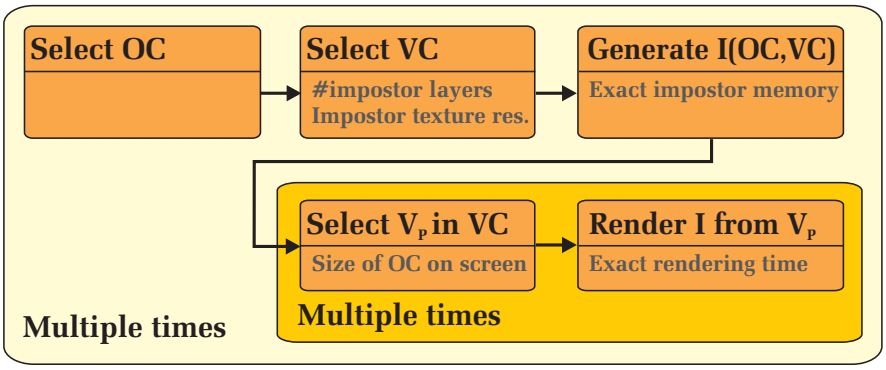

Figure 5.6: Setup for estimating the rendering time and the memory requirements for an impostor for the Vienna model.

cluster OC is stochastically selected. A shaft-shaped view cell VC is generated for OC by stochastically choosing the apex angle, the minimum allowed distance and the direction of the shaft. Afterwards, an impostor is generated from OC and VC. This impostor is rendered several times from within the view cell. This is repeated for a huge number of setups, recording the parameters of every test as well as the resulting rendering time. The resulting data is used for the regression calculation.

For the impostors, it turned out that the rendering time depends on the size of an impostor on screen (i.e., the number of pixels), so rendering an impostor can be called *fill-rate limited*. The size of an impostor on screen is typically the same as for the original object, so this value (which can easily be calculated) can be used for the candidate ranking.

The last information needed for the optimization process is the memory requirement of an impostor (see Section 5.7.2). The memory requirements for layered impostors are mainly defined by the sum of all impostor textures for every quadrilateral (see Section 3.7.2). In order to obtain an exact value for this, the impostor would have to be generated, which is too costly. However, it has been found that the resolution of the texture used for recording every impostor layer and the number of layers (both can be calculated in reasonable time) can be used to obtain an estimation that is sufficiently correct for the candidate ranking. The parameters for the heuristic are obtained using the same test setup as for the impostor rendering time estimation (shown in Figure 5.6).

5.8.2 Test Results

In order to show how the parameters of the algorithm influence its behavior, we first ran a "reference test" with the parameters described in the previous section and a target frame time of 16 milliseconds. Afterwards we successively changed various parameters. The respective results are discussed in the following subsections. Table 5.1 shows the resulting number of CPVs, impostor candidates and final impostors, and most importantly, the resulting impostor memory. The corresponding preprocessing times for the individual steps of the algorithm are summarized in Figure 5.7.

Test	Parameter	# CPV	# Candidates	# Impostors	MB
1	Refer. test (16ms)	9078	314659	6650	14.1
2	$t_{max} = 30ms$	2270	154696	1504	1.01
3	$t_{max} = 25ms$	3562	204110	2502	2.3
4	$t_{max} = 20ms$	5846	260891	4357	5.7
5	$t_{max} = 15ms$	10172	327600	7425	18.4
6	$t_{max} = 10ms$	18880	393202	14270	103.1
7	256x256 Pixels	9078	426944	6107	3.6
8	1024x1024 Pixels	9078	214345	7548	70.2
9	V_{3D} Approx: 46m	4509	310598	7749	30.6
10	V_{3D} Approx: 11.5m	24685	301537	6,306	11.6
11	Less Candidates	9078	33864	6152	28.6
12	More Candidates	9078	1544822	7022	13
13	Rectangular VC	9078	812248	16530	18.9
14	Per rectang. VC	9078	537776	137145	60.4
15	Environment maps	-	-	1545	233.1
16	No visibility (50ms)	36629	1467184	46845	80.1

Table 5.1: Statistics for the tests with the Vienna model.

Influence of target frame time: For tests 2–6, the target frame time was varied. As was expected, the more acceleration the impostors have to provide, the more memory and preprocessing time is needed. The required memory grows more than linearly with decreasing target frame time, since more and more closer objects have to be represented as impostors. This was already discussed in Section 3.7.2 and illustrates that impostors are most suitable for small and/or for distant objects.

Figure 5.8 shows the frame times for a sample walkthrough for these tests. While the target rendering time was met in all tests, a general over-

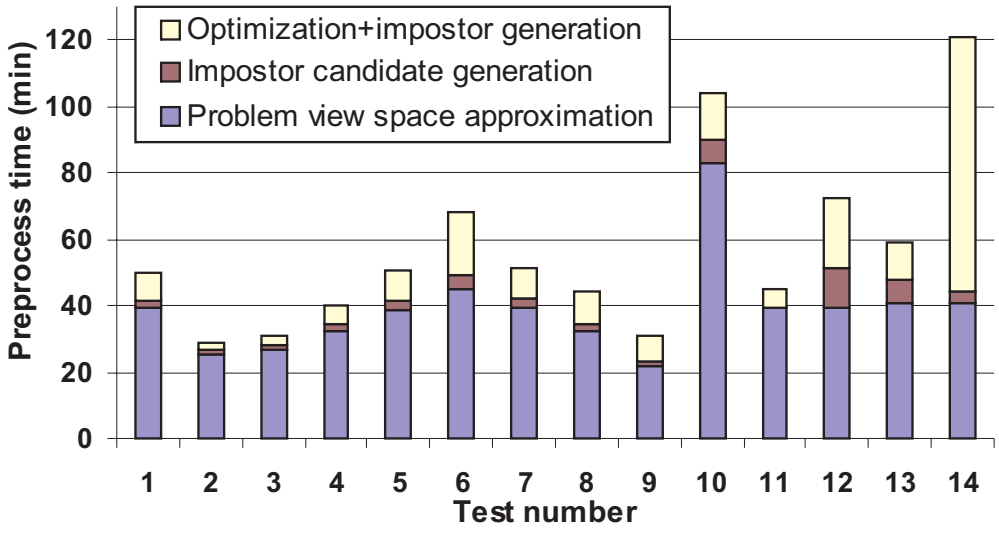

Figure 5.7: Times needed for the main steps of the algorithm.

conservativeness can be observed. The reason for this is that in our implementation, if a candidate has been chosen as an impostor, it is assigned to *every* problem view it accelerates, even if a problem view has already been solved. This induces no additional memory, but higher acceleration factors for many views.

Influence of output resolution/image quality criterion: Tests 7 and 8 analyze how the output resolution affects the necessary amount of impostor memory in comparison to the reference test. It can be seen that when doubling the output resolution, the memory increases by a factor of about 4 to 5. This can be explained by the fact that the number of impostor texels grows roughly quadratically with increasing output resolution and a higher resolution also results in a higher number of impostor layers, as already described in Section 3.7.2. Note that changing the output resolution is actually equivalent to changing the image quality criterion IQ for the layered impostor technique.

Influence of problem view space approximation accuracy: Tests 9 and 10 analyze how the problem view space approximation (i.e., different CPV sizes in V_{3D}) influences the result. Doubling the CPV size also doubles the required impostor memory, setting it to half the size shows diminishing returns, especially when taking into account the more than two-fold increase in preprocessing time. This shows that the approximation accuracy should be chosen with care.

Influence of the number of candidates: The number of candidates is a trade-off between a sufficiently large basis for a good optimization and reasonable pre-

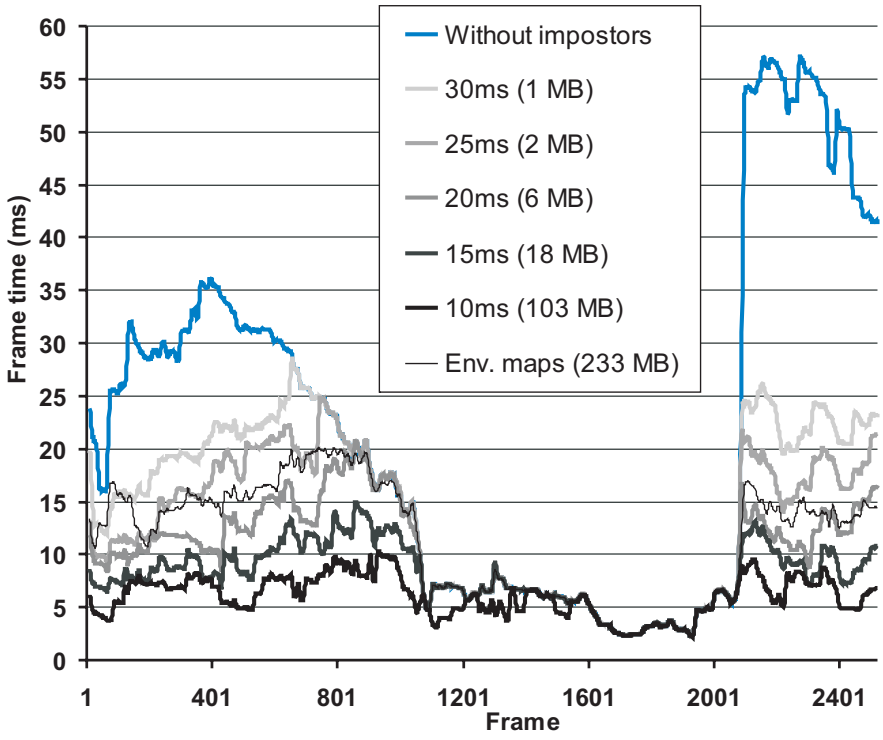

Figure 5.8: Rendering times for different target frame times for a walkthrough.

processing times. In test 11, only one tenth of the candidates compared to the reference test resulted in twice the amount of impostor memory. Test 12 shows that a three times increase of the candidate number barely improved the result, but increased the time for the candidate generation and optimization step.

Rectangular view cells: For test 13, nodes of the problem view space approximation were directly used for the impostor candidate generation, as was described in Section 5.6. Compared to the shaft-shaped view cells of the reference test, this results in a memory increase of roughly one third, which illustrates the advantage of shaft-shaped view cells for this type of scene. In cases were the view space is fragmented to small viewing regions as is the case, for example, in architectural models, rectangular view cells are expected to provide slightly better results.

Per-view cell placement: This test demonstrates the influence of Observation 5.2.2, i.e., the benefit of using large view cells for distant impostors. As for test 13, we used rectangular view cells that were directly obtained from the 3D view space hierarchy. In contrast to that test, only leaf nodes were allowed to serve as view cells for a candidate. We tried several sizes for the view cells (i.e., different view space hierarchy levels) with a best result of 60 MB for the impos-

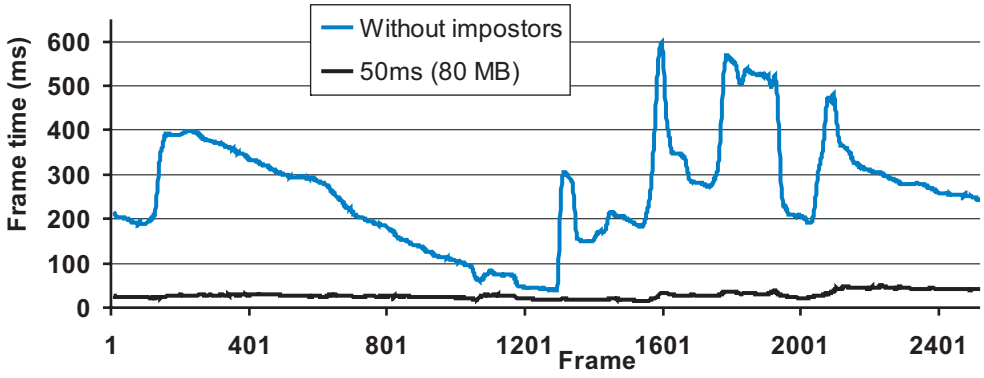

Figure 5.9: Rendering times for a walkthrough without occlusion culling.

tors, which is more than a three-fold increase compared to test 13 and a more than four-fold increase compared to test 1. Also note the much longer preprocessing time for generating the impostors. These results show that Observation 5.2.2 is an important factor for a good impostor placement.

Environment-map impostors: It is interesting to compare the results to a straight-forward approach (test 15), where impostors represent the whole scene from a certain distance (the so-called *far field*) for every view cell. Therefore, we implemented the layered environment-map impostors described in Section 3.8.2 (also refer to [Jesc02b]) by dividing the view space into a regular grid of view cells. For every cell, visibility culling was applied, and the impostors were arranged as environment-map layers, representing the whole visible scene from a certain distance. We tried several combinations of view-cell sizes and far-field distances and ended up with a view-cell sidelength of 25m and a far field distance of 300m as a good tradeoff. This resulted in slightly more than 4.5 hours of preprocessing time and 233MB memory for the impostors. Note that no frame rate guarantee is given by using impostors in this way, but Figure 5.8 shows that a frame time of 20ms is not exceeded during the walkthrough. In order to *guarantee* such a frame time, the new placement algorithm only needs 5.7MB of impostor memory, which is more than 40 times lower. This shows impressively that impostors should not be placed indiscriminately, but focussed on the scene parts that provide the best rendering acceleration.

Impact of visibility: In test 16, we turned off occlusion culling in order to see how impostor memory requirements increase for providing a frame rate of 50ms. The preprocess needed 7.6 hours. Figure 5.9 shows the rendering times for the same walkthrough as above. The results show that visibility culling is a significant factor for our test scene, and impostors should be used in conjunction with other

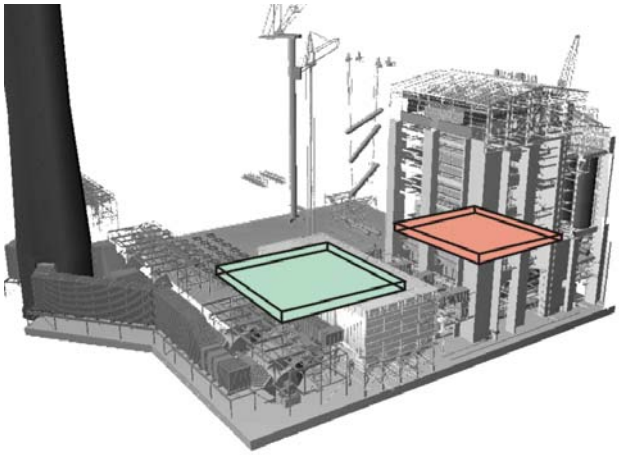

Figure 5.10: Two viewing regions at different positions in the UNC Power Plant model.

acceleration techniques in order to make best use of all available techniques.

5.8.3 Power Plant Results

To show how different view spaces influence the impostor placement result, we applied the algorithm to the UNC Power Plant model (see Section 1.6). For the object hierarchy, we created an 8-level deep octree of this model, where larger triangles were stored in interior nodes and the remaining ones in the leaf nodes.

We targeted the placement algorithm to limit the number of primitives in every output view to 100,000 polygons, similar to Aliaga [Alia99c]. A 40x40x2.5m viewing region was calculated for two side-by-side positions in the model (see Figure 5.10). The resulting impostor memory differs by almost a factor of 4 in this particular case, namely 146MB to 501MB. The reason for this difference is that in the red viewing region, many impostors are very near the view cell, causing a much higher memory consumption than impostors that are farther away. So the red viewing region requires more memory even though for the green viewing region, more geometry has to be represented by impostors.

This clearly shows how the required impostor memory may vary significantly for different view spaces, which makes it evident that comparisons between different impostor placement algorithms are only possible when using exactly the same parameters and view spaces for all tests, as we did in Section 5.8.2. This also shows that it is not adequate to extrapolate the memory requirements obtained in a small region to a whole model.

5.9 Discussion

In this chapter we presented an automatic impostor placement algorithm that guarantees a specified maximum frame time and a minimum image quality for every view within a scene. The approach integrates seamlessly with current real-time rendering systems and is not tied to a particular impostor technique. It can be used with a number of existing techniques (including billboard clouds [Deco03]) if they fulfill the image quality requirements mentioned in Section 5.2.2. It was shown that the memory required for impostors can be kept to a tolerable level even for large view spaces and scenes, when using impostors carefully and not indiscriminately. Several aspects have been taken into account in this connection.

First, impostors are only used for views that actually need them. These views are discovered using a rendering time estimation. This heuristic easily allows adapting the rendering acceleration provided by impostors to the rendering bottleneck of different target rendering systems. Note that previous approaches (for example, Aliaga and Lastra [Alia99c]) only concentrated on reducing the number of primitives, which is hardly related to the output frame rate on current graphics hardware.

Second, taking arbitrary combinations between *both* objects *and* view space regions for the impostor placement into account avoids generating many similar impostors for adjacent view space regions. We have shown that this main new insight significantly reduces the required impostor memory, while it was not addressed by any previous approach. The impostor placement optimization problem can be seen as a multiple choice multiple knapsack problem with partly overlapping items, each having a defined set of knapsacks it can be used for. While the greedy optimization presented in this thesis is not guaranteed to find an optimal solution, the impostor placements generated by the algorithm are very stable with respect to their input parameters. The algorithm seems to be well adapted to the problem because in typical cases, the items are small compared to the knapsack capacities, which constitutes a good condition for a greedy strategy. We tried using an exact algorithm at the end of the optimization phase, which barely improved the results.

Third, it was shown that taking visibility calculations into account *before* using impostors greatly reduces the required impostor memory and makes best use of all acceleration techniques at the same time. This is the reason why impostors have been used, for instance, for indoor scenes in portals (see Section 2.3.2). If no visibility culling is available for a scene, Jeschke et al. [Jesc02b] and Wilson et al. [Wils03] presented impostor techniques that include visibility calculations in the impostor generation process. This means that visibility is driven by impostor generation, not the other way round. However, in these approaches, impostors are

then again used separately for every view cell, which in turn leads to very high memory requirements as was shown in Section 5.8.2.

Note that our experiments concentrate on mid-range scenes, where the whole impostor database fits into graphics memory. This means that no restrictions on user movement speed are necessary for texture prefetching tasks, which is important for instance in computer games. However, the impostor placement algorithm is not restricted to such cases. If the resulting impostor database does not fit into graphics card memory, the application basically has two choices: either prefetching the impostor textures dynamically from harddisk, or lowering the image quality criterion IQ, e.g., by calculating impostors for a lower output resolution (see Section 5.8.2 for the effectiveness of this approach). Also note that the result of the placement algorithm might be used to generate the impostors on demand at runtime.

A general restriction of any impostor-based approach is that a scene together with the desired frame rate has to be suitable for impostor techniques. For instance, small objects which require a large part of the rendering time budget can easily overload nearby views. In this case, even a very fine view-space subdivision might not be sufficient to let impostors provide a desired frame rate. To state it more generally, the combination of the model, the specified rendering budget and the viewing region should allow for most of the nearby objects to be rendered using geometry.

Another point is that impostors are hardly suitable for accelerating the rendering of dynamic parts of a scene. While this problem cannot be easily overcome in itself, the impostor placement algorithm can still be used to accelerate the static parts of a scene. In order to guarantee a maximum frame time, the available rendering budget has to be split between the static and dynamic parts. The dynamic elements can then be treated with a predictive level-of-detail selection algorithm [Funk93], which makes sense since dynamic scene parts are often amenable to classical geometric simplification techniques.

In terms of future work, it is desirable to adaptively choose the degree of problem view space subdivision and the number of candidates that are generated in order to better adapt to different scene configurations. For instance, the problem view space subdivision might stop if no visibility changes are expected anymore and no better impostor candidates can be generated. This will reduce the number of problem views and allow processing extremely large view spaces. This is interesting because the impostor memory requirements do not necessarily increase with a growing view space. Furthermore, it is conceivable to automatically choose between different impostor techniques depending on the scene part to be represented and the viewing region to be served. Finally, in order to reduce the amount

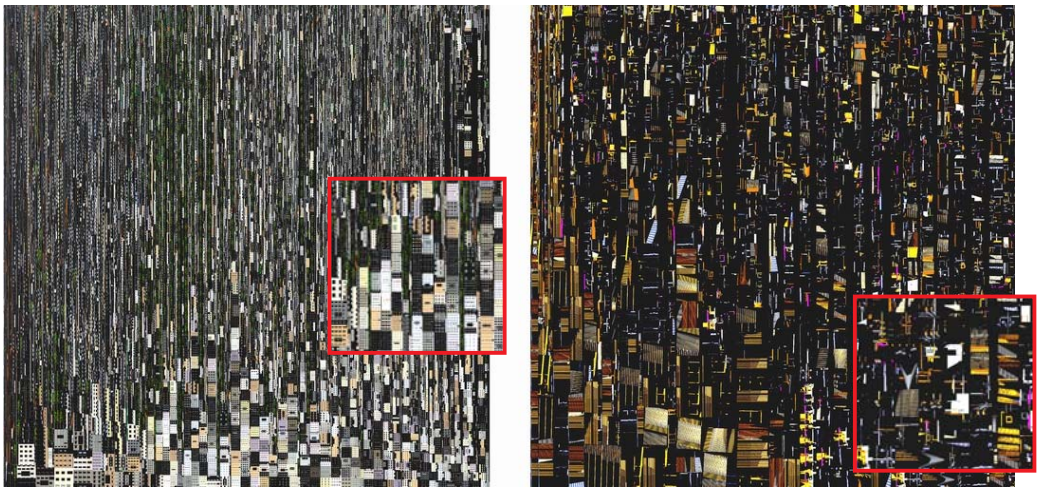

Figure 5.11: Impostor texture atlas for the Vienna model (left) and the UNC Power Plant model (right).

of impostor memory, it is interesting to identify similar textures for impostors and automatically instantiate them. Figure 5.11 shows texture atlases for the Vienna and the UNC Power Plant model. The close-up insert for the Vienna model shows numerous similar textures for trees and facades. An automatic instantiation of such similar textures might significantly reduce the number of textures with only marginally decreased image quality. This can be seen in the spirit of the work of Jakulin [Jaku00], who used instantiated impostors for complex objects (trees). Also note that in the texture atlas for the UNC Power Plant, similar textures are rare, so that instantiating the textures would be less effective in this case.

Chapter 6

Conclusions and Future Work

Rendering highly complex models in real time is a problem of high interest which still has not been solved until now. In order to reduce the complexity of a model for accelerated rendering, many approaches have been proposed in recent years. However, after even applying visibility culling, too complex geometry might remain, and geometric simplification techniques are not useful for arbitrary scenes. In such cases, impostors constitute a promising, fast-to-render representation that can replace arbitrary scene parts. However, while dynamic impostors put a high burden on the rendering system at runtime, static impostors suffer from problems like very high memory requirements, long preprocessing times and uncertain image quality. This has prevented impostors from being widely used in current real-time rendering applications. This thesis provides a number of algorithms and techniques in order to overcome these drawbacks.

6.1 Summary of Impostor Techniques

It was found that one key for the efficient usage of impostors is a representation that has low memory requirements and at the same time supports a large valid viewing region. Furthermore, for a convincing impostor representation it is highly desirable to avoid image gaps and rubber-sheet effects caused by disocclusions. In this thesis, a new layering scheme for acquiring appearance information from a scene part is presented. This scheme ensures that *all* visible scene parts are included with adequate resolution. Furthermore, invisible scene parts can be excluded efficiently. The new layered impostor technique consolidates the following desirable features:

- **Generality:** It can deal with arbitrary static models without the need for any knowledge about the scene structure.
- **Artifact-free representation:** The special layer setup guarantees that the differences between the impostor and the geometry it represents are imperceptible. In particular, image gaps or rubber-sheet effects due to missing information about hidden geometry are eliminated and aliasing effects due to over- or under-sampling are avoided. Popping artifacts that occur when switching between different representations are also practically imperceptible.
- **Low memory requirements:** The memory requirements for the impostors are kept quite low. This is partly the result of an efficient algorithm for encoding each layer, as well as an image-based visibility algorithm provided by the layering scheme.
- **Fast generation:** The impostor generation is computationally not costly. This allows impostor generation for larger scenes in reasonable time, as was illustrated in the thesis.
- **Fast display:** The method naturally supports conventional graphics hardware for fast impostor display by simply rendering alpha-textured polygons. Since no complex online calculations are necessary, optimal runtime efficiency is achieved.

These features make it possible to efficiently generate artifact-free representations for distant objects. On the other hand, layered impostors for near scene parts are less efficient due to the high number of required layers, which increases the required memory and the number of impostor polygons.

In order to overcome this drawback, we have presented a textured depth mesh impostor technique, which decouples the geometric complexity of the representation from the number of layers that are used to generate the impostor. Because of this fact, the technique is especially suited for representing nearby scene parts, where a high number of layers would make layered impostors less efficient in terms of memory requirements and geometric complexity. Compared to layered impostors, this allows making view cells larger while providing the same or even less complexity and memory cost. Furthermore, a single parameter is provided for controlling the tradeoff between mesh complexity and accuracy during the simplification algorithm. Graphics hardware naturally allows for fast impostor display in this technique. In addition, since it is also based on the layered scene recording technique, it shares many desirable features with layered impostors, such as generality, absence of image gaps or rubber-sheet effects, and the efficient exclusion

of hidden scene parts. The main additional cost for this technique is a relatively long preprocessing time. Depending on the application, this is counterbalanced by the more efficient representation.

6.2 Summary for the Impostor Placement Algorithm

A problem that usually leads to very high impostor memory requirements is the fact that previous approaches do not select scene parts to be displayed as impostors in an optimal way. This thesis presented an algorithm for automatically placing impostors into a scene so as to guarantee a minimum image quality and frame time for every view within the scene. The algorithm has the following desirable features:

- The algorithm is general because it works for arbitrary static scenes with many available impostor techniques.
- The placement is done “smart”, which means that impostors are only generated for views that actually need them. Furthermore, the use of a rendering time estimation function allows adapting the acceleration potential to different bottlenecks in every rendering system.
- It addresses the fact that for distant objects, a common impostor for adjacent view cells is likely to require less memory than separate impostors for every cell. This allows significantly reducing the memory needed for all impostors.
- The simultaneous use of visibility culling and geometric simplification techniques further reduces the required memory and allows making optimal use of all of these techniques at the same time.

It was shown that in combination with the new layered impostor technique, it is possible for a whole impostor database to completely fit into graphics hardware memory, even for larger scenes. This is highly desirable when prefetching the impostors dynamically is not an option, as for instance in computer games. On the other hand, a general constraint introduced with impostors is that the scene must be amenable for impostor usage: if very complex objects overload any nearby view, memory-intensive impostors would be necessary for these objects, thus reducing the efficiency of this approach.

6.3 Future Work

This thesis concentrated on providing low memory requirements for impostors while at the same time maintaining a high image quality. Another issue that has to be addressed by impostors is scene dynamics: impostors should appear like the original scene parts also if scene conditions change. Especially the desire for more scene realism causes the demand for a more flexible use of impostors.

For instance, in current computer games, every scene part may cast shadows, so it is desirable that an impostor can cast shadows as well. Until now, shadows were only presented for the billboard cloud technique [Deco03], which naturally supports the use of shadow mapping algorithms due to its view-independent characteristic. However, solutions must be found to support this feature also for view-dependent impostors.

Another point in this regard is the increasing use of programmable graphics hardware for realistic scene appearances. The support of complex shading effects like metal shading and environment map reflections is highly desirable for impostors. Many papers propose the use of normal maps for online lighting calculations (an example was presented for the billboard cloud technique [Deco03]). While this approach allows the reproduction of dynamic lighting, a problem occurs if various objects with different shading effects are represented by a single impostor. In this case, *all* shading programs would have to be present in the shading program for the impostor. The situation becomes even more difficult if the appearance of an object is defined by the vertex *and* pixel shader, as is the case for for environment maps. Wimmer et al. [Wimm01] presented a general approach that is based on a light field for representing view-dependent appearance changes, mainly at the cost of long impostor generation times and high memory requirements. However, the inclusion of dynamic effects like for example a skyscraper with moving clouds in the windows has not been solved until now. Changing lights make the problem even more difficult, because the appearance does not only depend on the view position but also on the (dynamic) configuration of the lights in a scene. Meyer et al. [Meye01] presented an approach for rendering trees over the day (with the changing sun illumination being a single dynamic light), but no general solution has been found until now.

In summary, it seems that solving these problems is challenging, particularly with regard to low memory requirements.

6.4 Conclusions

Impostors as image-based representations offer a convenient way to have the time needed to render a scene part depend primarily on the number of pixels it covers on screen rather than on the complexity of its geometric representation. Impostors are particularly useful for efficiently displaying numerous textured objects, which is still a problem for geometric simplification techniques, although these have been studied for a much longer time.

In order to decrease the amount of memory, which is the main cost of impostors, this thesis presented two new impostor techniques that are also aimed at eliminating image artifacts. Distant scene parts are especially suitable for an impostor representation, because for that case, complex geometry only covers few pixels on screen, which provides high rendering acceleration with low memory requirements. We presented a new impostor placement algorithm which gives a guarantee for a minimum frame rate for every output view. It was shown that by using impostors carefully and simultaneously applying additional rendering acceleration techniques, the amount of memory can be kept very low compared to previous approaches.

We believe that the algorithms and techniques presented in this thesis are capable of making impostors more useful for a broader range of applications in the context of real-time rendering, where impostors were not used before due to unacceptably high memory requirements and/or low image quality. The next challenge is the development of techniques that increase the flexibility of impostors with respect to the dynamic shading effects which programmable graphics hardware offers.

Bibliography

[Adel91] Edward H. Adelson and James R. Bergen. The Plenoptic Function and the Elements of Early Vision. *Computational Models of Visual Processing*, 1991. Cited on page 14.

[Aire90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 41–50, March 1990. Cited on page 19.

[Alia96a] Daniel G Aliaga. Portal Textures: Texture Flipbooks for Architectural Models. Technical Report TR96-049, Department of Computer Science, University of North Carolina - Chapel Hill, 1996. Cited on page 27.

[Alia96b] Daniel G. Aliaga. Visualization of complex models using dynamic texture-based simplification. In *Proceedings of the 7th conference on Visualization '96*, pages 101–ff. IEEE Computer Society Press, 1996. Cited on page 22.

[Alia97a] Daniel Aliaga, Jonathan Cohen, Hansong Zhang, Rui Bastos, Tom Hudson, and Carl Erikson. Power Plant Walkthrough: An Integrated System for Massive Model Rendering. Technical Report TR97-018, Department of Computer Science, University of North Carolina - Chapel Hill, August 28 1997. Tue, 7 Oct 1997 21:35:05 GMT. Cited on page 33.

[Alia97b] 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. Cited on page 33.

[Alia98a] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T.Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. A Framework for the Real-Time Walkthrough of Massive Models. Technical Report TR98-013, Department of Computer Science, University of North Carolina - Chapel Hill, 1998. Cited on page 33.

[Alia98b] D. G. Aliaga and A. A. Lastra. Smooth transitions in texture-based simplification. *Computers and Graphics*, 22(1):71–81, February 1998. ISSN 0097-8493. Cited on pages 22 and 27.

[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. Cited on pages 29, 34, 69, 79, and 91.

[Alia99b] Daniel G. Aliaga. *Automatically reducing and bounding geometric complexity by using images*. PhD thesis, University of North Carolina at Chapel Hill, 1999. Cited on pages 32 and 67.

[Alia99c] 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. Cited on pages 6, 7, 30, 34, 83, 89, 102, and 103.

[Andú00] Carlos Andújar, Carlos Saona-Vázquez, Isabel Navazo, and Pere Brunet. Integrating Occlusion Culling and Levels of Detail through Hardly-Visible Sets. *Computer Graphics Forum*, 19(3):499–506, August 2000. ISSN 1067-7055. Cited on page 20.

[Aube99] Amaury Aubel, Ronan Boulic, and Daniel Thalmann. Lowering the Cost of Virtual Human Rendering With Structured Animated Impostors. In V. Skala, editor, *WSCG'99 Conference Proceedings*. Univ. of West Bohemia Press, 1999. Cited on pages 5, 27, and 91.

[Aube00] Amaury Aubel, Ronan Boulic, and Daniel Thalmann. Real-time Display of Virtual Humans: Levels of Detail and Impostors. In *IEEE*

Transactions on Circuits and Systems for Video Technology, volume 10, pages 207–217, 2000. Cited on page 27.

[Bake78] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest. Orthogonal Packings in Two Dimensions. In *Proc. 16th Annual Allerton Conf. on Communication, Control, and Computing*, pages 626–635, 1978. Cited on page 60.

[Beck91] Becker, Franciosa, Gschwind, Ohler, Thiemt, and Widmayer. An Optimal Algorithm for Approximating a Set of Rectangles by Two Minimum Area Rectangles. In *CGMAA: Computational Geometry–Methods, Algorithms and Applications*, 1991. Cited on page 60.

[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. Cited on page 25.

[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. Cited on page 19.

[Bitt02] J. Bittner. *Hierarchical Techniques for Visibility Computations*. PhD thesis, Czech Technical University in Prague, 2002. Cited on page 20.

[Bitt04] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. In M.P. Cani and M. Slater, editors, *Rendering Techniques '04*, Eurographics Vol 23:3 (2004), pages 615–624. Blackwell, 2004. Cited on page 19.

[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. Cited on page 13.

[Bots02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 53–64. Eurographics Association, 2002. Cited on page 24.

[Catm74] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. Thesis, University of Utah, December 1974. Cited on page 23.

[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. Cited on page 16.

[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. Cited on page 15.

[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. Cited on page 24.

[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. Cited on page 15.

[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. Cited on page 14.

[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. Cited on page 14.

[Chen99] Baoquan Chen, J. Edward Swan II, Eddy Kuo, and Arie Kaufman. LOD-Sprite Technique for Accelerated Terrain Rendering. In David Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*, pages 291–298. IEEE Computer Society, 1999. Cited on page 25.

[Cign98] P. Cignoni, C. Montani, R. Scopigno, and C. Rocchini. A general method for preserving attribute values on simplified meshes. In *Proceedings of the conference on Visualization '98*, pages 59–66. IEEE Computer Society Press, 1998. Cited on page 60.

[Clar76] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10):547–554, October 1976. ISSN 0001-0782. Cited on pages 18 and 21.

[Coco02] Liviu Coconu and Hans-Christian Hege. Hardware-accelerated point-based rendering of complex scenes. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 43–52. Eurographics Association, 2002. Cited on page 24.

[Cohe96] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 119–128. ACM Press, 1996. Cited on pages 21 and 74.

[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. Cited on page 22.

[Cohe98b] Jonathan David Cohen. *Appearance-preserving simplification of polygonal models*. PhD thesis, University of North Carolina at Chapel Hill, 1998. Cited on page 22.

[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. Cited on page 19.

[Dach03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Transactions on Graphics*, 22(3):657–662, 2003. Cited on page 24.

[Dars96] L. Darsa and B. Costa. Multi-resolution representation and reconstruction of adaptively sampled images. In *SIBGRAPI'96 Proceedings*, pages 321–328, 1996. Cited on page 29.

[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. Cited on pages 28, 29, 77, and 91.

[Dars98] L. Darsa, B. Costa, and A. Varshney. Walkthroughs of complex environments using image-based simplification. *Computers and Graphics*, 22(1):55–69, 1998. Cited on pages 29, 33, and 81.

[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. Cited on pages 6 and 29.

[Deco02] Xavier Decoret, Fredo Durand, François X. Sillion, and Julie Dorsey. Billboard Clouds. Technical Report 4485, INRIA, Rhône-Alpes, 2002. Cited on page 31.

[Deco03] Xavier Decoret, Fredo Durand, François X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Trans. Graph.*, 22(3):689–696, 2003. Cited on pages 31, 32, 67, 91, 103, and 109.

[Deus02] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive Visualization of Complex Plant Ecosystems. In Robert Moorhead, Markus Gross, and Kenneth I. Joy, editors, *Proceedings of the 13th IEEE Visualization 2002 Conference (VIS-02)*, pages 219–226, Piscataway, NJ, October 27– November 1 2002. IEEE Computer Society. Cited on page 23.

[Disc98] Jean-Michel Dischler. Efficient Rendering Macro Geometric Surface Structures With Bi-Directional Texture Functions. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98*, Eurographics, pages 169–180. Springer-Verlag Wien New York, 1998. Cited on page 15.

[Dura99] Fredo Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Universite Joseph Fourier, Grenoble, France, July 1999. Cited on page 20.

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

Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 239–248. ACM SIGGRAPH, Addison Wesley, July 2000. Cited on pages 20, 54, and 58.

[E. P97] R. Scopigno E. Puppo. Simplification, LOD and Multiresolution - Principles and Applications. In *Eurographics'97 Tutorial Notes PS97 TN4*, pages 31–42, 1997. Cited on pages 21 and 74.

[Ebbe98] P. Ebbesmeyer. Textured Virtual Walls - Achieving Interactive Frame Rates During Walkthroughs of Complex Indoor Environments. In *Proceedings of the Virtual Reality Annual International Symposium*, page 220. IEEE Computer Society, 1998. Cited on page 27.

[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. Cited on page 22.

[Funk93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. In James T. Kajiya, editor, *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 247–254. ACM SIGGRAPH, Addison Wesley, August 1993. ISBN 0-201-51585-7. Cited on pages 21, 35, and 104.

[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. Cited on pages 21, 29, and 74.

[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. Cited on page 15.

[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. Cited on page 19.

[Gröl93] E. Gröller. *Coherence in Computer Graphics*. Ph.D. Thesis, Vienna University of Technology, 1993. Cited on pages 17 and 24.

[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. Cited on page 23.

[Harr01] M. J. Harris and A. Lastra. Real-Time Cloud Rendering. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3) of *Computer Graphics Forum*, pages 76–84. Blackwell Publishing, 2001. Cited on pages 5, 27, and 32.

[Hopp93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. In James T. Kajiya, editor, *SIGGRAPH 93 Conference Proceedings*, volume 27 of *Annual Conference Series*, pages 19–26. ACM SIGGRAPH, Addison Wesley, August 1993. ISBN 0-201-51585-7. Cited on page 21.

[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. Cited on page 22.

[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. Cited on page 22.

[Hopp98a] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings of the conference on Visualization '98*, pages 35–42. IEEE Computer Society Press, 1998. Cited on page 22.

[Hopp98b] Hugues Hoppe. Smooth View-Dependent Level-Of-Detail Control and its Application to Terrain Rendering. In *Proceedings IEEE Visualization'98*, pages 35–42. IEEE, 1998. Cited on page 22.

[Horo76] Steven L. Horowitz and Theodosios Pavlidis. Picture Segmentation by a Tree Traversal Algorithm. *J. ACM*, 23(2):368–388, 1976. Cited on page 59.

[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. Cited on page 19.

[Jaku00] A. Jakulin. Interactive Vegetation Rendering with Slicing and Blending. In A. de Sousa and J.C. Torres, editors, *Proceedings of Eurographics 2000 (Short Presentations)*. Eurographics, August 2000. Cited on pages 5, 8, 27, 28, 32, 67, 91, and 105.

[Jesc02a] Stefan Jeschke and Michael Wimmer. Textured depth meshes for real-time rendering of arbitrary scenes. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 181–190. Eurographics Association, 2002. Cited on page 9.

[Jesc02b] Stefan Jeschke, Michael Wimmer, and Heidrun Schumann. Layered Environment-Map Impostors for Arbitrary Scenes. In *Proceedings of the Graphics Interface 2002 (GI-02)*, pages 1–8, Mississauga, Ontario, Canada, May 27–29 2002. Canadian Information Processing Society. Cited on pages 9, 91, 101, and 103.

[Kuma96] Subodh Kumar, Dinesh Manocha, William Garrett, and Ming Lin. Hierarchical Back-Face Computation. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 235–244, New York City, NY, June 1996. Eurographics, Springer Wien. ISBN 3-211-82883-4. Cited on page 18.

[Lacr94] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *SIGGRAPH 94 Conference Proceedings*, pages 451–458, 1994. Cited on page 38.

[Leng97] Jed Lengyel and John Snyder. Rendering with Coherent Layers. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 233–242. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7. Cited on page 25.

[Levo85] Marc Levoy and Turner Whitted. The Use of Points as a Display Primitive. Technical Report TR 85-022, University of Carolina at Chapel Hill, 1985. Cited on page 23.

[Levo96] Marc Levoy and Pat Hanrahan. Light Field Rendering. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual

Conference Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. Cited on page 15.

[Li03] Wei Li and Arie Kaufman. Texture Partitioning and Packing for Accelerating Texture-Based Volume Rendering. In *Graphics Interface*, pages 81–88. CIPS, Canadian Human-Computer Commnication Society, A K Peters, June 2003. ISBN 1-56881-207-8, ISSN 0713-5424. Cited on page 59.

[Lipp80] Andrew Lippman. Movie-maps: An application of the optical videodisc to computer graphics. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 32–42. ACM Press, 1980. Cited on page 14.

[Lodi99] A. Lodi. *Algorithms for Two-Dimensional Bin Packing and Assignment Problems*. Ph.D. Thesis, University of Bologna, January 1999. Cited on page 60.

[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. Cited on page 22.

[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. Cited on page 20.

[Lueb97] David Luebke and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 199–208. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7. Cited on pages 21, 29, and 74.

[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. Cited on pages 3, 26, 32, and 35.

[Mail93] Jérôme Maillot, Hussein Yahia, and Anne Verroust. Interactive texture mapping. In *Proceedings of the 20th annual conference on Computer*

graphics and interactive techniques, pages 27–34. ACM Press, 1993. Cited on page 60.

[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. Cited on pages 25, 26, and 30.

[Maso99] Ashton E. W. Mason. *Predictive hierarchical level of detail optimization*. PhD thesis, University of Cape Town, 1999. Cited on page 22.

[Max95] N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In *Rendering Techniques '95*, pages 45–54. Springer, june 1995. Cited on page 23.

[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. Cited on page 14.

[McMi95] 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. Cited on page 14.

[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. Cited on page 30.

[Mese03] J. Meseth, G. Müller, M. Sattler, and R. Klein. BTF Rendering for Virtual Environments. In *Virtual Concepts 2003*, pages 356–363, November 2003. Cited on page 16.

[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. Cited on pages 38 and 52.

[Meye01] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. Interactive Rendering of Trees with Shading and Shadows. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 183–196. Springer-Verlag, 2001. Cited on pages 16 and 109.

[Möll02] Tomas Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters Limited, 2002. 2nd edition, ISBN 1568811829. Cited on page 22.

[Muth99] S. Muthukrishnan, Viswanath Poosala, and Suel Suel. On Rectangular Partitions in Two Dimensions: Algorithms, Complexity, and Applications. In Catriel Beeri and Peter Buneman, editors, *Proc. 7th Int. Conf. Data Theory, ICDT*, number 1540 in Lecture Notes in Computer Science, LNCS, pages 236–256. Springer-Verlag, 10–12 January 1999. Cited on page 59.

[Nico77] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometric Considerations and Nomenclature for Reflectance. Monograph 161, National Bureau of Standards (US), October 1977. Cited on page 15.

[Nire02] S. Nirenstein, E. Blake, and J. Gain. Exact from-region visibility culling. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 191–202. Eurographics Association, 2002. Cited on page 20.

[Pfis00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface Elements as Rendering Primitives. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 335–342. ACM SIGGRAPH, Addison Wesley, 2000. Cited on page 23.

[Pope98] Voicu S. Popescu, Anselmo Lastra, Daniel G. Aliaga, and Manuel M. de Oliveira Neto. Efficient Warping for Architectural Walkthroughs using Layered Depth Images. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of the conference on Visualization '98*, pages 211–216. IEEE, IEEE Computer Society Press, 1998. Cited on pages 30 and 33.

[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. Cited on page 15.

[Preu04] Gerke Preussner. Effiziente speicherung von impostern fr das echtzeitrendering komplexer 3D-Szenen. Diploma thesis, Fachbereich Informatik, University of Rostock, January 2004. Cited on pages 59 and 60.

[Raff98a] M. M. Rafferty, D. G. Aliaga, and A. A. Lastra. 3D Image Warping in Architectural Walkthroughs. In *Proceedings of the Virtual Reality Annual International Symposium*, page 228. IEEE Computer Society, 1998. Cited on pages 30 and 33.

[Raff98b] Matthew M. Rafferty, Daniel G. Aliaga, Voicu Popescu, and Anselmo A. Lastra. Images for Accelerating Architectural Walkthroughs. *IEEE Comput. Graph. Appl.*, 18(6):38–45, 1998. Cited on pages 8 and 33.

[Reev83] W. T. Reeves. Particle Systems a Technique for Modeling a Class of Fuzzy Objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983. Cited on page 23.

[Reev85] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 313–322. ACM Press, 1985. Cited on page 23.

[Rega94] Matthew Regan and Ronald Pose. Priority Rendering with a Virtual Reality Address Recalculation Pipeline. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 155–162. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0. Cited on pages 24 and 38.

[Rusi00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series, pages 343–352. ACM SIGGRAPH, Addison Wesley, July 2000. Cited on page 24.

[Scha95a] G. Schaufler and W. Stuerzlinger. Generating multiple levels of detail from polygonal geometry models. In *Selected papers of the Eurographics workshops on Virtual environments '95*, pages 33–41. Springer-Verlag, 1995. Cited on page 21.

[Scha95b] Gernot Schaufler. Dynamically Generated Impostors. In Dieter W. Fellner, editor, *GI Workshop on Modeling, Virtual Worlds,*, pages 129–135, November 1995. Cited on pages 26, 27, 32, 34, 43, and 64.

[Scha96a] Gernot Schaufler. Exploiting Frame to Frame Coherence in a Virtual Reality System. In *Proceedings of the 1996 Virtual Reality Annual International Symposium (VRAIS 96)*, page 95. IEEE Computer Society, 1996. Cited on page 35.

[Scha96b] 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. Cited on pages 5, 34, and 91.

[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. Cited on page 27.

[Scha98a] Gernot Schaufler. Image-based object representation by layered impostors. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 99–104. ACM Press, 1998. Cited on page 28.

[Scha98b] 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. Cited on pages 28, 32, 38, 46, 50, 52, and 59.

[Scha00] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 229–238. ACM Press/Addison-Wesley Publishing Co., 2000. Cited on page 20.

[Sequ01] Carlo H. Sequin and Maryann Simmons. Portal Tapestries. *The Pennsylvania State University CiteSeer Archives*, April 12 2001. Cited on page 33.

[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. Cited on pages 5, 27, 34, and 91.

[Shad98] Jonathan W. Shade, Steven J. Gortler, Li-wei He, and Richard Szeliski. Layered Depth Images. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 231–242. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8. Cited on pages 14 and 30.

[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. Cited on pages 8, 28, 29, 33, 34, and 69.

[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. Cited on page 25.

[Stam01] Marc Stamminger and George Drettakis. Interactive Sampling and Rendering for Complex and Procedural Geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 151–162. Springer-Verlag, 2001. Cited on page 24.

[Suth74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974. Cited on page 18.

[Suyk03] F. Suykens, K. vom Berge, A. Lagae, and P. Dutré. Interactive Rendering with Bidirectional Texture Functions. In P. Brunet and D. Fellner, editors, *Proceedings of the 24th Annual Conference of the European Association for Computer Graphics (EG-03)*, volume 22, 3 of *Computer Graphics forum*, pages 463–472, Oxford, UK, September 1–6 2003. Blackwell Publishing Ltd. Cited on page 16.

[Tell91] Seth J. Teller and Carlo H. Séquin. Visibility Preprocessing for Interactive Walkthroughs. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH 91 Proceedings)*, volume 25, pages 61–69. ACM SIGGRAPH, ACM Press, July 1991. Cited on page 19.

[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. Cited on page 25.

[Wand01] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Strasser. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 361–370. ACM Press, 2001. Cited on pages 24 and 36.

[Wand02] Michael Wand and Wolfgang Straßer. Multi-Resolution Rendering of Complex Animated Scenes. *Computer Graphics Forum*, 21(3):483–491, September 2002. Cited on page 24.

[Webe95] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 119–128. ACM Press, 1995. Cited on page 23.

[Wils00] Andrew Wilson, Ming C. Lin, Boon-Lock Yeo, Minerva Yeung, and Dinesh Manocha. A video-based rendering acceleration algorithm for interactive walkthroughs. In *Proceedings of the eighth ACM international conference on Multimedia*, pages 75–83. ACM Press, 2000. Cited on pages 28 and 33.

[Wils01] Andrew Wilson, Ketan Mayer-Patel, and Dinesh Manocha. Spatially-encoded far-field representations for interactive walkthroughs. In *Proceedings of the ninth ACM international conference on Multimedia*, pages 348–357. ACM Press, 2001. Cited on pages 28, 34, and 91.

[Wils03] Andrew Wilson and Dinesh Manocha. Simplifying complex environments using incremental textured depth meshes. In Jessica Hodgins and John C. Hart, editors, *Proceedings of ACM SIGGRAPH 2003*, volume 22(3) of *ACM Transactions on Graphics*, pages 678–688, 2003. Cited on pages 7, 29, 34, 57, 81, and 103.

[Wimm99] Michael Wimmer, Markus Giegl, and Dieter Schmalstieg. Fast walkthroughs with image caches and ray casting. *Computers and Graphics*, 23(6):831–838, December 1999. Cited on page 25.

[Wimm01] 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)*, pages 163–176. Eurographics, Springer-Verlag Wien New York, June 2001. ISBN 3-211-83709-4. Cited on pages 30, 57, and 109.

[Wimm03] Michael Wimmer and Peter Wonka. Rendering time estimation for real-time rendering. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 118–129. Eurographics Association, 2003. Cited on pages 18, 83, 89, 93, and 96.

[Wolb94] George Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 1994. Cited on pages 14 and 30.

[Wong97] Tien-Tsin Wong, Pheng-Ann Heng, Siu-Hang Or, and Wai-Yin Ng. Image-based Rendering with Controllable Illumination. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, pages 13–22. Springer-Verlag, 1997. Cited on page 15.

[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. Cited on page 20.

[Wonk00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 71–82. Springer-Verlag, 2000. Cited on pages 11, 20, 55, 56, and 95.

[Wonk01] Peter Wonka, Michael Wimmer, and François Sillion. Instant Visibility. *Computer Graphics Forum (Proc. Eurographics 2001)*, 20(3):411–421, September 2001. Cited on page 20.

[Wonk02] P. Wonka. *Occlusion Culling for Real-time Rendering Of Urban Environments*. PhD thesis, Vienna University of Technology, 2002. Cited on page 20.

[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. Cited on page 15.

[Xia96] Julie C. Xia and Amitabh Varshney. Dynamic View-Dependent Simplification for Polygonal Models. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 335–344. IEEE, October 1996. ISBN 0-7803-3673-9. Cited on page 22.

[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. Cited on page 19.

[Zwic01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378. ACM Press, 2001. Cited on page 24.

Selbstständigkeitserklärung

Ich erkläre, daß ich die eingereichte Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Rostock, Dezember 2004

Stefan Jeschke

Curriculum vitae

Personal information

Name:	Dipl.-Inf. Stefan Jeschke
Date of birth:	May 7, 1977 in Rostock, Germany
Marital status:	Single
Nationality:	German
Address:	Dorfstrasse 2, 18347 Dändorf, Germany
Email:	jeschke@informatik.uni-rostock.de
Languages:	German, English

Education

1983–1987:	Elementary School (Grundschule) in Dierhagen.
1987–1991:	Secondary School (Polytechnische Oberschule) in Wustrow.
1991–1995:	Secondary School (Gymnasium) in Damgarten.
June 1995:	Graduation (Matura).
1995–2001:	Studies of computer science at the University of Rostock, Germany, with special emphasis on Computer Graphics.
September 2001:	Graduation to “Diplom-Informatiker” at the University of Rostock.
Since April 2002:	Doctoral program at the University of Rostock, DFG-Graduiertenkolleg 466:“Verarbeitung, Verwaltung und Darstellung wissenschaftlicher Daten - technische Grundlagen und gesellschaftliche Implikationen”.

Jobs

Nov. 1998–Mar. 1999: Working for Philips Medical Systems in Hamburg.

May 1999–May 2000: Working for the University of Rostock, Computer Graphics Department.

Okt. 2001–Mar. 2002: Research assistant at the Institute of Computer Graphics and Algorithms, Vienna University of Technology.

Theses

1. Although the performance of common computer graphics hardware has dramatically increased in recent years, the demand for more scene realism for common scenes is growing even faster. Therefore, the rendering acceleration for three-dimensional scenes is a research area of big interest. The generation of a fluent animation with more than 60 frames per second is a special challenge. Possible applications are ship-, driving-, and flight simulators, virtual reality and computer games.

2. The only way for accelerating the rendering process is a reduction of the scene complexity for every output image. Especially the complexity of distant scene parts can be significantly reduced because of their high complexity but only low contribution to the output image.

3. Impostors are image-based entities used as an alternative representation of scene parts. Especially the appearance of distant scene parts hardly changes for a bounded viewing region, which makes the use of image-based representations suitable. Because the complexity of impostors is largely independent from the complexity of the represented scene part, they provide a very fast display of arbitrary complex geometry.

4. Impostors are the only way for accelerating the rendering process if visibility culling provides insufficient acceleration and geometric simplification techniques are not applicable.

5. The use of impostors must be transparent for the observer. This means, the difference in the output image between an impostor and the original rendered object must be very small, so that a convincing illusion of the original object is provided for the bounded viewing region. This has not been sufficiently addressed in previous approaches.

6. Impostors are generated either in a preprocess or at runtime, parallel to the rendering process. Problems for impostors generated in a preprocess are

very long preprocess durations and immense memory costs for the resulting impostors, even for small scenes. One reason for this is that previous impostor techniques need much memory and represent a scene part only from a small viewing region. This results in numerous memory intensive impostors needed for a whole scene.

7. We present a new impostor generation method, which allows guaranteeing a minimum image quality for a particular viewing region. Simultaneously, invisible scene parts can be efficiently excluded from the representation in order to obtain a more compact representation.

8. Based on this new method, two new impostor techniques were developed. One method allows the fast generation and efficient display of distant scene parts. The other technique focusses on a compact representation of scene parts near the viewing region. Both provide a minimum image quality for a relatively large viewing region and very low memory requirements compared to previous approaches.

9. Another reason for high memory requirements is the inefficient use of impostors in many previous approaches. Especially one aspect has been ignored during the selection of scene parts together with the according viewing regions: impostors for distant scene parts can be shared for multiple adjacent viewing regions because their appearance hardly changes.

10. We developed a new impostor placement algorithm that addresses especially the issue above. It provides a significant reduction of the required impostor memory. The algorithm is very general, which allows efficiently using impostors for arbitrary scenes and using arbitrary impostor techniques. Furthermore, impostor use can be aimed at different bottlenecks of a target rendering system.

11. The new impostor placement algorithm allows guaranteeing a minimum frame rate for every view within a scene. Together with the new efficient impostor techniques, this simultaneously provides a minimum output image quality as well as low memory requirements for the impostors.

12. The parallel use of visibility culling and geometric simplification techniques further reduces the required impostor memory. This offers to make the best use of all available techniques in a single framework.

13. If all impostors do not fit into graphics memory, they have to be dynamically fetched from harddisk during model display. This is not desirable for

numerous interactive applications. The new algorithms and techniques reduce the impostor memory so that all impostors completely fit into graphics memory, even for mid-range scenes.

14. The guaranteed minimum image quality and low memory requirements provided by the new algorithms and techniques account for the usability of impostors for many new applications.