Point-Based Multi-Resolution Rendering

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl. Inform. Michael Wand
aus Paderborn

Tübingen
2004

Tag der mündlichen Qualifikation:	30.06.2004
Dekan:	Prof. Dr. Ulrich Güntzer
1. Berichterstatter:	Prof. Dr.-Ing. Dr.-Ing. E.h. Wolfgang Straßer
2. Berichterstatter:	Prof. Dr. Markus Gross
	(Eidgenössische Technische Hochschule Zürich)
3. Berichterstatter:	Prof. Dr. Andreas Schilling

Zusammenfassung

Die vorliegende Arbeit beschreibt ein neues Paradigma für die effiziente Darstellung komplexer dreidimensionaler Szenen: die Verwendung von *punktbasierten Multiskalenmodellen*. Die Grundidee besteht darin, das Erscheinungsbild einer komplexen Szene aus einer (im Vergleich zur potentiellen Szenenkomplexität) kleinen Stichprobe von Oberflächenpunkten zu rekonstruieren. Hierarchische Datenstrukturen erlauben es, solche Stichproben effizient zu bestimmen, d.h. mit einer Laufzeit, die weitgehend unabhängig von der Komplexität der Szene ist. Dies ermöglicht es, Szenen mit sehr großer Komplexität effizient zu behandeln.

Es werden verschiedene Varianten von Datenstrukturen beschrieben, die eine solche effiziente Stichprobennahme erlauben, darunter auch eine Variante, die erlaubt, animierte Szenen (Keyframeanimationen) zu behandeln. Die Datenstrukturen dienen als Basis für verschiedene Darstellungsalgorithmen. Dabei werden zwei wesentliche Strategien behandelt: Die erste Klasse („forward mapping“) bildet die Szene unter einer einfachen perspektivischen Projektion ab. Eine Tiefenpuffertechnik rekonstruiert dann das Bild aus den Stichprobenpunkten. Der Rechenaufwand wächst dabei nur schwach mit der Szenenkomplexität, so daß Szenen mit enormen Mengen von Primitiva (Billiarden Dreiecke und mehr) in Echtzeit dargestellt werden können. Die erreichte Qualität entspricht dabei in etwa der von gewöhnlichen Tiefenpuffer-Algorithmen. Die zweite Klasse von Darstellungsalgorithmen („backward mapping“) verallgemeinert die Darstellungstechnik so, daß ein Raytracing auf punktbasierten Multiskalenmodellen durchgeführt werden kann. Dies erlaubt die Berechnung von Schatten, Spiegelungen und Brechung. Der Algorithmus benutzt eine Multiskalenhierarchie von vorgefilterten Stichprobenpunkten (also Stichproben eines jeweils entsprechend der Stichprobendichte bandbeschränkten Signals). Damit können Bilder ohne Aliasingprobleme erzeugt werden. Außerdem sind auch Approximationen von Effekten des klassischen „distributed raytracing“ wie etwa weiche Schatten, verschwommene Reflektionen oder Tiefenunschärfe mit geringen Kosten (ein Primärstrahl pro Pixel) möglich. Im Vergleich zum klassischen „distributed raytracing“ erzeugt der Algorithmus die Bilder effizienter, insbesondere wenn die Bildsignale eine hohe Farbvarianz aufweisen und Rauschartefakte vermieden werden müssen. Die Bildqualität ist grob vergleichbar mit der klassischen (korrekten) Lösung, die approximative Multiskalenstrategie führt nur zu kleineren systematischen Abweichungen.

Die vorliegende Arbeit führt eine theoretische Analyse der Stichprobenbestimmungs- und Darstellungsalgorithmen durch. Es werden obere Schranken für die Laufzeit des Darstellungsprozesses bewiesen, so daß eine Effizienz des Verfahrens unter relativ allgemeinen Bedingungen sichergestellt ist. Einzelne Schritte benutzen randomisierte Algorithmen. Hier wird gezeigt, daß die Wahrscheinlichkeit dafür, daß der Algorithmus zufällig nicht das gewünschte Ergebnis bestimmt, mit geringem Aufwand sehr klein (beliebig klein bei schwachem Aufwandswachstum) gehalten werden kann. Die verschiedenen Verfahren zur Stichprobenentnahme werden auch hinsichtlich des „oversampling“ verglichen, d.h. dem Verhältnis der Zahl der Stichprobenpunkte zur

tatsächlich notwendigen Anzahl bei einer optimalen Auswahl. Die besten vorgestellten Verfahren sind dabei nur um einen kleinen Faktor vom Optimum entfernt.

Weiterhin werden experimentelle Ergebnisse vorgestellt, die auf einer prototypischen Implementation der vorgeschlagenen Algorithmen beruhen. Damit wird der Einfluß verschiedener Parameter der Algorithmen auf Laufzeit und Bildqualität in der Praxis untersucht und mit den theoretischen Voraussagen verglichen. Außerdem werden Beispielanwendungen beschrieben, in denen Szenen mit sehr großer Komplexität in Echtzeit dargestellt werden können. Dabei wird auch gezeigt, daß effiziente dynamische Modifikationen der Datenstrukturen möglich sind, die für interaktives Editieren komplexer Szenen benötigt werden. Die Darstellungsalgorithmen werden zudem auf animierte Szenen angewandt, hier am Beispiel von Massenanimationen (z.B. Darstellungen großer Menschenmassen oder Tierherden mit dynamischem Verhalten).

Abschließend wird noch kurz auf Verallgemeinerungen eingegangen. Diese betreffen die Echtzeitdarstellung sehr großer Volumendatensätze, eine effiziente Darstellung von Datensätzen, die aufgrund ihrer Größe auf Hintergrundspeichermedien (Festplatte) gehalten werden müssen, sowie eine Anwendung auf die Berechnung von Geräuschkulissen, die ein interaktiver Beobachter in Szenen mit einer großen Anzahl von Schallquellen wahrnimmt. Ein weiterer kurzer Exkurs diskutiert eine Echtzeitberechnung von Kaustiken von ausgedehnten Lichtquellen, die ebenfalls auf punktbasierten Diskretisierungen von Oberflächen beruht.

Insgesamt erweitern die neu vorgeschlagenen punktbasierten Multiskalenansätze deutlich die bisherigen Möglichkeiten, Abbildungen komplexer dreidimensionaler Szenen effizient zu berechnen.

Abstract

This thesis describes a new rendering paradigm for handling complex scenes, point-based multi-resolution rendering. The basic idea is to approximate the appearance of complex scenes using a small set of surface sample points. Using hierarchical data structures, the sampling process can be performed in time mostly independent of the scene complexity. This allows an efficient display of highly complex scenes.

The thesis proposes different variants of sampling data structures that are useful in different application scenarios, including a variant for handling animated scenes (general keyframe animations). Two different rendering approaches are described: The first approach is a real-time forward mapping algorithm, being a point-based generalization of z-buffer rendering. In contrast to conventional z-buffer rendering, the point-based multi-resolution algorithm can render scenes consisting of trillions of primitives at real-time frame rates while maintaining a comparable rendering quality. The second approach is a backward mapping (i.e. raytracing) algorithm that aims at offline rendering. It is able to compute shadows, reflections, and refractions. It uses a hierarchy of prefiltered sample points to provide efficient antialiasing. Additionally, classic distributed raytracing effects such as soft shadows, depth-of-field, or blurry reflections can be approximated efficiently. In comparison with classic stochastic raytracing techniques, the new algorithm provides noise-free renderings at lower costs than stochastic oversampling for scenes of high variance. The image quality is roughly comparable to that of the classic approach; only a small bias is observed.

The thesis provides a theoretical analysis of the sampling and rendering process. Upper bounds for the rendering time are established. For the randomized components of some of the algorithms, analytical lower bounds for the failure probability are derived, showing that arbitrarily high confidence probabilities can be achieved at a small increase of computational costs. An analysis of oversampling properties of different sampling and stratification strategies allows a quantitative comparison, needed to choose the best technique for a certain application.

A prototype implementation is presented. The influence of different algorithmic parameters is evaluated empirically and compared to theoretical predictions. Practical applications of the proposed algorithms comprise real-time walkthroughs of highly complex static scenes as well as real-time visualizations of large crowd animations such as a herd of animals or a football stadium with ten thousands of animated football fans. In addition, dynamic modifications of the data structure as needed for interactive editing is examined.

Finally, extensions to volume rendering, out-of-core rendering, sound rendering, and simulation of caustics from area light sources are discussed briefly.

Overall, the presented techniques extend the possibilities for rendering of highly complex scenes to areas that could not be treated before with comparable efficiency.

Table of Contents

Preface

This thesis describes a novel output-sensitive approach to rendering of highly complex scenes: point-based multi-resolution rendering. The work on this topic has been started in mid 1999. At that time, the goal was to develop a rendering algorithm that permits rendering of highly complex scenes such as landscape scenes with natural vegetation at interactive framerates. The initial idea was to use a random sample of surface points, distributed according to perspective foreshortening, in order to reconstruct images independent of the actual scene complexity. It turned out that this rendering paradigm was able to render images of extremely complex scenes with good image quality and within reasonably short rendering times. The approach, the *randomized z-buffer* (which has been subject of the diploma thesis of the author of this thesis [Wand 2000a]), is the basis of this thesis.

The usage of sample points as rendering primitives has not been a completely new idea: Point-based rendering techniques have already been used earlier. In the 1980s, different authors proposed point-based rendering and simulation techniques ("particle systems") for rendering special effects (fire, clouds, smoke) and vegetation [Reeves 83, Reeves and Blau 85]). In the 1990s, point-based multi-resolution techniques have for example been used for the special case of real-time terrain rendering in computer games ([Novalogic 92, Freeman 96]).

Concurrently with the work on the randomized-z-buffer approach, other authors have also worked on the usage of point-based multi-resolution techniques for more general rendering problems. [Pfister et al. 2000] and [Rusinkiewicz and Levoy 2000] have proposed two deterministic point-based multi-resolution rendering techniques dubbed "Surfels" and "QSplat", respectively. These approaches have certain advantages and disadvantages in comparison to the randomized z-buffer approach. In consequence, for later work described in this thesis, both approaches have been combined in order to optimize rendering time and results.

Point-based multi-resolution techniques have just recently been recognized as general tool in computer graphics. Nevertheless, they have already been applied to treat a variety of problems and are still a hot topic of current research. This thesis should show that point-based multi-resolution approaches allow the rendering of general, highly complex scenes at real-time framerates with good image quality. Additionally, their applicability to several rendering problems ranging from raytracing to sound rendering should be demonstrated.

Acknowledgments

First, I would like to thank my advisor, Prof. Straßer. This work would not have been possible without his long term support. I also would like to thank Prof. Gross and Prof. Schilling who have agreed to be part of my graduation committee. I wish to thank colleagues and students at WSI/GRIS at University of Tübingen for their support. In particular, I would like to thank Michael

Doggett, Stefan Gumhold, Stefan Guthe, Michael Hauth, Ingmar Peter, and Stanislav Stoev for valuable discussions and help with papers and videos. The volume rendering architecture has been designed in corporation with Stefan Guthe, who also did all the implementation and evaluation, supported by Julius Gonser.

Martin Frisch and Amalinda Oertel have created the 3d-models used in the evaluation of the rendering technique for animated scenes. Matthias Mueller has implemented the Poser import filter. The Poser Animation software has been provided by Tobias Hüttner/EgiSys. I also wish to thank Michael Hauth, Johannes Mezger, Dirk Staneker, Ralf Sondershaus, and Gregor Wetekam for their help when the time schedule for finishing this thesis became tight.

The initial work on the randomized z-buffer technique has been started at University of Paderborn. I wish to thank in particular Matthias Fischer, Tamás Lukovszki, Christian Sohler and Prof. Meyer auf der Heide for their support. The implementation and evaluation of the out-of-core variant of the data structure has been carried out by Matthias Fischer, Jan Klein, and Jens Krokowski. I also wish to thank Matthias Zwicker from ETH Zurich for valuable discussion concerning fragment merging within the surface splatting approach.

Finally I also wish to thank my family for their support. Special thanks go to Birgit and Mathias Wand for proofreading.

Introduction

Rendering and Complexity

Today, interactive computer graphics has to deal with data sets of increasing complexity. This trend can be observed in many application areas: In scientific visualization, the complexity of data sets has grown enormously due to improvements in sensor and simulation technology. In the area of computer aided design (CAD), three-dimensional virtual prototypes of complex projects should be visualized and edited interactively. The entertainment industry has to deal with complexity problems, too: Recent computer generated feature films [Square 2001] already come close to photo-realistic simulations of reality, thus requiring highly detailed three-dimensional models. Special effects for feature films (for instance the large crowd animations in the recent Lord of the Rings trilogy [New Line 2003]) also depict highly complex scenes. Interactive entertainment applications, such as computer games, aim at a photo-realistic simulation of virtual worlds. Such interactive applications pose especially high demands on graphics algorithms: As the user is free to move interactively in a virtual world, a large amount of detail has to be modeled. Different scales, ranging from leaves of grass to complete mountains must be included to sustain a realistic impression for different view positions. This leads to scene data bases of huge complexity. Additionally, all computations must be handled in real-time. All these developments lead to the somehow paradox fact that handling of complexity is today still one of the major problems in computer graphics, despite the enormous advances in graphics hardware and algorithms.

In this thesis, we propose a new approach to render highly complex scenes: point-based multi-resolution rendering. The main idea of this technique is to reconstruct images from surface sample points: Instead of processing all primitives that describe a potentially highly complex three-dimensional scene, we only pick a small set of sample points from the surfaces. Sampling is done using a sampling distribution that facilitates the reconstruction of an image later on. Such a generation of the sample sets can be done efficiently, in time mostly independent of the scene complexity. Hence, it is possible to apply the rendering algorithm to scenes of very high complexity while maintaining acceptably low rendering costs. The paradigm of reconstruction from point sample sets is a very general technique that can be applied to a large class of scenes, being more general than many former approaches.

Related Work

A lot of work has been done in order to enable an efficient rendering of highly complex scenes. In general, two approaches should be distinguished: Forward mapping algorithms create images by projecting rendering primitives onto the screen, then resolving their visibility. The most widespread forward mapping technique (which is used predominantly in interactive applications today) is the z-buffer algorithm [Catmull 74, Straßer 74]. Backward mapping algorithms create

images by searching the object that is seen for each pixel, leading to the well-known raytracing algorithm [Appel 68, Whitted 80].

In order to apply forward mapping to highly complex scenes, two problems have to be solved: First, the level-of-detail of the projected objects has to be adapted to their on-screen size, demanding for automatic *simplification* algorithms. Second, occluded objects should be excluded from rendering by the use of *occlusion culling* algorithms [Cohen-Or et al. 2001]. Simplification is typically achieved by the use of *mesh simplification* algorithms [Garland and Heckbert 97, Puppo and Scopigno 97, Garland 99, Klein 99] that automatically adapt the level-of-detail of triangle meshes to the demanded level. Point-based multi-resolution rendering is essentially also a simplification technique, however, using point primitives instead of triangles as simplified representation. The advantage of the point-based approach is that it does not have to deal with topological constraints. Thus, it is often easier to implement and applicable to a wider range of scenes. Even highly irregular objects such as three-dimensional models of a forest or a large crowd of people can be handled effectively. Additionally, a generalization to animated scenes is straightforward while being more involved for mesh-based techniques.

Another option for a rapid display of complex scenes is image-based rendering: The scene or parts of it are replaced by prerendered images that can be displayed rapidly [Maciel and Shirley 95, Gortler et al. 96, Levoy and Hanrahan 96, Shade et al. 96]. Prerendered images are of course the most efficient way of displaying complex scenes. However, these techniques usually have to deal with issues concerning large memory and preprocessing demands as well as parallax errors in the rendering results. As a result, many advanced image-based rendering techniques use geometric information such as a per-pixel depth to compensate for parallax errors. This reduces the needed number of precomputed images, but still artifacts occur at borders: Holes may become visible due to missing information. As a consequence, multiple depth samples are stored at every pixel location, providing an approximate three-dimensional representation of the model [Shade et al. 98, Grossman and Dally 98]. This development naturally leads to point-based rendering, i.e. representing and simplifying geometry by the use of point clouds as representation primitive.

Point primitives have been used in computer graphics before, especially to model and simulate irregular phenomena such as smoke or fire [Csuri et al. 79, Reeves 83] or to render outdoor vegetation efficiently [Reeves and Blau 85]. In order to use point-based representations for interactive walkthroughs of complex scenes, multiple levels-of-detail have to be provided for all parts of the scene. In the early 1990s, computer games appeared that used point-based multi-resolution techniques for the efficient rendering of terrains [Novalogic 92, Freeman 96]. Hierarchical approximation of volume data sets has been investigated by [Laur and Hanrahan 91]. [Max 96] uses scene graph hierarchies of point clouds to render tree models efficiently. The work described in this thesis has been motivated especially by the work of [Chamberlain et al. 95, Chamberlain et al. 96]: An octree hierarchy is used to facilitate a display with different level-of-detail. The cube sides of the tree nodes are colored according to average color and alpha attributes. However, this still leads to some rendering artifacts; a save coverage of continuous surfaces cannot be guaranteed. These issues can be circumvented by replacing colored boxes with point clouds that ensure a certain sampling density. This strategy, which is the key idea described in this thesis, permits analytical guarantees for save surface coverage.

The same approach has been followed by other researches, too. In parallel to the initial point-based multi-resolution rendering technique described in this thesis (the *randomized z-buffer* algorithm [Wand 2000a]), the "*Surfels*" approach [Pfister et al. 2000] and the "*QSplat*" rendering system [Rusinkiewicz and Levoy 2000] have been developed. These two proposals also rely on point-based multi-resolution representations to render complex scenes. In contrast to the ran-

domized z-buffer technique, they employ deterministic, precomputed sample sets. Many ideas from the "Surfels" and "QSplat" system have been integrated with the initial randomized sampling approach for later work on point-based multi-resolution rendering described in this thesis.

Today, point-based techniques are a "hot topic" in computer graphics research. Point-based representations are now used to solve many non-rendering problems such as modeling [Zwicker et al. 2002a, Pauly et al. 2003a] or 3d photography [Matusik et al. 2002].

Overview

The thesis is structured as follows: First we describe the general background (Chapter 1). We discuss the rendering problem and related problems such as modeling of three-dimensional scenes. Additionally, we briefly discuss some background from literature on sampling techniques, especially stochastic sampling. Chapter 2 and 3 describe related work in the area of rendering complex scenes and point-based computer graphics, respectively. The description of the novel proposals starts at Chapter 4: Here, the different data structures employed for sampling are discussed. Later, different algorithms are applied to these data structures to extract suitable sample sets for rendering applications. Two main rendering strategies are considered: Chapter 5 describes forward mapping algorithms that aim at real-time rendering. They use variants of the z-buffer technique to rapidly construct images of complex scenes for perspective mappings. Chapter 6 discusses backward mapping, i.e. raytracing techniques. These techniques are more expensive. They have to be used offline but yield a higher image quality. The main goal is to create antialiased images of raytracing-based global illumination effects (including effects such as soft shadows) at reduced costs. In Chapter 7, we discuss the implementation of the different algorithms and data structures and empirical results. Chapter 8 describes some generalizations of the proposed techniques to volume rendering, out-of-core storage, sound rendering and rapid rendering of certain global illumination effects (caustics). Finally, in Chapter 9, we conclude with some ideas for future research. In the remainder of the introduction, we will summarize the main results of the different chapters and conclude with a statement of the main contributions of this thesis.

Data Structures

We will discuss two different basic data structures: The first is a *dynamic sampling* data structure that creates sample points on-the-fly by randomized sampling. This data structure is the basis of the randomized z-buffer algorithm [Wand 2000a, Wand et al. 2000b, Wand et al. 2001]. The second variant is a *static sampling* data structure, which uses precomputed sample sets. This data structure combines ideas of the randomized z-buffer technique and "Surfels" and "QSplat" [Pfister et al. 2000, Rusinkiewicz and Levoy 2000]. It can be extended to support animated scenes [Wand and Straßer 2002].

The dynamic data structure consists of a spatial hierarchy (an octree) of the scene. Each hierarchy node points to a piece of a nested *distribution list* (a list with summed area values of the primitives). The data structure is used to find groups of objects with similar spatial location. Then random sample points are chosen that are uniformly distributed on the area of the objects, according to the sampling density necessary at the spatial location of each group. Additional classification by similar orientation and similar area allows taking into account the orientation of surface fragments towards the viewer and to identify primitives (here: triangles) that receive many sample points. Such primitives can be excluded from point sampling and treated differ-

ently, typically by a more efficient standard rasterization algorithm. The data structure can be constructed in $O(n \log n)$ time for n triangles, using $O(n)$ space. Dynamic modifications (adding / removing primitives) can be performed in $O(h)$ time, with h being the height of the octree.

The static sampling data structure also employs an octree; it contains precomputed sample sets in its nodes. Large triangles are detected during preprocessing and stored at different hierarchy levels to be recognized during traversal. For the creation of sample sets, different methods are proposed. In addition to simple random sampling, different stratification algorithms can be employed that create sample sets of varying uniformity, leading to a different oversampling and thus to different rendering costs. We will perform a worst case analysis to determine upper bounds for oversampling. The average case is examined empirically. We will also derive confidence bounds, showing that the randomized sampling algorithms produce valid sample sets with high probability at low costs. Overall, static sampling data structures can be constructed in $O(hn)$ time. Using a *nested sampling* storage, $O(n)$ memory demands can be ensured. A *full sampling* approach (similar to the "Surfels" data structure), might yield superlinear memory demands, but in practice, the difference is usually not significant. The advantage of the latter organization is that one sample point will only be used at one resolution level, hence allowing for employing a prefiltering approach to point attributes in order to fight noise and aliasing artifacts [Pfister et al. 2000].

Forward Mapping Algorithms

Given a sampling data structure, real-time rendering of highly complex scenes can be performed using a *forward mapping* approach: The sampling data structures are used to extract sample sets with a maximum on-screen sample spacing. This means, the *perspective projection factor* is used as sampling density function.

The proposed data structures allow a conservative approximation of this specification: Sample sets are computed with an on-screen sample spacing that does not exceed the needed maximum (e.g. 1 pixel). A formal analysis shows that this sampling can be done efficiently: The depth component of the projection factor ("*depth factor*", i.e. perspective foreshortening of the projected area by $1/z^2$) can be approximated up to a factor of $(1+\varepsilon)$ in $O(\log \tau + h)$ time. τ denotes the *maximum relative depth range* of the scene, i.e. the ratio between the scene diameter and the near clipping plane of the viewer. Approximation means, that groups of objects are extracted from the spatial hierarchy so that the depth factor does not vary by more than $(1+\varepsilon)$. To obtain a full ε-approximation of the desired sampling density, two more effects have to be regarded: A (slight) increase of projected area towards the corners of the image and the influence of the surface orientation on the projected area. For the first factor, an efficient ε-approximation is possible. However, the second (orientation) factor cannot be handled strictly. Even using orientation classes, we cannot ensure a strict ε-approximation. However, this is not too problematic in practice: If we assume random normals with uniform distribution (on the unit sphere), only a small *average* overestimation is obtained. An additional issue is view frustum culling. As we do not need sample points for parts of the scene outside the view frustum, the approximation algorithm also performs approximate view frustum culling. Again, we cannot guarantee a formal ε-approximation of the visible projected area, but we can make sure that the cross-sectional area of the view frustum is overestimated by at most a user defined constant ε.

The analysis is valid for all variants of the sampling data structures. Thus, we obtain rendering costs of $O(sc + \log \tau + h)$, with sc being the costs for creating the sample points. The sample costs depend on the estimated projected area (this means that a coarse approximation causes more rendering costs) and on the employed sampling strategy. Stratified sampling pattern need

$O(\bar{a})$ sample points to cover an estimated projected area of $\bar{a}$. Random sampling patterns need $O(\bar{a}\log\bar{a} + \log f^{-1})$ sample points, with f being the probability that the sample sets fail to cover the image with sufficient sampling density. Additionally, sampling using dynamic sample selection causes costs of $O(\log n)$ per sample point while static sampling needs $O(1)$ time per point.

After a suitable set of sample points has been determined, an image has to be reconstructed. This process consists of two logical steps: First, the occlusion in the scene has to be resolved from the point set. Second, a continuous image has to be reconstructed from the visible points. To reconstruct the occlusion, we establish a bound for the on-screen sample spacing. Then, invisible points can be removed using a z-buffer technique. One option is to use a regular grid of z-values in screen space, project all points and keep only the closest in each grid cell. However, this leads to aliasing issues as the oversampled point clouds interfere with the regular resampling grid. A better option is to perform neighborhood splatting: Neighboring points are deleted if they are located within the sampling distance of another point with smaller depth. A small tolerance interval avoids depth dominance and cascaded deletion effects.

After determining the visible portion of the point set, a scattered data interpolation problem has to be solved to reconstruct a continuous image. Different options can be applied: simple pixel-filling, splatting, or local filtering. The latter option yields the best image quality. Alternatively, alpha-blended splats can be used with a radial Gaussian function in the alpha channel [Rusinkiewicz and Levoy 2000]. The opacity is modulated by a heuristic density estimate based on the local area around sample points. In terms of aliasing, this technique leads to suboptimal results. However, it yields good reconstructions for unstructured scenes showing subtle subpixel occlusion effects, such as a forest of trees.

Backward Mapping Algorithms

A point-based multi-resolution point hierarchy can also be used in the context of raytracing algorithms [Wand and Straßer 2003a]. Here, the goal is different from forward mapping. A multi-resolution approach only provides minor potential performance advantages as raytracing using hierarchical acceleration data structures (usually) already shows a strongly output-sensitive rendering time. The goal of the point-based multi-resolution raytracing algorithm is to devise a more efficient antialiasing scheme.

The algorithm uses extended ray cones corresponding to the area of one pixel each (modeled by screen space derivatives of ray parameters [Igehy 99]). The cross-section of the ray cones consists of ellipses with limited anisotropy. These ray cones are intersected with a point hierarchy, storing a multi-resolution point representation with prefiltered attributes. The algorithm uses points with a sampling distance corresponding to the smallest ray cone diameter. Footprint assembly [Schilling et al. 96] is performed to reconstruct a local piece of surface. For local reconstruction and compositing of multiple interactions between surface fragments and the ray cone, a modified per-ray variant of surface splatting [Zwicker et al. 2001a] is employed. The compositing quality can be improved by employing subpixel masks [Carpenter 84].

At the cost of one ray cone per pixel, the algorithm provides antialiasing as well as approximations of classic distributed ray tracing effects such as soft shadows, blurry reflections, and depth-of-field. In contrast to stochastic raytracing approaches, no noise artifacts are introduced into the images. However, this comes at the costs of being a slightly biased technique, as the point-based multi-resolution hierarchy only provides an approximation of the exact scene.

Implementation and Results

The different rendering algorithms proposed in this thesis have been implemented in the context of a rendering system for complex scenes, providing an extendible scene graph library. Special nodes allow for the integration of precomputed data structures into the scene graph, supporting hierarchical instantiation of these data structures in order to encode highly complex scenes.

Examining the implementations of the proposed techniques, we will discuss the influence of various rendering and precomputation parameters on rendering results and efficiency, allowing for optimizing these parameters empirically. In addition, we compare theoretical predictions (e.g. concerning the oversampling of different stratification techniques) with empirical measurements.

Different variants of forward mapping rendering are implemented: Reconstruction based on splatting is implemented using graphics hardware. This approach allows the visualization of highly complex scenes at real-time framerates with an image quality roughly comparable to conventional z-buffer rendering. Additionally, we will also examine more involved reconstruction techniques: Local filtering techniques yield the best antialiasing quality; alpha blending based techniques offer the best visual impression for unstructured scenes with complex occlusion. Prefiltering allows noise-free renderings even in real-time settings.

As a general result, we observe that point-based multi-resolution techniques cannot reproduce subpixel occlusion effects faithfully. Usually, the opacity is overestimated (for examples the opacity of a set of leafs of a tree mapped to a single pixel). This is the main restriction of the technique in terms of image quality. Apart from this effect, the rendering quality is comparable to conventional rendering approaches that do not employ a multi-resolution strategy.

The evaluation of the proposed backward mapping technique shows that images of highly complex scenes can be constructed with little aliasing and almost no noise artifacts. Similar to forward mapping rendering, good rendering results are obtained up to the reproduction of subpixel occlusion details. Again, the coverage is usually overestimated, leading to thickened silhouettes. Using subpixel masks, this effect can be reduced at the cost of increased rendering times.

Comparing the rendering performance with classic cone tracing, the multi-resolution approach yields a significant improvement. Point-based multi-resolution raytracing can handle highly complex scenes at moderate costs while classic cone tracing does not even terminate within measurable time (at least not if reflected or refracted cones are used). We will also compare the rendering performance with distributed raytracing, using a reference implementation based on the same code basis. It uses an adaptive sampling approach and subpixel stratification. Its rendering quality can be controlled by oversampling parameters, allowing a trade-off between noise removal and rendering speed. We examine two different scenes: A low complexity scene containing only a few sharp features and a high complexity scene showing large unstructured areas of high variance in the image. For the low complexity benchmark, the rendering times of the point-based technique are comparable to distributed raytracing, configured for a similar rendering quality (i.e. low noise level). For high variance images, the point-based rendering technique is more efficient. If the image does not contain complex silhouettes, no subpixel-masks are necessary. In this case, the performance advantage is significant. At least ten times more rendering time is needed to obtain an image with a sufficiently low noise level. If subpixel-masks are used, the performance advantage is smaller. Nevertheless, in applications where noise free rendering is more important than avoiding slightly biased images (e.g. rendering of animation sequences), the new technique may provide advantages in terms of performance. The concrete comparison depends of course heavily on the implementation and optimization. Thus, the results are only a first indicate of the relative performance.

For rendering special effects (such as soft shadows), subpixel masks have to be used in order to avoid silhouette overestimations. Then, the point-based rendering algorithm yields approximations that are qualitatively similar to distributed raytracing results, but with no noise artifacts.

Extensions

In addition to the main results, we will also discuss some extensions and applications of the point-based rendering framework. The first is multi-resolution volume rendering. This technique aims at a real-time visualization of large, regularly sampled volume data sets. The algorithmic structure is similar to point-based forward mapping rendering, but uses a slightly modified error criterion and a textured slices based renderer instead of point clouds. We will prove logarithmic running time (O(log n) time for n^3 voxels). Experiments show that very large data sets such as the well-known multi-gigabyte visible human data set [National Library of Medicine 2002] can be rendered in real-time on a conventional PC. The algorithm has been designed in cooperation with Stefan Guthe, who did also the implementation and evaluation. Thus, it is described only briefly in this thesis. For details, please refer to [Guthe et al. 2002, Guthe 2004].

The second extension is a modification to the static sampling data structure to support efficient out-of-core storage. We will discuss an externally efficient construction algorithm for large data sets and modifications to the point hierarchy to minimize external operations. A prototype implementation demonstrates the applicability of the proposed extension in practice. The technique has been designed in cooperation with Matthias Fischer, Jan Klein and Jens Krokowski from University of Paderborn, who also did the implementation and evaluation, in cooperation with coworkers. For details, see [Klein et al. 2002].

The third extension is an application to sound rendering. A similar point hierarchy as used for visual rendering of surface models can also be applied to approximate the sound caused by a large number of sound sources [Wand and Straßer 2003c]. This extension shows that the point-based multi-resolution paradigm is of use for other application areas than visual rendering.

The last extension discussed in this thesis is a real-time algorithm for rendering caustics of extended light sources. It is no multi-resolution algorithm, but it uses a dynamic discretization into surface sample points as intermediate representation for computing the global illumination effect. The idea is to compute caustics of extended light sources as an overlay of images projected by infinitesimally small surface fragments. An implementation using programmable graphics hardware allows real-time rendering of these effects, which is not possible using standard techniques such as photon tracing. Similar to all other extensions, this technique is also described only briefly, more details can be found in [Wand and Straßer 2003b].

Contributions of This Thesis

Concluding the introduction, we discuss the improvements upon the state-of-the-art (at the time of first publication) by the techniques described in this thesis:

- The randomized z-buffer algorithm (i.e. the dynamic sampling data structure described in Section 4.1 and the reconstruction techniques discussed in Chapter 5) enables rendering of highly detailed scenes with logarithmic costs (concerning the number of primitives in the scene). It allows handling of scenes with complexities such as 10^{15} triangles and more at interactive frame rates. At the time it has been

presented [Wand 2000a, Wand et al. 2000b], it has been the first algorithm that permitted an interactive visualization of such scenes with the corresponding provable guarantees for image quality and rendering time.

- The extended version of the algorithm published in [Wand et al. 2001] includes a caching strategy that allows real-time rendering. This extension has been inspired by [Pfister et al. 2000, Rusinkiewicz and Levoy 2000]. The algorithm improves on these two proposals in several ways: It provides very short preprocessing times and allows for efficient dynamic updates of the data structures. It can represent the input model at its original precision, not being limited to a maximum sampling resolution being fixed during preprocessing. The data structure has been generalized to support hierarchical instantiation, allowing an application of the algorithm to highly complex scenes.
- The main drawback of the randomized z-buffer is that it causes relatively large sampling costs. This is especially due to the fact that random sampling needs large sample sets to fight noise artifacts. This problem is circumvented in the static sampling data structure [Wand and Straßer 2002], which combines ideas of [Pfister et al. 2000, Rusinkiewicz and Levoy 2000] with the prior approach. The data structure can be generalized to support animated scenes. At the time of publication, the implementation of this approach has been the first system being able to render animated scenes such as highly complex crowd animations in real-time with the corresponding guarantees for image quality and efficiency.
- A formal analysis of the rendering costs is given, for both dynamic and static sampling, proving the efficiency of the approach for a large class of scenes. Different sampling and stratification options are analyzed, too. This includes proofs of confidence bounds for the randomized techniques to compute correct results and upper bounds for oversampling. Average oversampling is examined empirically.
- The different parameters for constructing point hierarchies are examined empirically. The observations are used to optimize the rendering time for different application scenarios (such as dynamic sampling, raytracing, hardware accelerated forward mapping).
- The point-based multi-resolution raytracing algorithm [Wand and Straßer 2003a] introduces an efficient method for creating sample sets with a sampling density corresponding to general ray densities. The algorithm is able to render images roughly comparable with images obtained by classic distributed raytracing while ensuring to avoid aliasing and noise artifacts. The performance is at least comparable to distributed raytracing, in adverse cases (high variance) it is faster than distributed raytracing if a similar noise level is demanded.
- The volume rendering system devised in cooperation with Stefan Guthe has been the first to permit real-time visualizations of highly complex data sets (such as the visible human data sets) using a conventional PC [Guthe et al. 2002].
- An efficient point-based multi-resolution rendering strategy for data sets that do not fit into main memory is proposed [Klein et al. 2002, Klein et al. 2004].
- The sound rendering algorithm [Wand and Straßer 2003c] is the first algorithm that permits real-time auralization of scenes containing a vast number of sound sources.
- The caustic rendering algorithm [Wand and Straßer 2003b] is the first algorithm permitting real-time rendering of caustics from extended area light sources.

Chapter 1
Background

In this chapter, we describe the background for this thesis. We start with a discussion of prerequisites for rendering three-dimensional scenes: Modeling of geometry, materials, and projection. Subsequently, we discuss our goal, an output-sensitive solution to the rendering problem. And finally, we summarize basic tools from literature for sampling and discretization, which are fundamental for rendering.

1.1 Rendering

The goal of this thesis is to improve the efficiency of image synthesis for highly complex three-dimensional scenes. Thus, the first thing to be discussed is the rendering task itself. In order to generate images of three-dimensional scenes, we need to define three components: First, we need a mathematical description of the scene geometry. Second, we must define the interaction of light with the elements of the scene. And third, we have to define a projection operation that maps the three-dimensional data set to a two-dimensional image that can be output to a display device. In the following, we will discuss all three topics, starting with the geometrical scene description:

1.1.1 Geometric Scene Description

The mathematical modeling of three-dimensional scenes is one of the fundamental problems in computer graphics. Thus, the last thirty years of research have led to a multitude of different approaches. A survey can be found in most textbooks on computer graphics such as [Foley et al. 96, Encarnação et al. 96, Encarnação et al. 97]. The different modeling techniques can be categorized according to several aspects. Some important aspects are:

Volume vs. surface models: Many modeling techniques restrict themselves to describing infinitesimally thin surfaces only. These are called *surface models*. This approach is often sufficient because in many scenes the relevant optical effects for image generation occur only at the surfaces of the objects in the scene. *Solid modeling* systems additionally describe the volume enclosed by a surface model. This is often needed in mechanical engineering, e.g. for simulations or computer aided manufacturing. However, usually only the surface of the objects is used for rendering. Fully *volumetric modeling* allows interaction between light and the scene at any point in

space, not only at specific surfaces. The problem of these techniques is often a large memory consumption and thus a limited precision.

Primitives: All modeling techniques use instances of a restricted class of primitive objects to describe the scene. One of the simplest surface primitives is a triangle. More elaborate primitives are for example parametric patches (such as Bezier patches), lathe objects or subdivision surfaces. There are also non-parametric techniques such as the isosurface of an implicit function. Volumetric modeling techniques frequently use basic primitives such as voxels (i.e. a grid discretization of the optical properties of space), tetrahedra, or particles to describe the scene.

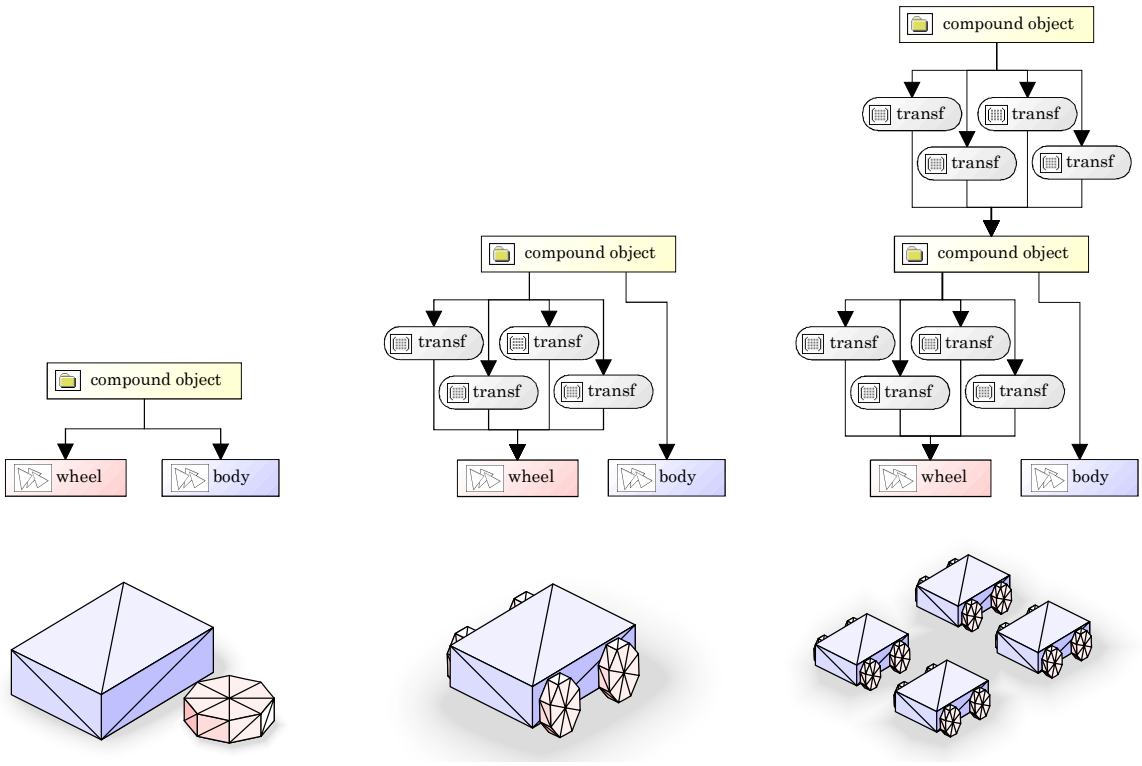
Encoding: To describe a certain scene, the primitives must be encoded in a computer-readable form. The simplest encoding is a list of parameters of primitive instances. A very common technique is e.g. the "triangle soup", i.e. one large list of individual triangles. The drawback of explicit lists of primitive parameters is the large (linear) memory demand. The constants in such a linear representation can be optimized using compression techniques [Gotsman et al. 2002], usually also enforcing a restriction of the possible data access patterns. To describe large primitive sets with sublinear memory, more implicit encoding schemes can be used such as fractal models or L-systems. Such techniques are also often categorized as *procedural models*: The primitive instances are created by the execution of an algorithm. If we allow general algorithms, we will not be able to analyze the scene for some kind of preprocessing without executing the algorithm and creating all the primitives. The reason for this is that it is not possible to analyze any non-trivial property of a general algorithm a priori using any systematic algorithmic procedure (Rice's theorem, see e.g. [Papadimitriou 94]). Thus, many techniques restrict themselves to certain predefined structures that do not offer full programmability.

Static vs. dynamic models: Up to now, we have not yet taken into account that the scene may change over time. To model dynamic scenes, several different approaches are possible: The simplest approach is to specify the evolution over time in advance. This can be done explicitly (analogous to primitive lists) by specifying a sequence of *keyframes* and interpolation rules or by more general procedural approaches, such as differential equations. The model may also depend on user interaction, i.e. the user can influence the evolution of the scene over time unpredictably (e.g. interactive editing).

Before devising a new rendering technique, we have to choose a suitable geometric representation, which will be used as input to our new algorithm. This is an important decision as the design and the efficiency of a rendering technique depend strongly on the underlying scene representation. In this thesis, our goal is to design general-purpose rendering techniques for common applications, targeting at scenes of high geometric complexity. To meet these constraints, we employ a *scene graph based* encoding of triangle lists.

The geometry of objects is described by lists of triangles. This simplifies the discussion and implementation of the proposed algorithms without loss of generality: All relevant surfaces in computer graphics can be approximated with arbitrary precision using triangle meshes. In addition, rendering of triangle-based models is efficiently supported by current graphics hardware. Thus, numerous applications in practice rely on triangle sets as geometric scene representations, making them a good candidate for a general purpose model description. Nevertheless, a generalization of the algorithms proposed in this thesis to other types of primitives should be straightforward in most cases. As an example, we will briefly discuss the generalization to volumetric models in Chapter 8.

An explicit encoding alone is not suitable for highly complex scenes: If the parameters of all triangles were stored explicitly in main memory (one large list), the size of the scene would be



(a) two objects **(b) instantiation** **(c) hierarchical instantiation**

Figure 1: Scene graph based instantiation.

restricted due to memory consumption[1]. In order to encode scenes of very high geometric complexity, we apply scene graph based encoding [Wernecke 94, Foley et al. 96]: Sets of primitives can be grouped and instantiated in the scene by specifying a pointer to the group along with transformation information describing the difference to the original instance. This technique can be carried on hierarchically: Sets of instances can again be instantiated at a higher level in the hierarchy. The references form an acyclic directed graph with transformation information in the inner nodes and primitive sets in the leave nodes. The transformations that distinguish the instances can be very general operations such as geometrical transformations and deformations or material mappings. To simplify the discussion, we will restrict ourselves to simple affine transformations described by homogeneous 4×4 matrices. Figure 1 shows an example for scene graph based encoding. A car is composed of its body and 4 instances of a wheel. Then, four cars are created by four instances of the base car, using a second (hierarchical) instantiation layer.

Scene graph based encoding provides an efficient representation for scenes that contain a certain amount of redundancy. It is possible to create a scene that encodes an exponential number of primitives with linear memory. For example, if we add another instantiation layer to the group of four cars in Figure 1c, we quadruple the scene complexity at constant memory costs. Adding more quadrupling instantiation layers creates a sequence of scenes with exponentially increasing complexity at linear memory demands. This example is of course somehow artificial. However, a reduction of memory demands is also often possible in practice: CAD models e.g. tend to use a set

[1] Note that contemporary graphics hardware is able to render up to some hundred million triangles per second [ATI 2004, nVidia 2004a]. This means that we might already expect interactive rendering times for models if a simple explicit representation of them just fits completely into main memory.

of standard components to build complex designs. In general, the large effort that is necessary to create models of three-dimensional objects from scratch usually also forces the modeler to employ some kind of instantiation to build complex scenes. However, there are cases where instantiation cannot be used efficiently. Therefore, we will also briefly consider a modification of our techniques to support out-of-core storage as an alternative in Chapter 8.

1.1.2 Shading

Up to now, we have encoded only the geometry of a scene. In addition, we also need a description of the surface appearance to render images. Thus, we need a *lighting model* (or *shading model*) that determines the color of the objects in the scene. Again, the computer graphics literature offers a wide variety of different techniques (see e.g. [Foley et al. 96, Encarnação et al. 96, Encarnação et al. 97]) ranging from simple ad-hoc solutions to photo-realistic simulations of light transport. Lighting models can be divided roughly into two groups: *Local lighting models* describe the interaction of a piece of surface (or volume) with light sources, independent of other elements in the scene. This excludes effects such as shadows, interreflection or scattering. In contrast, *global lighting models* also account for the influence of other objects in the scene to determine the lighting conditions at a point of the scene. The global propagation of light in a three-dimensional scene can be described by the "rendering equation" [Kajiya 86]: This integral equation states that the radiance exiting a point on a surface can be expressed (recursively) as the radiance emitted at that point plus the integral over all incoming radiance weighted by a function describing the reflectivity of the surface point in dependence on the incident and outgoing angles of reflection (BRDF, *bidirectional reflectance distribution function*). This model considers surface models only but it can also be extended to volumetric light transport. See e.g. [Glassner 95] for more details on global lighting models.

This thesis mainly deals with geometric multi-resolution techniques. Thus, there are no special requirements on lighting models and shading. In order to simplify the discussion, we just assume that every point on a surface has an associated shading function. It determines the color of the surface point depending on the observer position (and possibly some constant global settings such as light sources). For complexity evaluations, we assume that the shading function can be evaluated in O(1), independent of all the complexity parameters of the scene. Usually this is only possible for local shading techniques. However, some global illumination techniques such as shadow maps also fulfill these requirements.

1.1.3 Projection

The last ingredient of a rendering system is a projection operation: In order to present a three-dimensional scene, the model usually has to be projected onto a two-dimensional image[2]. Again, there are lots of choices for mapping functions, such as fish-eye rendering or orthographic projection [Foley et al. 96]. In this thesis, we always assume a *perspective projection*: The projection is described by specifying a center of projection, a camera coordinate system, a viewing angle and the aspect ratio of the image. The projection is then defined as follows (see Figure 2): A projection screen (a planar rectangle) is placed in front of the projection center, orthogonal to the viewing direction of the camera coordinate system, matching the specified viewing angle. The projection of a point in space onto the image is defined as the coordinates on the projection screen where a ray

[2] There are also three-dimensional projection techniques, such as holographic projections [Lucente and Galyean 95], but these are not commonly in use.

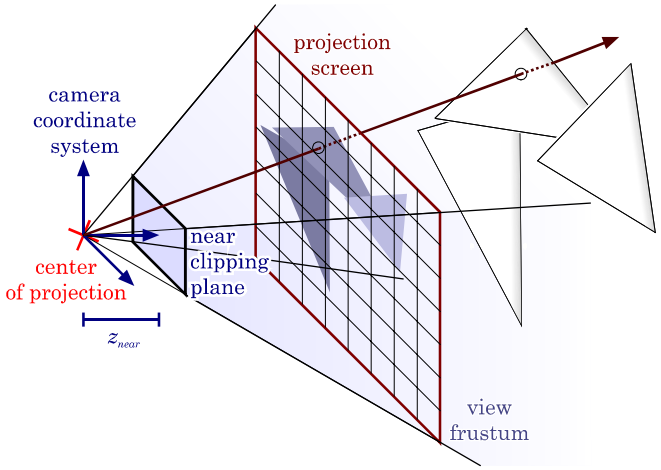


Figure 2: Perspective projection.

from the projection center to the point intersects the screen. This setup mimics the properties of an ideal physical pin-hole camera and thus is commonly used for image generation. It can be described by a homogeneous transformation matrix. The volume that is potentially projected to the screen is called the *view frustum*. In order to avoid singularities in the projection, the view frustum is usually limited by a near clipping plane, orthogonal to the viewing direction, that excludes all objects from rendering that are closer to the viewer (in viewing direction) than a user defined value z_{near}.

1.1.4 The Rendering Task

All these things taken into consideration, the complete rendering problem now can be tackled: The perspective projection defines a view ray for each coordinate on the projection screen. In the case of a surface model, we can assign a color to this point on the screen by evaluating the shading function on the first piece of surface (defined by the geometric scene description) that is intersected by the viewing ray. In the case of volume models, we have to consider the outcome of the shading function at all visible points along the viewing ray that are described by the geometric model. The corresponding radiance values have to be combined by (a numerical evaluation of) a compositing integral [Max 95a]. In either case we end up with a continuous image function that assigns a color value to each (real-valued) coordinate on the projection screen. The last (conceptual) step is to discretize the continuous image function into a set of pixels on a regular grid. This step is not trivial either: An inadequate discretization strategy can easily lead to artifacts such as noise or aliasing (structured moiré patterns in the image). We will discuss this aspect more in detail in section 1.3 in the more general context of sampling and discretization techniques.

1.2 Output-Sensitive Rendering

The goal of this thesis is to provide new techniques for rendering scenes of high geometric complexity. According to our geometric scene model, we define the geometric complexity n as the number of triangles in the scene S. These are encoded in a scene graph, using memory $enc(S)$. Usually, $enc(S) \ll n$ (many artificial test scenes employed in our evaluation use hierarchical in-

stantiation patterns with memory $enc(S) \in O(\log n)$, similar to the example described in section 1.1.1).

Rendering often has to fight complexity problems: On the one hand, there is a strong desire to display scenes of high geometric complexity, for example in order to provide a photo-realistic impression of natural scenes that contain a lot of details. On the other hand, the time for rendering should be as short as possible. In interactive and real-time applications, we even often have to deal with hard time constraints. In order to sustain a fluid interaction, a minimum frame rate (typically ≥ 20 Hz) is required. As the rendering time of many rendering algorithms depends strongly on the input complexity, the complexity of scenes that can be displayed in real-time is limited.

However, the output device for a rendered picture is usually a raster display (CRT/LCD-screen, video projector, printer). Such devices provide only a limited amount of information. That means that the output complexity of the rendering algorithm is fixed in advance. This naturally leads to the question whether it is possible to construct *output-sensitive* rendering algorithms [Sudarsky and Gotsman 96]. In general, output-sensitive algorithms are algorithms with a run time complexity that is not fully characterized by the input complexity but could also depend on the output complexity. For a rendering algorithm, we would of course like to have a running time that depends only very weakly on the input (scene) complexity because the output complexity is constant as soon as we have chosen the output device. Such algorithms could solve our complexity problems: If the dependence on the input complexity was weak enough, it would allow us to use input scenes of any complexity without a significant increase in rendering time. A lot of output-sensitive rendering approaches are known in computer graphics literature, providing varying compromises of efficiency, flexibility and output quality.

Output-sensitive rendering algorithms usually operate in two steps: First, an auxiliary data structure is precomputed. Second, images for different viewing conditions are generated (more efficiently) using the precomputed information. If the scene changes, e.g. due to editing operations, the data structure describing the scene must be updated dynamically. This leads to multiple performance criteria for the evaluation of such a rendering algorithm: The time and memory demands for preprocessing, the running time for dynamic updates and the running time for rendering an image. Output-sensitive rendering techniques are also frequently referred to as approximation algorithms: We are willing to accept inaccurate results as long as the approximation errors are small enough for a certain output resolution. The approximation error should converge to zero if we are willing to spend more and more system resources on rendering. The dependence of the running time on the output accuracy determines the convergence rate of the algorithm and thus determines the maximum resolution we will be able to achieve within a fixed amount of time.

An ideal rendering algorithm should be able to render images in O(1) time, using $O(enc(S))$ time to precompute the scene data base using $O(enc(S))$ memory, and should allow dynamic updates in O(1) time. Unfortunately, such an algorithm is not known today and there is evidence leading to the conjecture that it might not even exist (see e.g. [de Berg et al. 94]). However, we might try to come close to these idealized properties: For example, rendering and update times of $\tilde{O}(1)$ might be sufficient for many applications, and preprocessing time of $\tilde{O}(enc(S))$ is also often acceptable. Additionally, we might be willing to accept minor errors in the output image (approximation errors) or restrictions to special types of scenes in order to allow a more efficient rendering algorithm.

In the next chapter (Chapter 2), we will discuss related work on rendering complex scenes efficiently. The survey will show that a lot of specialized techniques are known that achieve

strong output-sensitivity for a multitude of special cases. This thesis will not provide the (probably) impossible ideal rendering algorithm either. However, we will develop techniques that can handle a range of cases that could not have been handled efficiently before. Before the discussion of related work, we first discuss basic discretization techniques and problems that are relevant for all rendering techniques:

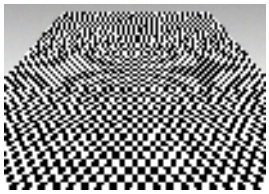
1.3 Sampling and Aliasing

1.3.1 Uniform Sampling

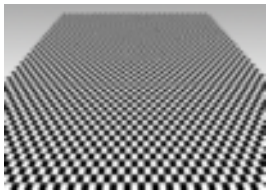
The generation of raster images requires the discretization of a continuous image function by a regular grid of pixels. This means that an image function with potentially infinite resolution is projected on a finite-dimensional subspace. For most image functions, the discrete representation is only an approximation, thus leading to a loss of information. The approximation error is inevitable, but it can become visible in the image in different forms: A typical problem is aliasing: Structured, regular patterns in the image function interfere with the regular sampling grid of the display and lead to low frequency moiré pattern in the image called *aliasing*. Figure 3 shows an example: An image of a chessboard is discretized on a low resolution pixel grid. In the area where the frequency of the chessboard pattern exceeds the raster frequency, low frequency moiré patterns become visible. These artifacts are undesirable because they distract the observer's visual system: Due to the low frequency structures, they are perceived as distinct shapes although they are not contained in the original image but are a mere artifact of the rasterization process. Therefore, we need *antialiasing* strategies to avoid the artifacts. Aliasing is a well understood problem in computer graphics. An analysis of the rasterization process in frequency space explains the reasons for the phenomenon and leads to a counter strategy. We will summarize the main results here following the exposition in [Foley et al. 96, Glassner 95]:

Let us consider an image function u. To model the rasterization process mathematically, we multiply the function by an impulse train function s consisting of equally spaced Dirac impulses (see Figure 5 for a schematic representation). This removes certain information from the original signal. In order to understand what kind of information is lost, we consider the Fourier transform of the resulting signal[3]: The Fourier transform of the impulse train is also an impulse train with spacing inversely proportional to the spacing in the spatial domain, i.e. the spacing is given by the frequency ν_s in the spatial domain. The Fourier transform of the image signal is a function that is symmetric to the y-axis and converging to zero for increasing frequencies. The multiplication of the two functions in the spatial domain corresponds to a convolution of their Fourier transformations in frequency space. This means that the spectra are replicated along the frequency axis with a spacing of ν_s, thus having a first overlap at the *Nyquest frequency* $\nu_s/2$ (Figure 5c). This has two implications: First, all frequencies in the image signal beyond the Nyquest frequency will be "mirrored" at $\nu_s/2$ and appear falsely as low frequency patterns in the image (i.e. as "aliasing"). Second, this means that an arbitrary image functions can be reconstructed exactly from a set of uniformly spaced samples if and only if the Fourier spectrum of the function does not contain any frequency components beyond the Nyquest frequency.

[3] Of course, we must assume that the Fourier integrals for the image function and the resampled function exist, but this is no restriction for computer graphics applications.

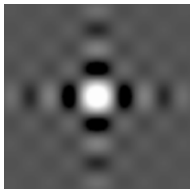


(a) rasterization of a chessboard showing aliasing artifacts

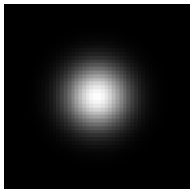


(b) rendering with antialiasing

Figure 3: Aliasing artifacts



(a) sinc (ideal low-pass filter)

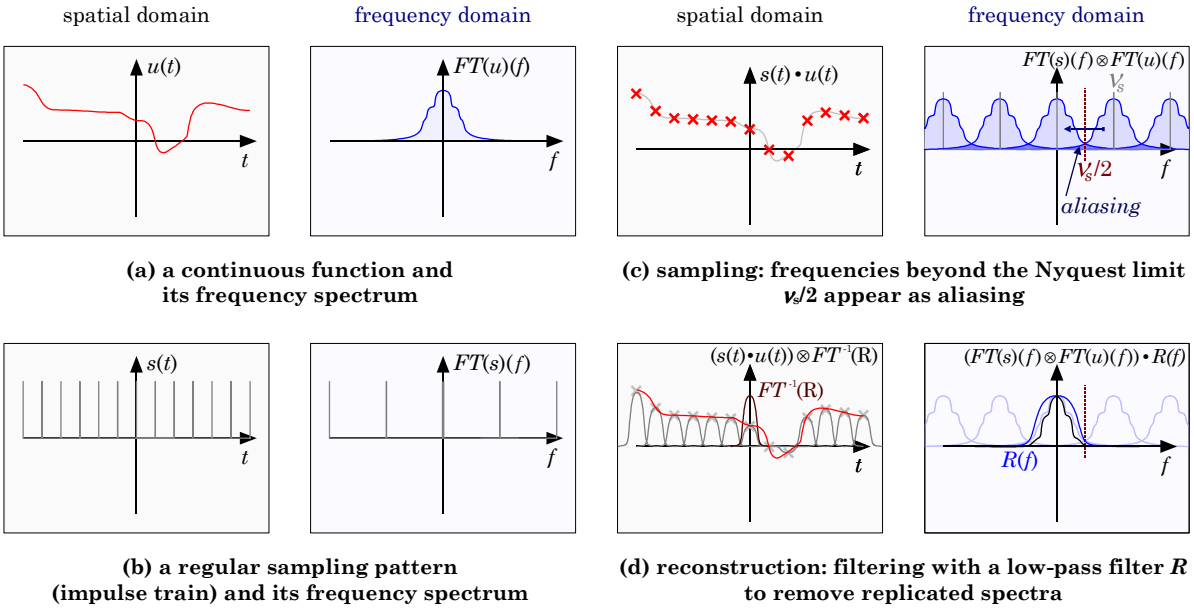


(b) Gaussian (more base band attenuation, better shape in spatial domain)

Figure 4: Convolution kernels for different low-pass filters

How can the reconstruction be performed? The sampled signal corresponds to a set of replicated spectra of the original function in frequency space. Therefore, we must remove the superfluous high frequency components in order to reconstruct the original spectrum and thus the original signal. This means we must apply a low pass filter that cancels all frequencies beyond $[-\nu_s/2, \nu_s/2]$. Thus, we have to multiply the function in frequency space by a suitable frequency attenuation function R. In the spatial domain, such a filter corresponds to a convolution operation. The convolution kernel is given by the inverse Fourier transform of the desired frequency attenuation function (Figure 5d).

What is a good choice for *sampling* and *reconstruction filters*? The obvious choice would be to use a *box filter* in frequency space that does not attenuate any frequencies below $\nu_s/2$ and does completely attenuate all frequencies above $\nu_s/2$. However, this does not lead to satisfactory results. The inverse Fourier transform of a box filter is a sinc function (i.e. a function $\sin(\varpi x)/(\varpi x)$) in the spatial domain. Thus, this "ideal" filter leads to ringing artifacts in the image. As an example, we regard sampling and reconstruction of a single Dirac impulse: The Fourier transform of a Dirac impulse is a constant spectrum. The ideal sampling filter restricts this spectrum to a box function. The sampling process creates again a constant spectrum by replicating the box function along the frequency axis. The resampling filter truncates the spectrum again to a box function. In the end, we have to consider the inverse Fourier transform of this spectrum, which is a sinc. Thus, after sampling and reconstruction with box filters in frequency space, the impulse is converted to a sinc function, which leads to ringing around the original impulse (Figure 4a). In more complex images, ringing artifacts occur similarly at all sharp borders of the image [Mitchell and Netravali 88]. An interpretation is that this filter tends to smear out the information in the spatial domain in order to preserve as much information in frequency space as possible, which leads to visual artifacts. As a consequence, we should use a filter with a better shape in the spatial domain. A typical choice in computer graphics is a Gaussian filter (Figure 4b): The Fourier transform of a Gaussian function is again a Gaussian with a support inversely proportional to the support in the spatial domain. By adjusting the support in the frequency domain, it is possible to cancel most of the aliasing while still preserving most low frequency content. This leads to image reconstructions that are a bit more blurry ("base band attenuation"), but do not show ringing artifacts. This is usually more desirable for visual rendering purposes than the behavior of the "ideal" box/sinc filter. Therefore, it is often used as a general purpose solution and we will also use this filter for sampling and reconstruction tasks in this thesis. However, it is possible to design better filters that are more optimized in respect to several characteristics for different applications, see e.g. [Mitchell and Netravali 88].



(a) a continuous function and its frequency spectrum

(c) sampling: frequencies beyond the Nyquest limit $\nu_s/2$ appear as aliasing

(b) a regular sampling pattern (impulse train) and its frequency spectrum

(d) reconstruction: filtering with a low-pass filter R to remove replicated spectra

Figure 5: Explanation of the aliasing phenomenon in frequency space [Foley et al. 96].

1.3.2 Non-Uniform Sampling

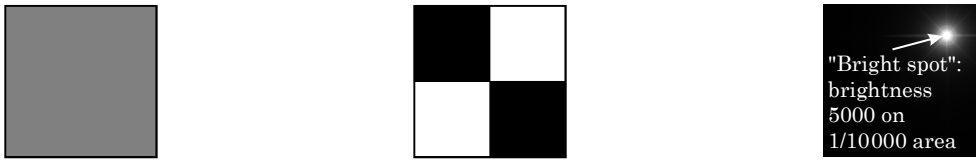
Regular sampling is not always possible or desirable: It is often not possible to obtain data samples from a function on a perfectly uniform regular grid. In this thesis, we will follow the approach of reconstructing images from surface sample points of irregular structure, independent of the screen grid. Another possible drawback of regular sampling are performance penalties: Regular sampling for example does not permit adapting the sampling density to the characteristics of the sampled function, such as taking more samples in regions of larger variation. Thus, we should also consider techniques for non-uniform sampling and reconstruction. In the following, we will summarize results from literature on two tightly related topics: Monte Carlo integration and non-uniform sampling and reconstruction of functions. The theory of Monte Carlo integration deals with the estimation of integrals from irregular, stochastically chosen samples. In more general settings, our goal is to reconstruct a function from a set of irregular sample points. In addition to numerical sampling errors, we have to deal with additional problems such as noise and aliasing artifacts in this case, too. We will summarize the main results here, for a more extensive overview, see e.g. [Glassner 95].

1.3.2.1 Monte Carlo Integration

Monte Carlo integration techniques try to evaluate the integral of a function f over a domain Ω using stochastic samples of f in Ω. The basic Monte Carlo integration technique chooses n points x_i in Ω with equal probability and estimates the integral as:

$$\int_{\Omega} f(x)\,dx \approx \frac{|\Omega|}{n}\sum_{i=1}^{n} f(x_i)$$

To analyze the error of this basic Monte Carlo estimator, we can employ the *central limit theorem* [Snedecor and Cochran 67]. It states that the average of n random variables that are identically distributed and stochastically independent with expected value μ and standard deviation σ is



(a) *zero variance* ($\mu = 0.5$, $\sigma = 0$)
Monte Carlo estimation always computes exact results.

(b) *small variance* ($\mu = 0.5$, $\sigma \approx 0.6$)
Several samples needed for a good estimate.

(c) *large variance* ($\mu = 0.5$, $\sigma \approx 70$)
Monte Carlo estimation becomes inefficient.

Figure 6: Monte Carlo integration is only efficient for functions with small variance.

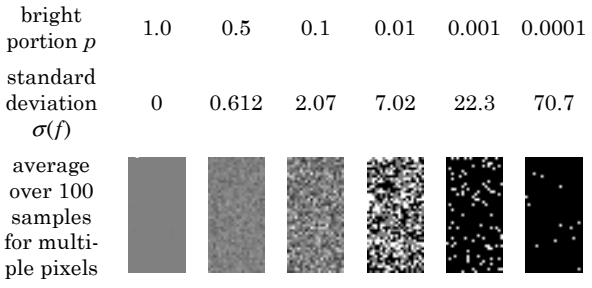


Figure 7: Monte Carlo integration ($n = 100$) over multiple pixels in which a portion p of the pixel has a brightness of $1/(2p)$ (i.e. the correct result is $\mu = 0.5$). As the non-uniformity increases, the simple Monte Carlo approach becomes less efficient.

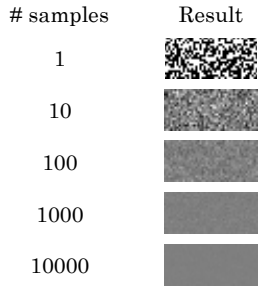


Figure 8: Monte Carlo integration over multiple pixels with varying sample size. Each pixel is half white and half black ($p = 0.5$, Figure 6b). Large sample sizes are needed to obtain a noise level below the level of perception.

asymptotically normal distributed with expected value μ and standard deviation $\sigma/\sqrt{n}$. Thus, we can expect an error (i.e. a standard deviation of the estimator) of the Monte Carlo estimation of the integrand of

$$error_n \in \mathrm{O}\left(\sigma(f)/\sqrt{n}\right)$$

where $\sigma(f)$ denotes the standard deviation of the random variable of making a random function evaluation in the domain Ω.

Monte Carlo integration is especially useful for high-dimensional domains: Conventional quadrature techniques such as Newton-Cotes or Gaussian quadrature [Köckler 94, Press et al. 95] use sampling grids of a fixed structure. This means that the computational costs increase exponentially with the dimension of the domain Ω. Thus, the quadrature becomes prohibitively expensive for high-dimensional domains even for well behaved functions f. The randomized quadrature algorithm ensures a stochastic convergence of $\mathrm{O}(\sqrt{n}^{-1})$, independent of the dimension of the base domain. Thus, Monte Carlo techniques are often used for high-dimensional integration tasks such as distributed ray-tracing [Cook et al. 84a]. The efficiency of Monte Carlo integration depends on the standard deviation of the function to be sampled. This can be illustrated by an example (see Figure 6): Imagine our task is to integrate an image function over a pixel of an image. If the function is more or less constant, the standard deviation $\sigma(f)$ will be small and the Monte Carlo estimation will deliver good results with little effort (Figure 6a). If half of the pixel is black and the other half is white (Figure 6b), we will need several sample function evaluations to converge but the Monte Carlo approach is still applicable, independent of the concrete distribution of black and white in the integration domain. However, if a small fraction of the pixel area (say 1/10,000) is very bright (say 10,000 times as bright as in the second example), we face a problem: The stan-

dard deviation of the estimator is now very high (Figure 6c). Most samples will miss the bright spot so that we need a very large sample set to attain an acceptable sampling error. Figure 7 shows an experiment: We integrate over several pixels (using $n=100$ sample points) and reduce the bright area in each pixel while increasing its brightness correspondingly. Thus, the expected value stays constant and we should obtain a neutral grey image in all cases. However, as the percentage p of bright area decreases, the noise artifacts increase. To reduce the noise in the image, we are forced to increase the number of sample points used for integration. Figure 8 shows the stochastic convergence for an increasing sample size. As the expected error decreases only with $O(\sqrt{n}^{-1})$, we need a large sample size to reduce the noise artifacts to an acceptable level. Even with 1000 sample points and a small variance of the sampled function ($\sigma \approx 0.6$), we still obtain visible noise artifacts (i.e. ≥ 1 bit deviation for an 8 bit image).

These experiments show two rules of thumb: First, we should note that Monte Carlo integration is not efficient if the relevant information (here: high function values) is not discovered by a random sample with sufficiently high probability. This is a general property of random sampling techniques: Randomized sampling allows us to obtain a good estimate of the correct result efficiently if the relevant information can be observed by a stochastic sampling process with sufficiently high probability. Second, the $O(\sqrt{n}^{-1})$ stochastic convergence leads to the effect that we are usually able to rapidly obtain a rough impression of the solution, but need a lot of additional effort to obtain a good, noise-free solution. The drawbacks that stem from these observations can be reduced by improving the basic Monte Carlo algorithm, as described in the following two subsections.

Importance Sampling

The first improvement, *importance sampling*, aims at a reduction of the standard deviation of the function used for sampling, thus reducing the error in the final result according to the central limit theorem. A big problem for Monte Carlo integration is that in many integration tasks it is highly unlikely to be able to observe the relevant regions of the integration domain by uniform probabilistic guessing. A typical example is lighting calculation: To evaluate a local illumination model, we have to integrate the incoming radiance over a hemisphere. A scene is often illuminated by a few small light sources (maybe even point lights) only. In such cases, a naive Monte Carlo integration approach will lead to unacceptable errors. To deal with such cases, we have to incorporate a priori knowledge of the problem: In the case of hemisphere sampling we know for example in which portions of the hemisphere the light sources are located. We just do not know whether they are visible and need to determine the integral value of the transported light. Thus, we would rather not perform a stochastic sampling of the complete hemisphere but concentrate the sampling on the light source area only. The generalization of this idea is known as importance sampling: The integration domain is not sampled with uniform probability but the probability for the sample generation is increased in important regions. To compensate for this bias of taking samples non-uniformly, the samples have to be weighted by the inverse of the sampling probability. This leads to the following strategy: We draw samples x_i from Ω according to a probability distribution function p: $\Omega \to \mathbb{R}^{>0}$ and estimate the integral as:

$$\int_{\Omega} f(x)\,dx \approx \frac{|\Omega|}{n} \sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)}$$

This modification changes the standard deviation from that of a uniformly sampled f to that of f/p sampled with probability density p. As a rule of thumb, the sampling density should be similar to f in order to decrease deviation: In the ideal case of p being proportional to (a positive) f, we would obtain an exact estimate at any sample size n. However, this would require computing the inte-

gral (at least for normalization) and is thus not applicable in practice. Consequently, usually a function similar to f is used: The integrand often consists of a product of two functions $f = g \cdot h$ and one of the factors is known analytically. Typical examples of such factors are BRDFs in lighting calculations or perspective foreshortening in rendering. Typical unknown parts are the surface color of a sample direction or visibility information. For many applications, importance sampling is the key to an efficient random sampling algorithm. In this thesis, we will use importance sampling by perspective foreshortening in order to obtain suitable surface sample sets for efficient rendering.

Stratification

A second improvement to basic Monte Carlo integration is stratification. This technique is usually orthogonal to importance sampling and aims at an improvement of the convergence speed. Up to now, we have chosen sample points independently from one another. This leads to a random clustering of sample points: Although the distribution of sample points will match the employed probability distribution well on a large scale, the placement of points on a micro scale will be random and non-uniform. In some cases, the convergence rate of the Monte Carlo integration can be improved by enforcing a more uniform placement of sample points in the base domain. A typical approach is "jittered sampling": The domain is divided into evenly sized and spaced areas ("*strata*", typically a regular grid) and one random sample point is taken from each. This leads to sample sets that are not stochastically independent any longer and thus void the prerequisites of the central limit theorem. This can improve the convergence rate, as analyzed in [Lee et al. 85], [Mitchell 96]: If the function f is Lipschitz-smooth within Ω, the standard deviation of the estimator is improved to $O(n^{-(1+2/d)/2})$ for a d-dimensional integration domain, using n samples created by jittering a d-dimensional regular grid. In the case of integrating over a pixel of a smooth portion of an image this leads to an $O(n^{-1})$ convergence rate instead of $O(n^{-1/2})$ for simple Monte Carlo. For functions with a k-dimensional discontinuity in a d-dimensional integration domain, Mitchell shows a standard deviation of $O(n^{(k/d-d)/2})$. For sampling discontinuity edges in images this means for example a convergence rate of $O(n^{-3/4})$, which is still a substantial improvement. These improvements can be explained by expressing the variance of the stratified estimator as a sum of the variances in the sub-strata (which are independent random variables). In the case of a single discontinuity line in two dimensions, only $\sqrt{n}$ of n samples contribute a high variance. In the case of a Lipschitz-smooth function, the variance in each strata is bounded by a constant, resulting in an $O(n^{-2})$ overall variance and thus an $O(n^{-1})$ standard deviation. The benefits of stratification vanish with increasing dimension. Stratification is not effective in the case of functions with quasi-randomly varying values in the integration domain either. In such cases, the placement of sample points does not improve the convergence: In the worst case of a purely random function f with stochastically independent values at each location in the domain, the sequence of sampled values is purely random, independent of the set of sample points it has been taken from. Thus, the convergence rate is determined by the central limit theorem again, leading to a standard deviation of $O(n^{-1/2})$.

As a rule of thumb, we can note that stratification is useful for integrating functions with little variation (smooth functions, functions with a single discontinuity). For sampling complexly structured, quasi-random functions, stratification does not improve accuracy. Additionally, we should note that the efficiency of stratification decreases with the dimension of the base domain. This is also an intuitive result: As the costs of constructing a regular base grid increase exponentially with the dimension d, the benefits of the stratification of a fixed sized sample set should vanish with increasing dimensionality.

Jittered grids are a popular but not an optimal stratification technique. It is possible to reduce the constants for the standard deviation of the Monte Carlo integrator by designing sample sets with special uniformity properties (e.g. low discrepancy patterns [Zaremba 68]). However, this does not improve the asymptotic behavior [Mitchell 96].

Other Improvements

The efficiency of Monte Carlo integration can be improved by other techniques, too: An alternative to stratification is to weight the sample points by their distance to neighboring points (i.e. the volume of the Voronoi-cells in higher dimensions). This technique can be combined with a higher order interpolation of the sample values to improve the integration accuracy for smooth functions. The technique is efficient in low dimensions of Ω only. Other improvements are adaptive sampling strategies: There are techniques that try to identify important regions of the function automatically without a priori knowledge. Other approaches can be employed to perform an adaptive sample size control that tries to estimate the quality of the current estimation and increase the sample size adaptively in case of uncertainty [Lee et al. 85]. In both cases, special care must be taken to avoid a systematic bias in the estimate. A complete discussion of Monte Carlo integration techniques is beyond the scope of this brief survey. An extended discussion of Monte Carlo techniques in the context of computer graphics can e.g. be found in [Glassner 95].

1.3.2.2 Non-Uniform Sampling and Reconstruction of Functions

Similar techniques as used for Monte Carlo integration can also be applied to estimate functions based on stochastic samples. The process consists of two conceptual steps: First, a set of sample values of a function f within a domain Ω is created. Second, an approximation to the original function f has to be reconstructed from the samples. We will now discuss these two steps more in detail.

Sampling

To provide sample values, we have to choose sample positions x_i within the domain Ω. If we use a regular sampling pattern, we may obtain aliasing in the reconstruction if the sampled function is not band limited to the Nyquest frequency of the sampling pattern (see Section 1.3.1). If we use an irregular, random sampling pattern, the approximation error will appear in the form of noise instead of structured moiré patterns ([Cook 86], [Glassner 95]). Although this does not necessarily improve the numerical approximation quality, the results from irregular sampling are often visually more pleasing. The reason for this is that low frequency aliasing structures are more likely to be perceived as distinct structures than pure noise, thus being often more distractive to the viewer.

To place the sample points, several strategies are possible. Similar to the Monte Carlo integration problem, we can use importance sampling and adaptive sampling to increase the sampling density in problematic regions of the function such as areas of high variation or large curvature. Stratification can also be used to improve the quality of the reconstruction. In contrast to simple Monte Carlo integration, where only a single value has to be computed, we must pay special attention to noise and aliasing artifacts if a complete function is to be reconstructed. We must avoid regular structures in the sampling pattern that may correlate with the structure of the sampled function, thus leading to moiré artifacts. Such problems may arise for example if a small pattern with a favorable sample point distribution is precomputed and replicated over the domain Ω on a regular grid. A second important topic is the structure of the noise that is induced by the sampling pattern itself. To analyze the structure of this noise, [Cook 86] considers the Fourier transform of the sampling pattern. For a completely random pattern, the spectrum contains

both low frequency and high frequency components. Sampling according to the sampling pattern is equivalent to a convolution in the frequency domain: Thus, noise artifacts in all frequency bands are obtained. This behavior can be changed by constraining the sampling pattern. If we avoid clusters of nearby samples, for example by requiring a minimum distance among adjacent sample points ("*Poisson disc sampling*"), we obtain a Fourier spectrum that contains only high frequency components (besides the DC value at frequency zero). Thus, we usually obtain reconstructions of higher quality because high frequency noise artifacts are less disturbing than low frequency artifacts. The effect is even more drastic as the reconstruction process performs a low pass filtering to band limit the reconstruction to the pixel grid and to remove the noise. If the sampled signal still contains considerable low-frequency noise, it will even be emphasized, leading to very undesirable results. In the case of Poisson disc sampling, most of the noise will occur in the high frequency band and this will be attenuated during the reconstruction, leading to a higher reconstruction quality.

Poisson disc sampling patterns can be constructed using a variety of techniques [Glassner 95]. A classic approach is to generate random points and reject a new point if it is too close to a former point. Other methods are e.g. number theoretic techniques that directly produce equidistributed patterns [Zaremba 68, Warnock 72] or point repulsion methods [Turk 92].

Again, it should be noted that stratified sampling can only remove noise artifacts induced by the sampling grid. As described in the context of Monte Carlo integration, noise contained in the original function to be sampled cannot be reduced by choosing an optimized sampling pattern. If we choose sample points from an unstructured, quasi-random function, the resulting sample values will be random values, independent of the structure of the sampling grid. However, in the case of reconstructing smooth functions or functions with only a few discontinuities, the reconstruction quality can be greatly improved by using an optimized sample point placement.

Reconstruction

Now we assume that we are given the values $f(x_i)$ of a function f at a set of sample positions x_i in a base domain Ω. Our task is now to reconstruct an approximation to the original function f from the sample. We can distinguish several variants of the reconstruction task: First, we can try to find a function that interpolates the sample values ("*scattered data interpolation*"). Second, if the samples are noisy, we are often interested in fitting a function with reduced degrees of freedom to the data points ("*scattered data approximation*"). Noise removal is also necessary if the sample values are exact measurements (which is typically the case in image synthesis) as the irregular sampling process itself also (usually) leads to noise artifacts. A special case of scattered data approximation is low-pass filtering: In computer graphics, we are often not interested in the function f itself but rather in a band limited version $f \otimes h$ that has been convolved with a low-pass filter h (thus also being an approximation technique). This is for example necessary to avoid aliasing if the reconstructed function should be displayed using a raster display. Low pass filtering can also be desirable to remove noise induced by the sampling process.

To perform scattered data interpolation / approximation, several algorithms are known. Some of the most common are ([Glassner 95], [Köckler 94]):

Warping: A straightforward idea for the reconstruction from non-uniform sample sets is to warp the domain by a suitable warping function to obtain a regular grid of sample points. Then, a reconstruction technique for regular sample patterns can be used to obtain the reconstructed function (as described in section 1.3.1). This yields a formal criterion for the ability of a perfect reconstruction from non-uniform sample points: If we know a warping function so that the concatenation of the original function and the inverse warping function is band limited according to

the regular grid, we can perfectly reconstruct the original function from the irregular sample points by convolution with an optimal low-pass filter on the regular sample points and subsequent inverse warping. Note that we must know the warping function in addition to the sample points as it is not apparent from the sample itself but a suitable choice depends on the spectral properties of the original function. The practical application of warping reconstruction leads to several problems: The problem of determining a suitable warping function while preserving the spectral requirements for sampling is a non-trivial problem and requires control over the sampling process along with a priori knowledge of the sampled function. For more than one-dimensional base domains Ω, the construction of a warping function becomes especially problematic: The sample points must be transformed into a multi-dimensional regular grid, which might not always be possible without violating the local topology of the original sample points. Thus, warping techniques are not commonly used in computer graphics applications.

Least-squares fit: A well known technique for approximating functions from irregular samples is least-squares fitting according to a set of basis functions. The approach of minimizing the squared distances between the approximation and the sample points leads to a linear system of equations. Weights can be used to control the fitting results, e.g. to compensate for an irregular spacing of sample points in the base domain. The problem of this approach is that the basis functions are determined independently of the sample points. This global approximation approach is often not flexible enough. To improve on this, finite element methods can be used:

Finite element methods: Instead of using a fixed function basis for the fitting, a more elaborate approach uses basis functions defined on a suitable finite element mesh in the base domain (e.g. a spline basis). The mesh can be constructed with regard to the sample points and the desired reconstruction task, thus providing more degrees of freedom. Using the functional basis on the mesh, several variational problems can be considered and solved numerically to obtain the desired approximation. Typically, a function is constructed that minimizes the distance to the sample points and the deformation energy of the resulting surface ("thin-plate splines"). The drawback of this technique is the large effort it takes to build an appropriate mesh and to solve the variational problem. Elaborate numerical techniques are needed to perform this efficiently for large numbers of input sample points.

Local filtering: Approximations with minimal deformation energy are not an optimal approach to image reconstruction problems. In this area, it is not important to guarantee smoothness properties but we must assure a band limited reconstruction of the original function from the sample set according to e.g. the pixel spacing on the screen. The most common technique for a band limited reconstruction is local filtering: We convolve the sample points with a suitable low pass filter[4]. However, this approach will produce brighter images in areas of higher sampling density. We can circumvent this problem by employing a renormalization step. We modify the filtering process by taking a *weighted average*: The value of each pixel is computed as the sum of the sample values multiplied by the weights of the filter kernel, divided by the sum of the filter weights [Cook 86]. A typical choice for the filter kernel is a truncated Gaussian (parts of the kernel with low contribution, say < 1%, are set to zero). More sophisticated filters can be used, too [Mitchell and Netravali 88]. The local filtering method with normalization works well as long as the sampling density is roughly uniform over the reconstruction domain. However, if the sampling density varies, this can lead to problems: First, we must adapt the filter size to the sampling density in order to avoid noise artifacts in sparsely sampled regions (or even singularities in the normalization term if the filter support is too small). This can for example be done by determining

[4] Note that this technique is equivalent to performing a Monte Carlo integration of the product of the function and the filter kernel for each reconstructed function value.

the radius in which the k-nearest neighbors (for a small $k \in \mathbb{N}$) are located. Second, as [Mitchell 87] points out, the reconstruction is biased at borders between different sampling densities: Areas with higher sampling density have more influence on the resulting color. To avoid this artifact, we could enforce a uniform sampling density by inserting additional sample points with average function values in low density areas. A more efficient technique is proposed in [Mitchell 87]: The signal is reconstructed on a hierarchy of grids, starting with a fine granular grid with a spacing according to the sample spacing in high density regions. First, each grid cell that contains sample points is set to the average of the contained sample points. Then the resolution is decreased successively. At each grid resolution, the reconstructed value at the grid cell is obtained by averaging higher resolution grid cells with uniform weighting. For performance reason, the usage of a simple box filter in the hierarchical process is proposed.

Hierarchical pull-push interpolation: A generalization of Mitchell's algorithm leads to the *hierarchical pull-push* algorithm [Gortler et al. 96]: First, the sample points are projected onto a fine granular regular grid, filling some of the grid cells with averages of sample values. These are obtained by approximating the samples using an appropriate set of basis functions for each grid cell. Then, in a second "*pull*" step, grids of lower granularity are filled by approximating the sample values using a lower-dimensional basis. In the third "*push*" step, the holes in the fine granular grids are filled with values from the coarser grids until all holes are closed. To avoid discontinuities, the values from different approximation levels are blended according to their "weight". The weights are given by the sum of the basis functions at the sample points for each grid cell. This algorithm is especially attractive for reconstructing functions from very large sample sets with highly irregular sampling density.

In addition to these techniques, many more methods for reconstruction from irregular sample sets are known. Again, we refer the reader to [Glassner 95] for a broader discussion of techniques for computer graphics applications.

1.3.3 Sampling Statistics

Up to now, we have discussed numerical problems of stochastic sampling techniques such as aliasing, noise and optimized sampling techniques to avoid these problems. In this thesis, we will also have to deal with discrete problems such as the question whether a random sample set will probably leave holes after a discretization to a pixel grid. To answer these questions, we need some combinatorial results that should be summarized in this section. The overall question of this section can also be posed as "How uniform are the results of a uniformly random sampling process?". The answer to this question will be crucial for the efficiency of the randomized sampling techniques proposed in this thesis.

A simple, abstract model is the bins-and-balls-model: Assume that we are given n bins and throw k balls into the bins with uniform probability. The so-called "*coupon collector's problem*" poses the question: How many balls must be thrown until every bin has received at least one ball? The expected ratio between thrown balls and the number of bins characterizes the expected non-uniformity of a uniform random distribution. [Motwani and Raghavan 95] show the following properties to answer this question:

Expected Value: The expected value for the number of balls k that have to be thrown independently with equal probability at n bins until every bin received at least one ball is

$$\mathrm{E}(k) = n \cdot H_n \tag{1}$$

where $H_n := \sum_{k=1}^{n} 1/k = \ln n + \mathrm{O}(1)$ is the nth harmonic number.

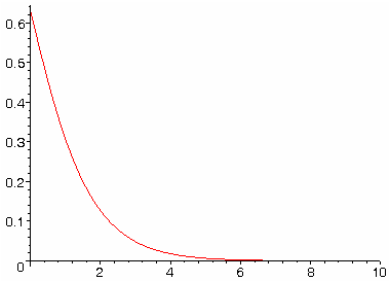


Figure 9: Plot of $1-e^{-e^{-c}}$

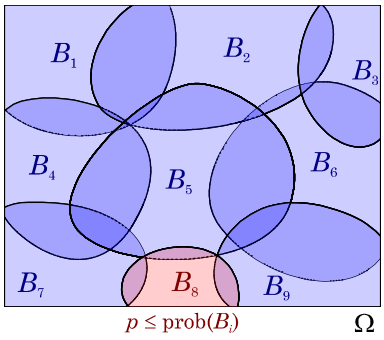


Figure 10: Overlapping bins. p is a lower bound for the probability of each bin receiving a sample point.

Asymptotically sharp threshold: Let X denote the random variable that yields the number of balls that have to be thrown to obtain at least one ball into every bin. Then, for every $c \in \mathbb{R}$, we obtain:

$$\lim_{n\to\infty} \text{Prob}[X > n\ln n + cn] = 1 - e^{-e^{-c}} \qquad (2)$$

These results explain the difference between filling n bins deterministically and stochastically: The expected value for the number of balls that has to be thrown randomly until each bin has received a ball is about $n \cdot \ln n$. This means that we expect a logarithmic overhead for filling all bins by random selection. This overhead grows only slowly with an increasing number of bins. Thus, it might be acceptable for many applications. However, a small expected value would be of little use if we had to expect a large variation. The second result shows that this is very unlikely, at least for a large number of bins. The probability for a larger relative deviation cn drops rapidly (see Figure 9 for a plot of the bound).

As a rule of thumb, we can note that a random uniform distribution among n entities leads to some uniformity that grows weakly with the number of entities. The factor by which a random visit of all entities is more expensive than a deterministic visit is about $\ln n$.

In many applications, we are interested in ensuring to fill all bins with a given confidence: We want to compute the number of balls that are necessary to fill all bins with at least a given probability s. An upper bound for the minimum number of balls can be established by an elementary analysis. We will make assumptions that are slightly more general, as needed later in the thesis:

We assume that the n bins B_i are subsets of a base set Ω, not necessarily disjoint (later, Ω will typically be a subset of $\mathbb{R}^2$ or $\mathbb{R}^3$). Then, we draw points $p \in \Omega$, independently of one another. We assume that the probability p_i for $p \in B_i$ is larger than a global constant p for all $i = 1\ldots n$ (Figure 10). Now, we want to determine a bound for the number k of points that have to be drawn until each bin B_i has received at least one point.

Let $E_{i,j}$ be the event that bin B_i receives a point in the j-th round (i.e. when the j-th point is drawn). Then obviously $\text{Prob}(E_{i,j}) \ge p$ and $\text{Prob}(\overline{E_{i,j}}) \le 1-p$. Now let N_i be the event of B_i not receiving any point after k rounds. As all points are drawn independently of one another in each round, we obtain:

$$\text{Prob}(N_i) = \prod_{i=1}^{k} \text{Prob}(\overline{E_{i,j}}) \le (1-p)^k$$

Now let $N := \bigcup_{i=1}^{n} N_i$ be the event that at least one of the bins does not receive a point. This is the event we are trying to avoid. The events N_i are not independent of one another. However, we can still give an upper bound for the probability Prob(N). For the union of any sets of events, not necessarily independent, we know that the probability cannot exceed the sum of the individual probabilities, thus we obtain:

$$\mathrm{Prob}(N) = \mathrm{Prob}\left(\bigcup_{i=1}^{n} N_i\right) \le \sum_{i=1}^{n} \mathrm{Prob}(N_i) \le n(1-p)^k$$

If we want to ensure that Prob(N) ≤ $1 - s$ (i.e. all bins receive a point with at least a probability of s), we can solve for k and obtain:

$$k \ge \frac{\ln n - \ln(1-s)}{-\ln(1-p)} \tag{3}$$

In our applications, the bins B_i usually form a roughly uniform partition of Ω into n bins, possibly with a small overlap. In that situation we have $p \in \mathrm{O}(n^{-1})$ so that we can express p as $p = c/n$. Additionally, n is usually very large (such as the number of pixels on the screen). Therefore, $1 - p$ is very close to 1 and we can approximate the logarithm in the denominator by the Taylor expansion $\ln(x) \approx x - 1$. Using these assumptions we obtain the estimate

$$k \ge \frac{n}{c}\left(\ln n - \ln(1-s)\right) \tag{4}$$

$$= \frac{n}{c}\left(\ln n + \ln(f^{-1})\right) \tag{5}$$

for large n with $f := (1 - s)^{-1}$ being the probability that we fail to fill all bins. The *oversampling* factor, i.e. the ratio between the number of bins and the number of rounds, is at most the natural logarithm of the number of bins n plus the natural logarithm of the inverse of the probability to fail filling all bins (divided by the average overlap factor c in case of overlapping bins). Therefore, the number k of points that have to be drawn from Ω to fill all n bins is in $\mathrm{O}(n \log n)$ for any given constant confidence level s. In conclusion, we see that random sampling for visiting/filling a partition of uniformly sized bins is still efficient if we require a fixed probability for the success of the procedure.

Chapter 2
Rendering Techniques

In this chapter, we summarize the main algorithmic approaches known from literature to obtain output-sensitive running times for rendering complex scenes. The rendering techniques are divided into two conceptual approaches: Forward mapping (projection) and backward mapping (raytracing) algorithms, which both need different algorithms and data structures for an efficient implementation.

2.1 Classification

To create projections of three-dimensional scenes, two different basic strategies are known: *forward mapping* and *backward mapping*. Forward mapping strategies take primitives from the scene data base and project them onto the screen. Then, visibility is resolved and invisible portions are discarded while visible portions are drawn. Backward mapping strategies start in the image plane and directly search for the objects that are visible within each pixel. Thus, they perform visibility calculation and projection in one step. In this thesis, we will discuss both forward and backward mapping techniques that make use of point-based multi-resolution data structures to accelerate the computation. The techniques are variants of the two predominant implementations of the mapping techniques: z-buffer rendering and raytracing. A basic implementation of one of these two techniques is not output-sensitive and thus not suitable for handling complex scenes. However, a lot of improvements are possible to accelerate the rendering. The following subsection summarizes these techniques for forward mapping algorithms, the next one those for backward mapping.

2.2 Forward Mapping

2.2.1 The z-Buffer Algorithm

The most popular[5] forward mapping algorithm is the *z-buffer algorithm* [Catmull 74, Straßer 74]: It uses two buffers: A color buffer that will contain the rendered image afterwards and an auxiliary z-buffer that stores scalar depth values. It is initialized with the largest representable depth value. The algorithm projects all primitives onto the screen and rasterizes them, i.e. it determines the pixels covered by the primitive. For each such pixel the depth of the primitive at that point is determined and compared to the depth buffer entry of the pixel. If the point on the primitive is closer than the depth buffer entry, the color and the depth buffer entries for the pixel are overwritten with the values of the primitive. In the other case, the corresponding portion of the primitive is invisible and will be ignored.

The z-buffer algorithm is simple to implement and very efficient for scenes of moderate complexity: Obviously, the rendering time is $O(n+a)$ [Heckbert and Garland 94], where n is the number of primitives (usually triangles) and a is the projected area (number of pixels) of the primitives on the screen, including hidden areas. The algorithm can be implemented very efficiently in hardware: Current graphics accelerator boards for PCs are capable of processing up to $3 \cdot 10^8$ triangles per second (accounting for the complexity parameter n) and up to $2 \cdot 10^9$ pixels per second (accounting for parameter a) [ATI 2004, nVidia 2004a]. Due to this enormous processing power, the algorithm is currently the predominant rendering technique in interactive applications.

An important variant of the z-buffer algorithm is the *a-buffer algorithm* [Carpenter 84], which adds support for transparency and edge antialiasing to the original z-buffer method: For each pixel, the a-buffer stores a list of surface fragments that provide a color and z values along with a subpixel mask and an optional transparency value. The subpixel mask is a simple bitmask that represents the subpixel coverage of the fragment within the pixel. To obtain the final image, each fragment list has to be sorted. Then, the color values are determined by compositing the fragments according to their transparency values and subpixel coverage. Theoretically, the algorithm is similarly efficient as the z-buffer algorithm (we may lose a logarithmic factor due to sorting). Nevertheless, it is currently rarely used in real-time applications. The problem is that hardware implementations of this technique are much more involved due to the need of sorting and dynamic memory management for the fragment lists. Thus, no commodity hardware implementations are currently available[6]. A typical workaround implemented in current applications is the use of z-buffer hardware with brute-force oversampling for antialiasing and, in a second step, alpha-blending of depth sorted primitives for rendering partially transparent objects.

2.2.2 Limitations

Despite the efficient implementation, the z-buffer algorithm is still a linear time algorithm. This means that scenes with very large complexity parameters n or a cannot be handled in real-time. In such cases, the algorithm has to be modified: One or more filtering algorithms can be employed prior to z-buffer rendering to reduce the complexity of the original scene description. Two differ-

[5] There are many other forward mapping algorithms, which are mostly based on depth sorting, see e.g. [Foley et al. 96] for a survey. However, due to the high performance hardware implementations of the z-buffer algorithm, they are nowadays of minor importance.

[6] There are some implementations in research systems, see e.g. [Schilling and Straßer 93], [Winner et al. 97].

ent, orthogonal strategies can be identified: *simplification algorithms* try to reduce the geometric complexity n of the scene, for example by reducing the amount of detail for portions of the scene that are far away from the viewer. *Occlusion culling* algorithms try to exclude all primitives from the rendering process that are invisible because they are occluded by other objects in the scene (or outside the view frustum, "*view frustum culling*"). A third direction for speeding up the rendering process is image-based rendering: These techniques replace parts of the scene by precomputed images. This solves both occlusion and simplification problems. However, we now need a strategy to provide suitable images that act as replacements.

The basic architecture for a hierarchical filter strategy has already been described by [Clark 76]: Clark proposes describing the scene using a hierarchical tree structure. The leaf nodes contain the geometry while the inner nodes e.g. might contain successively simplified versions or image-based replacements. The hierarchical structure also allows for efficient culling of invisible parts of the scene. Whole subtrees can be removed by checking whether a bounding box of the geometry intersects with the view frustum. Similarly, hierarchical occlusion culling can be implemented. This hierarchical architecture is still found today in most systems for visualizing complex scenes. Since Clark's rather abstract architectural proposal, a lot of algorithms have been developed to perform simplification, culling and image-based replacements. In the following, we will discuss related work from all three directions.

2.2.3 Simplification

The basic idea of simplification algorithms is to reduce the accuracy (and thus the costs) of the representation in situations where this reduction is not (or only slightly) visible. For example, it might not be necessary to display a model with a million triangles when it covers only 5 pixels on the screen. In that situation, a rough approximation of the original model with a lower geometric complexity should be sufficient. A lot of different strategies have been published that automatically adapt the level-of-detail to the presentation requirements. Good surveys of level-of-detail techniques can be found e.g. in [Heckbert and Garland 97, Puppo and Scopigno 97, Garland 99, Luebke et al. 2003]. In this section, we will summarize the basic concepts.

2.2.3.1 Manual Level-Of-Detail

An obvious simplification strategy is to model multiple versions of the same object and switch among them according to the distance to the viewer. However, this approach has several drawbacks: First, the manual preparation of models of different level-of-detail is expensive as it means a lot of extra work for the human modeler. Second, switching among different discrete levels of detail can lead to "popping artifacts" at the point of transition. These artifacts can be reduced by blending between adjacent levels-of-detail, either in image space (transparency) or geometrically (mesh-morphing, see e.g. [Alexa 2002]). Additionally, the simple level-of-detail technique only permits replacing entire objects globally. It is not possible to use different levels of detail within one and the same object, e.g. for a large terrain model that is seen at strongly varying viewing distances at the same time.

2.2.3.2 Mesh Simplification

In the early days of computer graphics, the capabilities of real-time 3d-rendering hardware were very limited. Thus, the extra work of manual level-of-detail generation was acceptable. Nowadays, it is not unusual to employ highly-detailed models with some hundred thousand primitives in real-time rendering applications. Besides the increased capabilities of rendering hardware, the

development of automatic 3d-scanning devices (see e.g. [Levoy et al. 2000]) has also motivated the development of automatic mesh simplification methods. These scanning devices usually produce regularly sampled data that is often highly oversampled in smooth regions of the scanned object. Given these developments, a manual preparation of levels-of-detail has definitively become too expensive. Thus, automatic simplification strategies have been developed during about the last ten years, providing a large variety of algorithms.

Strategies that construct optimal approximations in a strict sense are not feasible in practice (see [Heckbert and Garland 97] for details): While the optimal approximation of curves can be done in polynomial time, it can be shown that the problem of approximating convex surfaces using polygons in an L_∞-optimal sense is NP-hard. This renders an optimal approximation of more general shapes like arbitrary triangle meshes also impractical. There are no efficient optimal algorithms known for other error metrics either. Thus, all practical algorithms in computer graphics rely on heuristics to optimize the appearance of an object using a small number of primitives. The techniques can be roughly divided into two classes, depending on the input they are able to handle: parametric techniques and techniques for more general meshes. In addition, both classes of techniques need an appropriate error criterion to guide the simplification process.

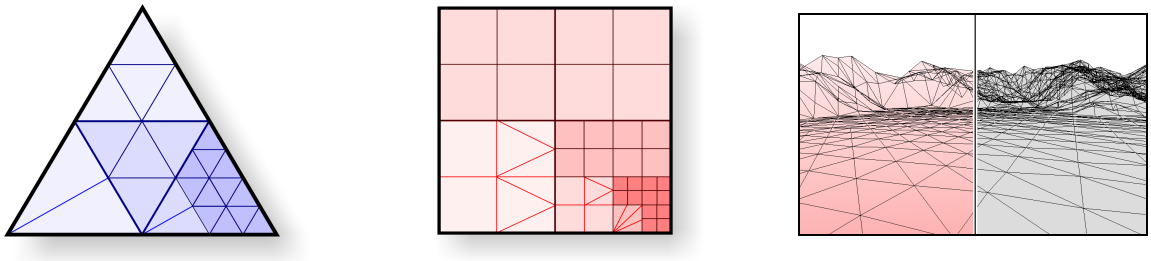
Error Measurement

The goal of an error measure for simplification is to quantify the deviation between an original model and its approximation. Most error measures describe the distance between the original and the simplified surface. Choosing a suitable error measure is usually a trade-off between accuracy and speed. Rigorous geometric error measures such as the Hausdorff distance [Klein et al. 96] or the integral L_1, L_2 or L_∞ distance between the original and the simplified surface are expensive to compute. Thus, heuristic error criteria are often employed, such as a nearest neighbor distance between a point on the surface and the original surface. An important heuristic error criterion that is often used in practice is the "*quadric error metrics*" technique [Ronfard and Rossignac 96, Garland and Heckbert 97]: It measures the deviation of a vertex in a simplified triangle mesh from the original surface by summing the quadric distances to the planes of the original triangles. This technique is especially efficient for estimating the cumulative effect of multiple simplification operations and offers good results for smooth surfaces. However, for rough surfaces, the error is usually strongly overestimated.

In addition to measuring the geometric distortion, it is often also necessary to take into account surface attributes such as color, surface parameterizations (e.g. texture coordinates [Schilling and Klein 98]), normals, appearance under lighting [Klein et al. 98b], or discontinuity edges. Again, several strategies can be implemented. A common technique is to incorporate attribute differences in the geometric error measure by applying the geometric measure to a higher-dimensional attribute space that contains additional dimensions for various surface attributes, in addition to spatial information [Garland and Heckbert 97]. However, more involved criteria have also been proposed in order to allow a stricter control, e.g. to penalize distortions at sharp features and attribute discontinuities [Hoppe 96].

Parametric Techniques

The first class of simplification methods is that of parametric level-of-detail techniques. These methods require a parameterization of the surface to be simplified. In the simplest case, the parameter domain must be a square in $\mathbb{R}^2$. More elaborate techniques also allow a more general parameter domain, e.g. a base mesh of triangular or rectangular patches. Parametric techniques are usually employed to display surfaces that are described by parametric functions (e.g. spline patches, subdivision surfaces, fractals, or regularly sampled data such as height fields or dis-



(a) triangular subdivision **(b) quadrilateral subdivision** **(c) heightfield simplified with quadrilateral subdivision connectivity meshes (right: original)**

Figure 11: Subdivision connectivity meshes.

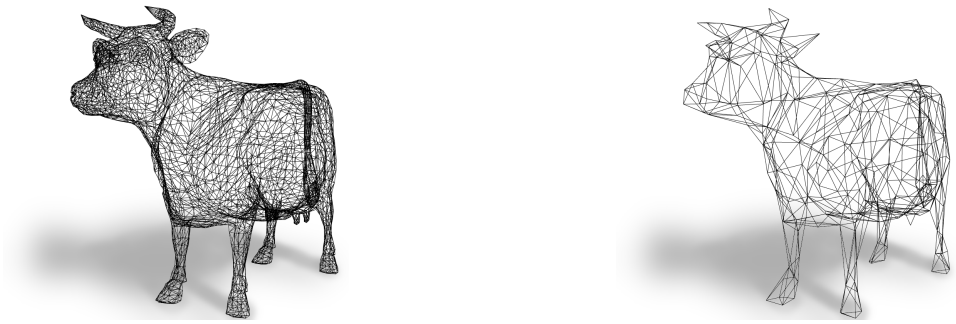
placement maps). In order to apply parametric techniques to general triangle meshes, a parameterization has to be computed before the simplification procedure.

Parametric simplification subdivides the parameter space into elements (usually triangles) according to some tessellation strategy. The granularity of the tessellation is guided by some error criterion that enforces a denser tessellation in regions of higher importance (i.e. for example closer to the viewer, larger surface curvature or roughness). Common tessellation strategies are regular grids, subdivision connectivity meshes and unstructured meshes:

Regular grids: The subdivision of the parameter domain into a regular grid is a very simple technique that does not allow a local adaptation of the sampling density. Nevertheless, such techniques are frequently used due to the ease of implementation.

Subdivision connectivity meshes: These meshes offer more flexibility: They start with a parameter domain that is composed of a base mesh of polygonal elements (usually triangles or rectangles). Then, each element is divided into subelements of similar shape (Figure 11). This process can be carried on recursively, yielding a subdivision hierarchy. The main problem is now to ensure continuity on the borders of elements that have different subdivision depth. This is fulfilled by constructing a *conformal mesh*, i.e. a mesh where no vertices are adjacent to (non-subdivided) edges. Several strategies are known to construct conformal meshes: One possibility is to design a subdivision scheme that always ensures the construction of conformal meshes, such as the Rivara bisection scheme [Rivara 84]. Another possibility is the construction of an unconstrained subdivision hierarchy followed by a *balancing* step [de Berg et al. 97]. In the balanced hierarchy, sibling nodes in the hierarchy are forced to differ only by one hierarchy level in their subdivision depth. This is enforced by subdividing nodes adjacent to more deeply subdivided siblings. It can be shown that this increases the number of nodes only by at most a constant factor [de Berg et al. 97.]. In the balanced hierarchy, each edge may be adjacent to a subdivision element of the same depth or with one more subdivision level. Thus, only a small number of cases are possible for the tessellation of each cell (8 for triangle meshes, 16 for quadrilateral meshes) so that the cases can be handled explicitly.

Unstructured meshes: Unstructured meshes are computed by performing a more general subdivision of the base domain. Typical examples are techniques based on Delaunay triangulations. For example [Klein et al. 98a] insert and remove points into and from the base domain dynamically to sustain a point density according to the accuracy requirements. A tessellation into triangles is then obtained using an incremental Delaunay triangulation. Unstructured mesh techniques overcome the main drawback of subdivision connectivity meshes where the parameter domain is always subdivided equally in all dimensions. This leads to isotropic mesh structures that do not always allow an optimal adaptation, as e.g. in the case of a cylindrical object that



(a) original model (5,804 triangles) **(b) simplified mesh (500 triangles)**

Figure 12: Example of mesh simplification using edge collapses and quadric error metrics. Simplified using QSlim 2.0 [Garland and Heckbert 97]. Model from [Garland 2003].

shows a large curvature in one direction only. However, the implementation of unstructured techniques is usually more involved and the runtime costs may be larger.

More General Input Meshes

The second class of simplification strategies comprises those that directly operate on meshes of primitives and do not require an a priori parameterization. This is convenient as many models in practical applications are given as a collection of primitives ("triangle soups") rather than parametric functions. On the one hand, this is due to software interface problems: Modeling software is often not able to interoperate at a high level of description. Thus, models consisting of higher order geometric primitives such as NURB-patches are nevertheless interchanged as simple triangle soups. On the other hand, there are a lot of models that are inherently non-parametric, such as implicit surfaces or models from certain types of 3d-scanning devices. The general mesh simplification algorithms can be categorized roughly into three subclasses: clustering, decimation and parameterization algorithms:

Clustering: A very simple simplification strategy is vertex clustering [Rossignac and Borrel 93]: A regular grid is superimposed on the mesh and all vertices that fall into the same grid cell are collapsed to one vertex. Afterwards, degenerated triangles are deleted. The method usually provides only a crude approximation quality in comparison to more involved simplification strategies. However, it is very fast, easy to implement and allows for topological simplification. The algorithm has also been used for an efficient out-of-core simplification of large data sets that do not fit into main memory [Lindstrom 2000]. A similar idea is used by [He et al. 95]: They voxelize the geometry into a regular grid, using low-pass filters to avoid aliasing. Then, a marching cubes algorithm [Lorensen and Cline 87] is used to extract a simplified model.

Decimation: A more flexible mesh simplification can be achieved by repeatedly applying local decimation operations to the mesh. [Heckbert and Garland 97] distinguish vertex, edge, triangle, and patch decimation techniques that repeatedly delete the corresponding primitive type from the mesh. Most decimation algorithms follow the same greedy approach: The mesh elements are tested for whether they can be removed without violating the topology. Then, all candidates for removal are inserted into a priority queue sorted by the error the removal would introduce. Finally, the candidates causing the least error are removed from the queue iteratively and the error estimates of neighboring candidates are updated if necessary.

The first decimation technique was proposed by [Schroeder et al. 92]. The algorithm performs a greedy vertex removal and tries to re-triangulate the holes caused by the removal. If this is not possible without using an additional vertex, it is excluded from simplification. The error of

a vertex removal is quantified by the distance to an average plane. [Hoppe et al. 93] propose a mesh simplification based on a heuristic optimization of an "energy function" that describes the deviation from the original mesh. Three different local mesh modifications (edge collapse and split, and edge swap) are applied to the mesh in a random descent search algorithm. After mesh modifications, the vertex positions are optimized using an iterative optimization technique. These two steps are iterated until a satisfactory minimum for the energy function is found. As the technique permits more complex simplification procedures than the simple greedy algorithms, it produces results of high quality. However, the runtime costs are very high, too. In later work, Hoppe proposes using only greedy edge collapse operations as this provides a better quality/runtime trade-off [Hoppe 96]. The proposed "progressive mesh" technique uses a complex error criterion accounting for sharp features and a variety of surface attributes in addition to geometric error measurement. Hoppe also proposes using the history of edge collapses in inverse direction ("vertex splits") to provide progressive transmission. Applying the operations in both directions allows a dynamic adaptation of the level-of-detail. The technique has been extended to topological simplification by allowing the collapse of vertices that are close but not connected by an edge [Popovic and Hoppe 97, Garland and Heckbert 97]. Today, most simplification techniques are based on greedy edge collapses as this technique has proven to produce good results while being quite simple to implement.

Parameterization: Parametric techniques can also be applied to general meshes if a parameterization is computed for the mesh. [Eck et al. 95] construct approximate triangular subdivision connectivity meshes for triangle meshes of arbitrary topology. They propose a wavelet-based encoding of the displacements to the base mesh. This allows the usage of wavelet-based approximation techniques, as known from image compression, to be applied to simplification. A drawback of this approach is that the subdivision connectivity mesh can only approximate the original mesh, thus retaining a small error even at high resolutions. Additionally, the approximation using a wavelet basis aims at the reproduction of smooth functions and thus does not optimally preserve sharp features. Since the paper of Eck et al., many other parameterization algorithms have been published that can (also) be used for surface simplification. See e.g. the recent papers of [Gu et al. 2002, Khodakovsky et al. 2003, Praun et al. 2003] for a survey.

Other techniques: In addition to decimation and parameterization techniques, there are also some approaches based on different paradigms. [Varshney 94] proposes a method with guaranteed L_∞ error bounds: The algorithm constructs two enveloping offset surfaces and uses a heuristic technique to construct an approximating mesh within the envelope (the construction of an optimal solution is shown to be NP-hard). However, the algorithm has high computational costs [Heckbert and Garland 97]. Thus, it is not well-suited for larger meshes. A different idea is proposed by [Turk 92]: His algorithm uses repulsive forces on the surface to compute an equidistribution of vertices that is then triangulated to obtain the simplified surface. This technique is especially interesting for point-based computer graphics as it can also be applied to compute well-distributed sample point sets for surface models [Pauly et al. 2002]. Another option for improving the quality of simplified meshes is the application of bump maps or normal maps [Cohen et al. 98] that improve the shading of the simplified surfaces.

2.2.3.3 Multi-Resolution Representations

Mesh simplification algorithms allow the automatic creation of discrete levels-of-detail. This can be used to create simplified models and to switch among them according to some metric such as the viewing distance. However, this is not flexible enough for many applications: We would like to adapt the level-of-detail in a fine granular way, especially using multiple levels-of-detail within

one and the same object, e.g. to display objects that are seen under a large depth range, such as terrain models.

Many simplification techniques can be extended to construct *multi-resolution representations* that support these requirements. A multi-resolution representation allows the dynamic extraction of different levels-of-detail from a single data structure. The proposed techniques vary in generality and flexibility. Some techniques support only the extraction of discrete levels of detail while others support a more or less arbitrary specification of the complexity of the representation ("*continuous level-of-detail*"). Many techniques also allow the extraction of varying levels-of-detail for different parts of the model. Some techniques permit a very general adaptation (e.g. [Klein et al. 98a]) while others are restricted to predefined regions (e.g. octree cells) or isotropic subdivision patterns.

Most parametric simplification techniques provide multi-resolution inherently. Subdivision-based techniques (such as [Eck et al. 95]) can easily be implemented dynamically: The subdivision depth in different regions is adapted dynamically to the application requirements. The Delaunay-triangulation-based technique of [Klein et al. 98a] has also been constructed explicitly to provide dynamic control over the level-of-detail in different regions. The simple grid-based techniques allow a trivial multi-resolution usage. However, they do not support the extraction of non-uniform levels-of-detail.

Mesh simplification techniques based on mesh element decimation also allow for flexible multi-resolution usage: First, the subsequent removal of the mesh elements (vertices, edges, triangles) is recorded. For every removal operation, we can also consider the inverse re-insertion operation. Thus, after preprocessing, we can apply these removal and re-insertion operations dynamically to obtain a fine-granular control over the current level-of-detail. This technique has been proposed by Hoppe ("*progressive meshes*", [Hoppe 96]) in the context of edge-collapse/vertex-split operations. To obtain different levels-of-detail in different regions of one and the same mesh, we could basically just choose the refinement operations for the corresponding regions. However, there is a pitfall: The decimation operations are not independent of one another. For example, a vertex split can only be performed if the corresponding vertex already exists in the mesh, which might require a series of additional vertex splits. [Puppo and Scopigno 97] analyze this problem in a general framework (a solution for progressive meshes is also given in [Hoppe 97]): They show that the repeated application of decimation operations leads to a hierarchy (more precisely to an acyclic directed graph) of decimation operations. The edges in the graph correspond to the vertex dependencies and the total order (for the acyclic graph) is due to the order of decimation operations. Levels-of-detail can be extracted from this graph by extracting an *upper set* of the graph, performing all refinement operations starting from the root node until the desired accuracy is met. The paper also gives several formal criteria concerning the adaptivity and efficiency of this process that can be used for guiding the construction process.

For an efficient implementation of multi-resolution mesh simplification data structures, some additional problems have to be solved: For efficient rendering, the fine granular level-of-detail control of decimation hierarchies is not always desirable. The overhead for managing the decimation and refinement operations can easily annihilate the gains in rendering speed. Similar problems occur if very large meshes should be processed that have to be stored in secondary memory (hard disc). In these settings, the execution speed is dominated by latency times for random access. Thus, it is not efficient to perform only little work (e.g. only one edge split) at each random access operation. Instead, it is desirable to batch mesh modification operations. Recently, this has been done by decomposing the mesh into parts using an octree hierarchy. Then each node in the octree contains a similar amount of geometry. For inner nodes, representations of constant

complexity are constructed using mesh simplification. Continuity at the borders between octree cells can be guaranteed by generalizations of subdivision connectivity techniques [de Berg et al. 97] or suitable rendering techniques at the borders [Guthe et al. 2003].

2.2.3.4 Animated Scenes

Only a few mesh simplification and multi-resolution rendering techniques are able to handle animated scenes. [Friedrich et al. 98] propose the generalization of a parametric simplification technique: They describe an algorithm for the interpolation of keyframe hierarchies based on the Rivara bisection scheme. [Shamir et al. 2000] propose a general technique to modify multi-resolution mesh data structures to handle animated data sets. Only a few mesh simplification algorithms for animated data sets have been proposed yet. This is probably due to the complexity of retaining topological constraints over both time and scale. In this thesis, we will present a point-based multi-resolution approach to simplify animated meshes. Point clouds put fewer constraints on the simplification procedure than meshes. Thus, it is easier to devise a stable simplification technique based on point representations.

2.2.4 Image-Based Rendering

A different approach to dealing with complexity is *image-based rendering*. Instead of using sets of geometric primitives at different levels-of-detail, the appearance of three-dimensional models is represented by images of the object.

2.2.4.1 Purely Image-Based Rendering Techniques

Image-based rendering techniques rely on a sampled representation of the "*plenoptic function*" [Adelson and Bergen 91, McMillan and Bishop 95]. The plenoptic function assigns a color value to each viewing ray in a certain domain. The base domain of this function is five-dimensional (sometimes higher dimensions are given if time or wavelength dependence is modeled explicitly). Thus, a systematic sampling of this function poses a considerable challenge for a space-efficient representation.

The (probably) most image-centered rendering technique is the technique known as lightfield [Levoy and Hanrahan 96] or lumigraph [Gortler et al. 96] rendering: Both variants of the data structures represent the radiance along rays emanating from an object under fixed lighting conditions. The observer is assumed to be located outside the convex hull of the object. Thus, the position of the viewer along each ray is not relevant. This reduces the dimensionality of the data to four dimensions. The rays are parameterized by their intersection with two parallel planes (for a full view from all directions, multiple planes are used). The radiance values are then stored in a (large) four-dimensional array indexed by the intersection coordinates with the two planes. The drawback of lightfield/lumigraph approaches is the large storage overhead resulting from the high-dimensional base domain and the regular discretization. Thus, compression techniques must be used to reduce the space requirements. [Levoy and Hanrahan 96] use vector quantization techniques. Other methods known from image compression (such as wavelet-based compression, see e.g. [Peter and Straßer 2001]) can be applied, too.

2.2.4.2 Hybrid Techniques

The high dimensionality and the resulting large memory requirements restrict the usage of purely image-based representations. For a fixed image and parallax resolution of $O(1/n)$, we obtain storage requirements of at least $\Omega(n^4)$. Thus, it is impossible to represent complex scenes at a

high level-of-detail that would allow a strong variation of the viewing distance. To overcome these limitations, a lot of methods employ a hybrid approach that uses both geometric and image information.

The classic example of such a combination is texture mapping: Images are mapped onto surfaces (via a surface parameterization) to provide additional details [Catmull 74, Heckbert 86, Heckbert 89]. Textures can supply various information, such as color values, material parameters (glossiness, reflectivity etc.), transparency, surface roughness or even surface displacements that augment the local geometry [Blinn 78a, Blinn 78b, Cook 84b, Gardner 85].

A variant of texture maps is the environment map [Blinn and Newell 76]: All radiance values from different direction incident at a point of the scene are stored in a texture map: The texture map usually consists of a parameterization of a simple, closed, convex surface such as a sphere or cube. Each point on the surface represents a possible incident direction, and a color sample is stored at the parameter coordinates of that point. Environment maps are used to simulate global lighting effects such as reflection or refraction. Various surface characteristics can be simulated by prefiltering the environment map [Heidrich and Seidel 99]. Environment maps can also be used to integrate synthesized objects into real world scenes [Debevec 98]. An environment map constructed from photographs serves as an (approximate) light source to illuminate the scene with natural light. These techniques are for example used for computer generated special effects in movie productions where a close match of the lighting conditions is vital for a plausible effect.

Modern texture mapping systems permit the specification of procedural shader scripts that can combine several texture mapping strategies to define the surface appearance of objects [Cook 84b, Cook et al. 87]. These techniques are nowadays also supported in graphics hardware for triangle rasterization [Lindholm et al. 2001, ATI 2004, nVidia 2004a].

The idea of texture mapping can also be combined with the concepts from lightfield rendering: *Surface lightfields* [Wood et al. 2000] store the outgoing radiance at discrete sample positions on the surface of an object to describe the appearance under fixed lighting conditions. A similar approach is view-dependent texture mapping [Debevec et al. 96]: A rough geometry model is augmented with detail textures that are picked from several photographs of a scene, selecting the best matching ones for each textured surface. There are also techniques that try to acquire the surface reflectance properties from images of a scene in order to allow a display under arbitrary lighting conditions [Yu et al. 99, Matusik et al. 2002]. However, this is in general a hard inverse problem.

Another class of hybrid geometry and image-based rendering techniques are image warping approaches [Chen and Williams 93, McMillan and Bishop 95, Shade et al. 98]: Each pixel in a set of input images is assigned a position in space. This can be done for example by estimating the depth from stereo parallax information in natural scenes or z-buffer readbacks in synthetic scenes. Then, new views of the scene from different viewpoints can be generated by reprojecting ("warping") the sample points from the images according to the new view position. These techniques rely on a sample point discretization of the scene and were among the predecessors of point-based rendering techniques similar to those described in this thesis (see Section 3.2.1 for a more detailed discussion).

2.2.4.3 Rendering of Complex Scenes

Image-based rendering techniques can be applied to speed up the rendering of complex scenes. The most obvious application is the usage of texture mapping techniques already in the modeling

phase in order to avoid the construction of complex geometry. This approach is very commonly used and is the key to high-quality rendering both in real-time (e.g. computer games) and offline rendering applications. However, hand-crafted textures and shader scripts do not help if we already have a very complex model that should be simplified in order to allow a real-time display.

Automatic image-based simplification strategies have been proposed by many authors. The general idea is to substitute rendered image information ("*imposters*") for geometric objects to reduce rendering costs. [Regan and Pose 94] propose substituting far field geometry with environment maps: Objects with a distance larger than r are rendered onto a cube that is displayed in subsequent frames using texture mapping. This technique is extended by using multiple cube maps with exponentially increasing spacing around the current view position. Transparency information in the texture maps is used for compositing. The cube maps are updated when the viewer moves with a frequency inversely proportional to their diameter. This allows avoiding rendering of far off geometry at every frame as its depiction probably will not change significantly for small relative movements. A similar idea is proposed by [Torborg and Kajiya 96]: They describe the "Talisman" rendering architecture that allows a user-controlled replacement of geometry by warped images. [Shade et al. 96] propose a hierarchical replacement scheme that replaces nodes in a BSP-tree of the scene by warped textures that are updated dynamically depending on an error metric. A hierarchical scheme with precomputed textured imposters is proposed in [Maciel and Shirley 95].

In order to understand the potential benefits of dynamic image-based replacements, it is useful to do a formal analysis of the costs in a simplified model [Wand 2000a]: We assume that the scene is a flat disc with radius R, containing $\Theta(n)$ uniformly distributed objects. This is a rough model of a typical situation in many applications where cities or landscapes are to be displayed. We also assume a very simple error metric for our replacement strategy: An image of an object can be reused as long as neither the parallax nor the scaling factor under which it is seen exceeds a certain threshold. This criterion means that we must update any image containing the object when we have made a *relative movement* of more than a constant ε. The relative movement is defined as a movement of the viewpoint by a certain distance divided by its proximity d to the object. Thus, we obtain an update frequency of $O(1/d)$ for the objects with a distance d. Now, we assume that the viewer performs a small movement in the center of the scene. Then we can estimate an average update frequency for the objects in the scene:

$$f_{av-upd} = \frac{1}{\pi R^2} \int_{O(1)}^{R} \frac{rendering\_costs(r)}{r} dr = \frac{1}{\pi R^2} \int_{O(1)}^{R} \frac{O(r)}{r} dr = O(R^{-1}) = O(\sqrt{n})$$

The average update rate (and thus the rendering costs of a linear time rendering algorithm) grows proportionally to the square root of the number of objects in the scene (for objects uniformly distributed in a disc-like scene). The constant depends on the parallax errors that we are willing to tolerate. As parallax errors are the main artifact of image-based replacement strategies, this result can be considered a lower bound for the asymptotic runtime gains. Note that image-based replacement algorithms also cause additional compositing costs for displaying the "cached" content that is not accounted for in this consideration.

The strategy of [Regan and Pose 94] already achieves the lower bound: It rerenders all objects located between two cube maps if the parallax error of these objects becomes too large. Thus, we obtain average rerendering costs of

$$\sum_{i=1}^{O(\log r)} O\Big(\underbrace{(c^i)^2}_{\substack{\#\text{Objects for} \\ \text{cubemap i}}} \cdot \underbrace{c^{-i}}_{\substack{\text{Update-} \\ \text{frequency}}} \Big) = \sum_{i=O(1)}^{\lceil \log_c r \rceil} O\left(c^i\right) = O(r) = O\left(\sqrt{n}\right)$$

(c = spacing factor by which the cubemap radius increases)

Additionally, we have O(log n) costs for every frame (independent of the viewer's motion) due to compositing the cube maps. Thus, this strategy is already (nearly) asymptotically optimal in terms of our simple cost model. However, the later approaches such as the technique of [Shade et al. 96] allow a more adaptive placement of replacement textures. Therefore, we can expect a strong reduction of actual compositing costs in practice.

The main problems of texture mapping based imposters are parallax errors: The update costs could be reduced drastically if we were able to use the same image-based replacement for a larger range of viewing angles. Therefore, a lot of techniques have been proposed in order to compensate for the parallax errors of image-based imposters: [Sillion et al. 97] use textured meshes created from depth images as far field imposters in urban scenes. [Mark et al. 97] propose using image warping to recalculate views after small movements of the viewpoint to speed up rendering of complex scenes. The warped images may contain holes in regions not visible in the reference image. Therefore, the authors warp multiple nearby reference images to fill the holes. [Rafferty et al. 98] apply a similar technique to replace far field geometry behind portals in architectural scenes. The hole-filling problem can be circumvented by storing multiple depth values for every pixel [Max and Ohsaki 95b, Shade et al. 96], leading to a point-based rendering approach (see Section 3.2.2 for an in-depth discussion). In addition to point clouds, stacks of partially transparent textures can also be used as imposters ("*layered imposters*", [Schaufler 98]). This leads to similar effects and may be more efficient on hardware platforms with low point reprojection but high texture rendering speed.

2.2.4.4 Animated Scenes

Image-based replacements have also been used to speed up the rendering of complex animated scenes. [Tecchia and Chrysanthou 2001] describe a rendering technique for complex crowd animations such as a large group of humans walking through a virtual city. In a preprocessing step, images of the animated entities are rendered into textures from several viewing directions and for several timesteps of the animation. During rendering, the closest time step and viewing direction is determined and the corresponding texture is rendered. The method leads to very fast rendering times. However, the discretization of time and viewing angle causes parallax and continuity errors. This does not matter for a far field approximation, but the method cannot be used to simplify single, large animated objects of high complexity. [Aubel et al. 2000] describe a dynamic image caching algorithm for crowd animations. The image of a rendered person is reused as texture over several frames. The speed-up of this method is limited as coarsening the time discretization too much will result in jerky motion.

2.2.5 Occlusion Culling

A forward mapping rendering algorithm such as the z-buffer technique has to process all objects in the scene independently of whether they are visible or not. The basic algorithm is not capable of excluding hidden objects by itself, although they have no influence on the output image. Thus, in order to display scenes with much occlusion efficiently, a filter algorithm needs to be applied before rendering to exclude invisible objects. This *occlusion culling* problem is orthogonal to the

simplification problem described in the preceding sections: Simplified objects with little geometric detail might still cover large areas in the image plane that have to be rasterized and lead to considerable costs. Occlusion culling algorithms can be roughly classified into two groups of techniques: From-point visibility techniques and from-area visibility techniques. In the following, we will give a brief overview of published techniques, for a more detailed discussion, see a survey paper such as [Cohen-Or et al. 2001].

2.2.5.1 From-Point Visibility

From-point visibility algorithms are given camera parameters and a scene data base. Their task is to exclude invisible objects from rendering. Usually, this is done in a conservative manner, i.e. the algorithm potentially returns a superset of the visible (*PVS, potentially visible set*) objects and a final z-buffer rendering step (or a similar technique) is used to resolve the exact per-pixel visibility. It is possible to do exact, analytical from-point visibility computation. However, this is an $\Omega(n^2)$ problem and thus not suitable for complex scenes [Foley et al. 96]. The z-buffer algorithm computes a solution to the discretized from point-visibility problem (visibility is resolved up to a pixel) in linear time. Therefore, an occlusion filtering algorithm should have output-sensitive, sub-linear run-time requirements to be able to speed up the computation. Most output-sensitive from-point visibility algorithms follow the same algorithmic paradigm, *front-to-back rendering*:

The scene data base is traversed in front-to-back direction, picking groups of objects with smaller z-distance earlier. The objects are projected onto the screen and dropped if the corresponding space is already occupied. The algorithms vary in the implementation of this paradigm: How is the depth-sorting achieved? How are groups of objects identified that are potentially occluded? (Grouping is important for sub-linear running time.) How is the occupation of the screen-space represented?

One subclass of strategies consists of *portal rendering* algorithms [Luebke and Georges 95]: The scene is divided into cells that are connected to each other via *portals*. This connection is represented as an adjacency graph (see e.g. [Haumont et al. 2003] for an automatic construction algorithm). The rendering algorithm traverses this graph starting from the cell that contains the viewpoint. During traversal, the set of visited portals is projected to the screen and the intersection of the portal area is calculated incrementally. When the intersection becomes empty, the traversal is stopped. To perform these intersection calculations efficiently and numerically stable, the portals are usually approximated by a simple representation such as screen-axis-aligned bounding boxes. Portal rendering algorithms show two main drawbacks: First, the subdivision into cells and portals is not feasible for arbitrary scenes. Usually, portal rendering is applied to architectural models. Second, the efficiency of the culling algorithm can be bad in adverse cases: Depending on the traversal strategy, the algorithm might need time superlinear in the number of visible cells: A depth-first traversal could visit one and the same cell multiple times (via different portal sequences). A breadth-first traversal visits every cell only once. However, it must store sets of portals and intersect them with new sets of portals so that the complexity of the intersection calculations can grow during traversal.

A second subclass consists of *hierarchical* front-to-back rendering strategies [Greene et al. 93]: The scene is organized in a spatial hierarchy such as an octree. This hierarchy is traversed top-down and front-to-back. At each visited node, a test is performed against the screen content that has been drawn so far. If the node is occluded, the traversal is stopped and the complete subtree is culled. If the node is visible, the traversal is continued. In the case of leave nodes, the associated geometry is drawn on the screen. The main issue is again the representation of the occluding geometry that has already been drawn on the screen: [Greene et al. 93] suggest the usage of a

multi-resolution pyramid of z-values that allows testing the visibility of a bounding box of a node in the hierarchy without always rasterizing all of its pixels. The drawback of this approach is that of the update costs for the z-pyramid after drawing a set of primitives. An alternative is a brute-force testing of all z-values in the frame buffer [Bartz et al. 99]. As this technique is supported by modern graphics cards in hardware with enormous rasterization speed, it is a viable alternative to a software implementation of a hierarchical z-buffer. Additionally, several optimizations are possible to reduce the costs of the brute-force approach [Staneker et al. 2003]. Recent graphics cards even optimize the rasterization internally by employing a hardware implementation of Greene's z-pyramid (see e.g. ATI's "Hyper-z" technique, [ATI 2004]). Thus, the different approaches are probably converging.

2.2.5.2 From-Region Visibility

From-region algorithms are based on the determination of the parts of the scene that are invisible from extended regions. Usually, this information is computed during a preprocessing stage: [Teller and Séquin 91] precompute the visibility for all cells in a cell and portal graph. During rendering, a simple access to a list of visible cells is sufficient to obtain a potentially visible set. A portal rendering approach is then applied to this PVS in order to further reduce the set of objects to be rendered. A similar approach is used in the "Quake" rendering engine, which is the basis of several popular computer games [Abrash 97]. The technique is refined by using a general BSP-tree for the automatic generation of cells instead of the k-d-tree proposed by Teller and Séquin. From-region visibility can also be used dynamically: A potentially visible set for a small neighborhood of the current viewpoint is estimated and can be reused over several frames.

The major technical problem is the computation of cell-to-cell visibility information. A conservative solution is equivalent to determining whether a single ray between to cells exists that is not occluded by a geometric primitive in between the two cells. The space of all rays connecting two regions in three-space is four-dimensional. This makes a discretization of the problem much harder than for the two-dimensional from-point visibility problem. An exact solution is possible [Teller and Hohmeyer 93, Nirenstein et al. 2002], but leads to considerable computational costs. A stochastic solution using random raytracing can also be applied [Gotsman et al. 99]. However, this approach cannot ensure to compute a correct solution. With a certain probability, visible cells may be missed and not included in the PVS. Current research focuses on approximation algorithms that guarantee a conservative estimate of the PVS [Durand et al. 2000, Schaufler et al. 2000, Leyvand et al. 2003].

2.3 Backward Mapping

In comparison with forward mapping rendering strategies, backward mapping solves the visibility problem just in the opposite direction: Instead of processing all primitives and depth-sorting them in screen space after projection, backward mapping performs a search in the scene database to find the objects that are visible in each pixel of the image.

2.3.1 The Raytracing Algorithm

The backward mapping paradigm directly leads to the raytracing algorithm: A ray from the center of projection through the center of every pixel is created and the scene data base is searched for the first intersection with a primitive [Appel 68]. The technique can be generalized to handle ideally reflective or transparent objects by applying the algorithm recursively [Kay 79, Whitted

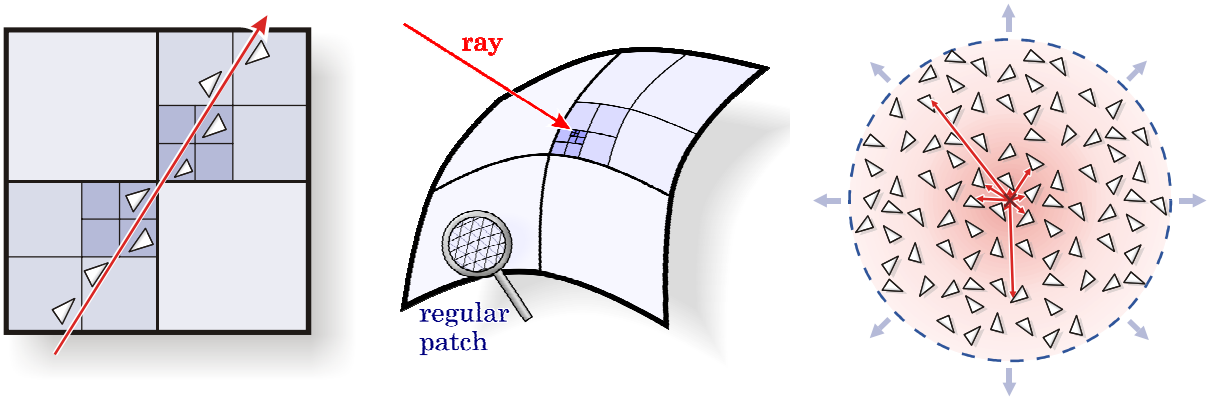
80]: At intersection points, *secondary rays* are shot into the scene to calculate the light contribution due to reflection or refraction. Shadows are calculated by shooting a ray towards a light source and testing for any intersection on the way. It is also possible to simulate more complex effects using stochastic ray sampling: The *distributed raytracing* algorithm [Cook 86] shoots multiple rays through each pixel stochastically and computes the average color to obtain an antialiased image. To simulate glossy reflections or transmissions, the outgoing directions of secondary rays are altered randomly. Soft shadows can be computed by jittering the ray position on an area light source stochastically, and even depth-of-field effects can be obtained by a stochastic modification of the parameters of the primary rays.

An important advantage of the raytracing algorithm is its flexibility: In contrast to forward mapping algorithms, raytracing is able to simulate global lighting phenomena such as shadows or reflections easily. Especially stochastic raytracing techniques can be generalized further to obtain fully featured simulations of global light propagation; see e.g. [Jensen et al. 2003] for a survey. A general drawback of backward mapping algorithms is a performance penalty: In contrast to forward mapping techniques, the algorithm is forced to search the visible primitives for each pixel. A naive implementation (test all primitives for each pixel) without recursive tracing leads to a time complexity of $O(v \cdot n)$ for n primitives and v pixels on the screen, which is always worse than the complexity of a z-buffer renderer. However, a raytracer usually uses spatial data structures to accelerate the ray queries. This changes the run-time behavior of raytracing drastically and makes a performance comparison with forward-mapping techniques less obvious, as discussed in the next subsection.

2.3.2 Data Structures for Efficient Ray Queries

The acceleration of raytracing is a classic computer science problem: We are given a large data base of geometric objects and we rapidly want to find an object that meets a certain formal requirement. In this case, we need to find that primitive (if there exists one) that leads to the first intersection with a query ray.

The classic solution to this problem is the usage of spatial subdivision structures: The simplest solution is to use a regular grid in space and store a list of all primitives that intersect a grid cell in each such cell [Fujimoto et al. 1986]. Then, only the grid cells intersecting with the ray have to be searched for primitives. This technique is quite efficient in many cases. However, it is not well-suited for general scenes. If the distribution of the objects in space is very uneven (which is very likely to be the case; note that we are usually dealing with hollow surface models), the method wastes storage and computation time by handling empty boxes. Thus, the subdivision techniques are usually applied hierarchically: The scene is divided into a hierarchy of bounding volumes such as an octree [Glassner 84] or more general bounding volume hierarchies [Rubin and Whitted 80]. Then, a ray query is performed using hierarchical tests against the bounding volume hierarchy. There are also other approaches such as 5-dimensional ray-space subdivision (see [Arvo and Kirk 91] for a broad survey of raytracing acceleration techniques). Ray query efficiency has also been investigated in computational geometry literature: [de Berg et al. 94] construct a ray query data structure that ensures to answer ray queries in $O(\log n)$ time. However, they need $O(n^{4+\varepsilon})$ space and precomputation time. [Szirmay-Kalos and Márton 98] show a lower bound of $\Omega(\log n)$ for any ray query algorithm and $\Omega(n^4)$ space requirement bounds for any $O(\log n)$ ray query data structure within the algebraic decision tree model. Therefore, such raytracing acceleration techniques with optimal query time are not suitable for practical applications.



(a) worst case linear running time: a ray passes nearby a set of triangles, intersecting all their bounding boxes

(b) regular tessellations of smooth patches: logarithmic running time

(c) dense scenes: expected running time is constant

Figure 13: Different scene models lead to different performance characterizations for raytracing with spatial hierarchies as acceleration data structure.

In practice, the predominant acceleration techniques today are variants of hierarchical spatial subdivision methods (octree, k-d tree etc.). How efficient are these acceleration techniques? It is easy to see that hierarchical spatial subdivision methods have a worst case $\Omega(n)$ bound for scenes of n primitives (Figure 13a). However, these worst case scenarios are not commonly found in practice. Thus, we must make assumptions about the expected input model to find a more realistic complexity characterization. [Szirmay-Kalos and Márton 98] assume a scene with uniformly distributed objects and show an O(1) expected ray query time for octrees (after a O(log n) location of the viewpoint). Their argument is quite intuitive: They assume a scene populated with objects of similar size, with uniform density (Figure 13c). Now the scene is enlarged (more objects are added, the diameter increases) without changing the object density and the complexity is considered in dependence on the number of objects. They show that the expected free length on a ray before it hits an object is finite for infinitely growing scenes. If the ray queries are implemented by a neighborhood search in an octree, the costs are (at most) proportional to the free length of the ray divided by the leave node size (which is constant in this model). Thus, they obtain O(1) costs.

However, this analysis is based on a very restricted scene model. [Wald et al. 2001] report an empirical O(log n) complexity measured for a terrain height field with different sampling resolutions. This observation can be explained by a simple model: First, we assume that our scene consists of a flat square, tessellated uniformly into primitives. The employed hierarchical decomposition of the scene will result in a tree with a structure comparable to a quadtree on the object: On each hierarchy level, the number of objects in the subtree will be reduced by a constant factor on the average. Thus, the localization of an intersection with a ray orthogonal to the surface will require O(log n) time (see Figure 13b). A similar running time can be expected for many scenes that consist of tessellated surfaces seen under a normal angle that is not too large.

In conclusion, we can note that a quantification of the efficiency of raytracing techniques strongly depends on the scene model. The worst case bound is linear (leading to a non-output-sensitive rendering algorithm). However, there are strongly sub-linear bounds for certain scene models that may come closer to practical applications. In addition to the asymptotic costs, it is also important to consider the absolute costs of raytracing. The traversal of a spatial hierarchy and the corresponding intersection tests need a considerable number of computations. A highly

optimized implementation can yield interactive framerates even for complex models [Wald et al. 2001]. However, a distributed implementation using a cluster of several high-end PCs is necessary to provide real-time frame rates at high image resolutions or for rendering complex secondary ray effects [Wald et al. 2003].

2.3.3 Antialiasing

The classic raytracing algorithm performs point sampling in image space. Thus, we have to deal with aliasing issues (see Section 1.3). The rendering literature provides two classes of approaches to solve the aliasing problems: *Supersampling* and *extended rays*.

2.3.3.1 Supersampling

As described in Section 1.3.2.2, the reconstruction of an aliasing-free image leads to a numerical integration problem: For each pixel, we have to compute a weighted average of the continuous image function in its neighborhood. This can be done elegantly using Monte Carlo integration techniques that shoot multiple random rays for each pixel and determine the pixel color as a weighted average [Cook 86]. There are also regular supersampling techniques [Whitted 80]; however, they are more susceptible to structured aliasing. In the worst case, stochastic integration techniques (as well as deterministic techniques applied to a worst case quasi-random image signal) have a convergence speed of only $O(n^{-1/2})$, n being the number of rays evaluated for each pixel. Thus, supersampling techniques can be quite expensive in adverse cases. However, improvements such as adaptive sampling and stratification techniques can improve the convergence rate in many cases (see Section 1.3.2 for details). A big advantage is the flexibility of the stochastic approach: It is easy to incorporate advanced global illumination effects into the stochastic raytracing framework [Cook 86].

2.3.3.2 Extended Rays

A conceptual alternative to point sampling techniques are methods that use extended ray volumes for intersection calculations: The *cone tracing* algorithm [Amanatides 84] uses cones with circular cross-section. It starts with cones with the apex at the viewpoint and a cross-section corresponding to a pixel in the image plane. Then the algorithm determines all objects that are intersected by the cones and uses an area based compositing strategy to blend together the color contributions of the intersected object fragments. As the algorithm is carried on recursively, the shape of the cones is adjusted according to surface curvature, surface roughness and light source size in order to perform antialiasing of secondary rays and to approximate soft shadows and glossy reflections. A similar idea, the *beam tracing* algorithm, has been proposed by [Heckbert and Hanrahan 84]: The algorithm starts with a pyramid ("beam") corresponding to the whole screen and successively clips it to planar surfaces. At reflecting and refracting surfaces, secondary "beams" are sent into the scene to calculate the corresponding global illumination effects. These techniques have been refined by other authors later on [Kirk 87, Ghanzanfarpour and Hasenfratz 98]. However, ray tracing with extended ray volumes is used only rarely today. The main drawback of these techniques is the complexity of the intersection calculations: In complex scenes, it is very likely that the extended ray volumes (especially secondary rays) intersect with a large number of primitives, leading to prohibitively large running times.

As a compromise, hybrid techniques have been suggested that employ extended ray volumes only to guide the evaluation of a traditional raytracer: Extended ray volumes can for example be used to estimate the variance in the image by detecting borders or regions with large color

variation [Amanatides 96, Genetti et al. 98]. Then, a traditional stochastic integration approach is used to determine the pixel color. [Igehy 99] introduces the concept of *ray differentials*: The first order derivative of the ray position and direction in respect to the screen coordinates are considered to estimate the *ray footprint*. This is a first order approximation to the volume covered by all rays through a single pixel. The ray footprint can then be used to (for example) guide mipmap selection for anti-aliased texture mapping. The technique is especially appealing due to its conceptual simplicity, improving on previous proposals such as [Shinya et al. 87, Collins 92]. A similar technique is used in [Schilling 2001] in the context of antialiased environment mapping. The application of hybrid techniques to improve conventional stochastic ray tracing techniques is still an active area of research. Ray differentials techniques have for example been used recently to guide an on-demand tessellation strategy for highly complex scenes consisting of higher order primitives [Christensen et al. 2003]. A further recent proposal is the usage of polygonal multiresolution models to speed up raytracing [Karabassi et al. 2003]. A "ray disparity", corresponding to the ray spacing, is computed for the rays intersecting objects of the scene. According to this measure, different levels-of-detail are selected. The "ray disparity" is computed analytically for primary rays only. For secondary rays, a heuristic degradation measure is used, taking into account attenuation and recursion depth.

Chapter 3
Point-Based Multi-Resolution Rendering

In this chapter, we will give an overview of point-based multi-resolution rendering techniques. We will start with a motivation, arguing that the point-based rendering paradigm can complement shortcomings of former output-sensitive rendering techniques. In the second subsection, we will describe related work in point-based rendering, including many recent developments that were published concurrently with the techniques that are subject of this thesis. The third subsection concludes with an overview of the methods proposed in this thesis and the structure of their exposition.

3.1 Limitations of Previous Techniques

The rendering techniques discussed in the preceding section are able to render many classes of highly complex scenes at interactive framerates. However, there are still problematic cases that cannot be handled efficiently. Important examples are natural scenes like trees, grass, forests or entire landscapes. When the work on the methods proposed in this thesis was started, an interactive rendering of such outdoor scenes at high quality was not possible (at least with reasonable efforts in hardware, such as a current PC) without strong simplifications such as e.g. simple textured imposters for trees or using non-interactive rendering times. One of our main motivations was to devise techniques to render such scenes efficiently.

The problem of natural landscape scenes is that they consist of a large number of individual objects, forming the scene by their assembly. This creates surfaces of highly irregular structure and complex topology of the corresponding primitive meshes. The same problems also occur in related cases, such as crowd scenes with large crowds of humans or animals. Mesh simplification algorithms have problems to simplify such scenes efficiently. Mesh simplification algorithms try to approximate the surface with flat primitives. This works well for smooth surfaces that are locally flat. Even rough surfaces with small “bumps” can be modeled using normal maps [Cohen et al. 98]. However, highly irregular objects such as a complex tree with thousands of individual

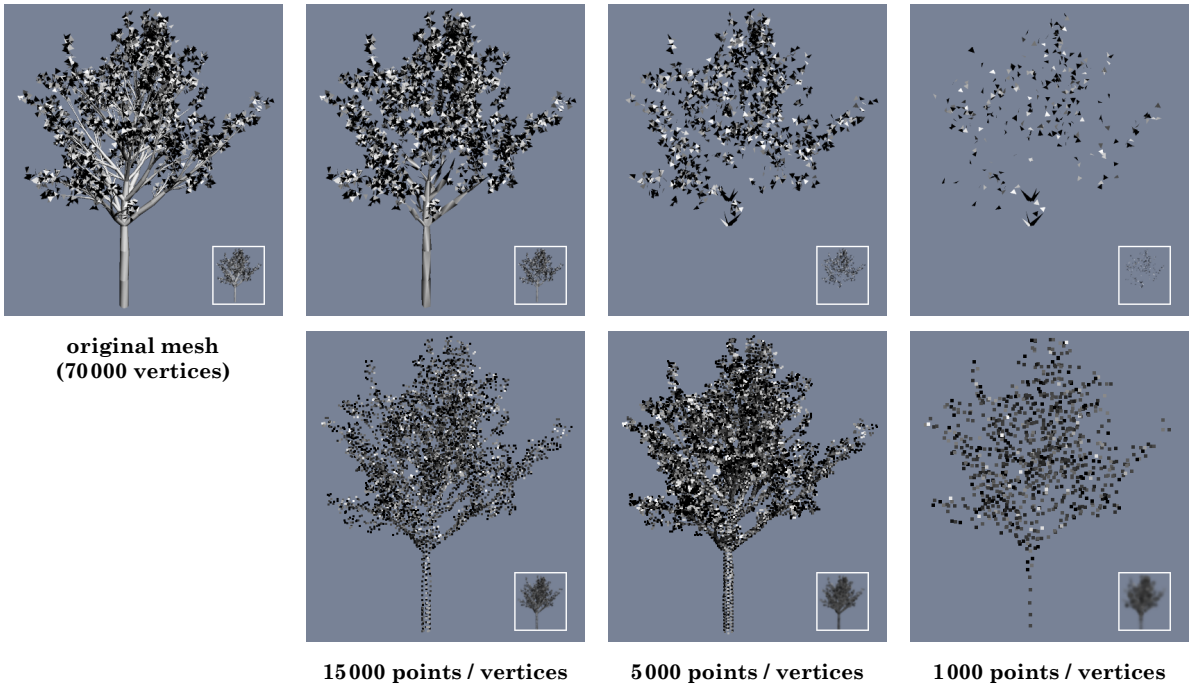


Figure 14: Comparison of triangle-based and point-based simplification.
Top row: triangle mesh simplification (QSlim 2.0, [Garland and Heckbert 97], default parameters)
Bottom row: point-based simplification (same number of points as vertices in the triangle mesh)

unconnected leaves and subbranches or a field of grass cannot be approximated well with flat triangles.

An alternative is the usage of image-based rendering techniques. Image-based imposters such as billboards [Rohlf and Helman 94] have been used extensively to render landscapes and outdoor scenes [Maciel and Shirley 95, Deussen et al. 98]. However, image-based replacements have to deal with parallax issues that lead to artifacts, especially in interactive walkthrough animations. To fight parallax problems, large numbers of images have to be used to represent objects faithfully. However, this leads to considerable processing and storage costs.

Another possible alternative is raytracing techniques. They can display scenes of arbitrary structure and topology and provide highly output-sensitive rendering times for many classes of scenes. Nevertheless, raytracing techniques still require considerable hardware (and implementation) effort to achieve real-time frame rates [Wald et al. 2001]. Thus, they are not always an alternative in interactive settings.

As none of the classic rendering approaches is fully satisfactory for an interactive display of natural scenes, we need a new rendering paradigm. In this thesis, we propose a new approach to the simplification of highly complex scenes: Instead of trying to fit triangular primitives to the surface, our approach is to reconstruct images from a set of sample points taken from the object surfaces. Conceptually, such a multi-resolution point-sample rendering algorithm works in two steps: First, a set of sample points is chosen from the surfaces of the scene objects. Second, an image is reconstructed out of the sample points. The distribution of the sample points is chosen in order to support an efficient image reconstruction later on. This usually means that the sampling density should be roughly proportional to the projected size of the objects in the image.

The usage of points as rendering primitives allows a more general approach to the simplification of complex objects. Point primitives are symmetric while triangles are oriented primitives. Thus, it is much easier to create strongly simplified representations of objects that are not locally flat without introducing a directional bias by specifying artificial surface orientations of the triangles. Additionally, traditional triangle meshes that are handled by automatic simplification algorithms always have to maintain the local connectivity of the primitives. However, a strongly simplified, highly irregular surface has to change its topology drastically in order to allow a lightweight representation. Thus, the topology of the simplified mesh will have little in common with that of the original mesh. Nevertheless, traditional mesh simplification algorithms enforce a smooth transition of the original topology into the simplified one. At every step during the simplification, correct local connectivity information has to be maintained. In addition to complicating the algorithm, this can lead to an additional bias as the algorithm tries to keep things connected in cases where it does not make sense. In connection with an error metric that tries to minimize geometric distances, this can lead to inappropriate results. Figure 14 shows an example of a simplification of a complex tree model[7] using the QSlim simplification package that is available online [Garland and Heckbert 97]. The original 70,000 faces model was simplified to 15K, 5K and 1K vertices. In comparison, a point-based simplification using the same number of points is given, too. Two observations can be made: First, some of the main features of the object such as the main branch get lost at a strong level of simplification. The point-based simplification is not able to produce fully correct results either. However, it is able to preserve the overall shape of the object much better (see the smaller images for a "distant view"). Second, the simplification of the foliage of the tree somehow converges towards a point cloud consisting of triangles, but at higher costs (3 vertices per primitive, maintaining topological information). This also indicates that a point-based representation is better suited for such cases. It is important to note that the quadric error metrics used by the QSlim simplification package might be not well suited for rough and irregular surfaces. Thus, another error metric would probably have avoided some of the artifacts. However, a point-based representation still seems to be more efficient to represent the large set of smaller branches and leaves at a strong level of simplification.

The point-based rendering approach also avoids problems of classic image-based rendering approaches. As every sample point stores a three-dimensional position in space, parallax errors are avoided. The idea is very similar to image warping techniques; the connection will be discussed more in detail in the next section. Rendering of scenes from sample points has also a connection to raytracing. In some sense, it is a kind of inverse raytracing: Instead of systematically choosing sample points for each pixel on the screen, we choose sample points directly in object space with a sampling density that assures sufficient accuracy for each pixel of the image. This can be done more efficiently, as no complex inverse problem has to be solved.

Point-based rendering is another building block for handling complex scenes in real-time. It is a geometric simplification technique. The remainder of this thesis is supposed to prove that point-based multi-resolution rendering is a valuable supplement to the current state-of-the-art for handling complex scenes.

3.2 Related Work in Point-Based Rendering

The idea of using sample points for rendering purposes is not new. Many techniques such as particle simulations and certain image-based and volume rendering techniques have also employed

[7] The tree model was generated using the freeware L-system parser "lparser" [Lapré 2002], example file "tree11".

point primitives for efficient rendering. However, in contrast to the techniques presented in this thesis, most of them did not make extensive use of multi-resolution techniques in order to support handling of large scenes. We will discuss previous point-based techniques in computer graphics in the next subsection (Section 3.2.1). A good survey paper of related techniques is also provided by [Zwicker et al. 2000].

In addition, several new point-based rendering and modeling techniques have been proposed in the last three years, being developed in parallel to the work presented in this thesis. In order to clarify the influence of the recent developments on the techniques described in this thesis, we will discuss the corresponding literature in an extra subsection (Section 3.2.2).

3.2.1 The History of Point-Based Computer Graphics

3.2.1.1 Early Work and Particle Systems

Point primitives first appeared in the rendering literature to model volumetric phenomena without distinct surface such as clouds or smoke. In 1979, Csuri et al. propose a general purpose rendering architecture for complex scenes [Csuri et al. 79]. They use large point sets (300,000 points in an example scene) to render clouds of smoke. Additionally, they note that points can be used as universal modeling primitive (also for opaque surfaces and volumetric models), especially for highly complex scenes. In [Blinn 82], a shading model for clouds of dust is derived using a stochastic particle model. However, no point primitives are used for display but the particle approach is only used for mathematical modeling.

The idea of using point clouds to simulate gaseous and fluid phenomena was extended by Reeves [Reeves 83]: A point-based representation is combined with a dynamic model to create realistic animations of such phenomena. The method was used to create the “Genesis” explosion sequence in the movie Star Trek II [Paramount 82]. The particle system technique is very general and can also be extended to render phenomena such as flowing water, spray, smoke and fire. The basic idea is very general: Each particle stores a set of attributes such as position, velocity, color, lifetime, opacity etc. The trajectory of the particles is calculated by solving ordinary differential equations based on these attributes. The particles are generated by stochastic rules and deleted after their lifetime expires. Additionally, a hierarchical generation is often useful: High level particles generate secondary particle systems stochastically (e.g. think of fireworks rockets and glow after explosion of the rockets).

In [Reeves and Blau 85], the idea of particle systems is extended to rendering of static, procedurally defined objects: Trajectories of particles are drawn into a single image to depict objects such as trees, leafs or flowers. A heuristic illumination model and shadow mapping is employed to create realistic images. The approach also allows for a simple level-of-detail control: By adjusting the initial particle density, the complexity of the representation can be adapted. The techniques are used to render the impressive background landscapes of the animated short movie “The adventures of André and Wally B.” [Lucasfilm 84]. The goal of this technique (efficient rendering of irregular objects such as landscapes) is similar to ours. The two main restrictions in comparison with the techniques described in this thesis are: First, the method works on procedural models, i.e. a particle system has to be devised manually for each type of object to be rendered. Second, the level-of-detail control is not a true multi-resolution technique: Reducing the number of particles alters the shape of the object and it is not possible to perform a level-of-detail control within one and the same object.

The idea of particles systems has been extended in several directions: Flocks and Herds of animals and humans can be simulated by implementing behavioral rules for each particle based on its local neighborhood [Reynolds 87]. The addition of topological elements like springs allows the approximation of continuum mechanical models for the simulation of deformable plates and solids. An alternative approach is the class of SPH methods (smoothed particles hydrodynamics) that have been developed originally in computational physics: Radial basis functions around the particles are used as a basis for a discretization of differential equations of a physical model of motion. For a survey of physically-based modeling and simulation techniques, including particle system approaches, see e.g. [Baraff et al. 2003]. Today, particle systems are a standard simulation technique used in real-time applications such as computer games as well as in photo realistic rendering for special effects [Trojansky 2001].

3.2.1.2 The REYES architecture

Another concept related to point-based rendering is rendering with micropolygons: A parametric surface is subdivided recursively in parameter space until the resulting fragments are smaller than a pixel [Catmull 74]. Afterwards, the resulting "micropolygons" are shaded and rasterized using z-buffering, a-buffering [Cook et al. 87] or a scan-line approach [Lane et al. 80]. This approach is the basis of the "REYES image rendering architecture" [Cook et al. 87]. The REYES-architecture is implemented in the well-known "Renderman" software package [Apodaca and Gritz 99], which has been used extensively in the production of computer generated movies and special effects. The main advantage of the approach is a high output quality at moderate costs: A hierarchical subdivision up to the subpixel level ensures smooth rendering of higher order surfaces. Additionally, the forward mapping approach is computational less intensive than raytracing techniques. In contrast to raytracing techniques, it especially allows a "stream" processing of the geometry and the resulting micropolygons. Thus, large scenes that do not need to fit into main memory can be handled by "streaming" from disc. The idea of an object space sampling up to the pixel level is the same as in point-based multi-resolution rendering. However, the approach has still restrictions: The subdivision hierarchy is only build for individual high level primitives. Therefore, the approach is not efficient for highly complex scenes in which even the high level description is too complex for a linear time rendering strategy.

3.2.1.3 Point-Based Surface Modeling and Rendering

In 1985, Levoy and Whitted proposed using point-based representations as universal rendering primitive [Levoy and Whitted 85]. They argue that a point-based representation is the greatest common divisor of all rendering primitives. They describe a technique to convert parametric smooth surfaces into a point-based representation. During rendering, several problems have to be solved: First, holes within renderings of continuous surfaces have to be avoided. Second, aliasing has to be avoided in the image reconstruction. And third, transparency and edge antialiasing has to be handled. The rendering technique reconstructs the image by associating a Gaussian splat in image space with each point. The size is adjusted according to the partial derivatives of the parameterization. Additionally, the radius of the Gaussian is restricted to one pixel in order to avoid aliasing. An a-buffer algorithm is used for visibility determination, allowing for transparency effects. A depth threshold around each sample point is used to distinguish between points on the same surface that should be merged by weighted averaging and points on different surfaces that should be combined by back-to-front alpha blending. The weighted averaging uses the values of the Gaussian and a normalization term based on a point density estimation using the partial derivatives. Small weight sums are interpreted as object borders and thus reduce the alpha values during blending.

This technique is similar to that which we employ for image reconstruction in Section 5.3.2.5. It is also the basis of other recent proposals for high-quality image reconstruction from point clouds [Zwicker et al. 2001a].

Point-based techniques have also been used to model three-dimensional objects: In [Szeliski and Tonnesen 92], surfaces are described by "oriented particles" that are called "*surfels*" by the authors. The particles are attached to one another by forces that are designed to yield locally flat surfaces. The surface can be edited by cutting, welding and free form deformations. A local modification of the force functions (removing the smoothness conditions) is employed to model creases. To account for stretch during editing, the point density is adapted automatically by inserting new points dynamically. An important advantage of such a point-based modeling technique is that it allows modifications of the topology of the objects without having to deal with base meshes or patch boundaries (as in the case of subdivision surfaces or NURBS).

Another application of point-based techniques for surface modeling is the extraction of implicit surfaces: [Figueiredo et al. 92] propose sampling implicit surfaces using particle systems: The algorithm is initialized with random particles that are attracted to the implicit surface by forces based on the gradient field of the implicit function. Repulsive forces are used to obtain a uniform point distribution. Finally, a triangulation is determined to create a polygon mesh. The technique has been refined by [Witkin and Heckbert 94]. They improve the base technique and propose a generalization to dynamically changing implicit surfaces.

Point-based rendering techniques have also been used to render complex terrains: The "*voxel space*" technique [Freeman 96] renders a regularly sampled grid of surface evaluations by picking sample points with a spacing according to the perspective projection. Solid bars from the ground are rendered in back to front order to resolve visibility. The technique has first been used in the computer game "Comanche" [Novalogic 92], which is a helicopter flight simulator for the PC platform. The game appeared already in 1992 and was the first PC flight simulator that allowed real-time rendering of fairly detailed landscapes. At that time, the target platform was a simple Intel 80386 based PC system with 4 MB of system memory. Thus, the complexity of the rendered landscapes was quite impressive. The technique has been refined and used for terrain rendering in several subsequent computer games (such as the action adventure "Outcast" [Infogrames 99], in addition to several sequels to the original "Comanche" game).

3.2.1.4 Image-Based Rendering

Another thread of publications leading to a point-based rendering paradigm can be found in the image-based rendering literature (see also Section 2.2.4). The main problem of purely image-based rendering is the large memory consumption. Thus, it soon became obvious that geometric information should be incorporated (if available) to improve the expressiveness of the description. This was first implemented in image warping techniques that deform depth images to new views. However, these had to deal with hole-filling problems. Thus, a straightforward generalization is to store multiple depth samples for each image point. This effectively creates a point cloud representation of the object that can be rendered from arbitrary view directions without the risk of displaying holes. The approach is described by Shade et al. as "*Layered Depth Image*" (*LDI*) [Shade et al. 98]. Their data structure uses a two-dimensional image array that stores a list of sorted depth samples for each pixel, allowing for fast incremental rendering. The paper describes techniques for the creation of the data structure from images and from synthetic objects. The latter are discretized using a modified raytracer. A similar technique was also proposed by Max and Ohsaki [Max and Ohsaki 95b] to render trees efficiently. In contrast to Shade et al., they use a multi-layer z-buffer approach instead of a raytracer to create the data structure: The objects are

rasterized and all fragments are recorded as sample points. A hierarchical version that uses LDIs with different resolution and a fallback to the original geometry for close-ups is described in [Max 96]. This approach is already quite similar to our proposal. However, a procedural hierarchy is used that has to be specified manually by a hierarchy of rendering scripts. It does not yet allow processing arbitrary scenes automatically.

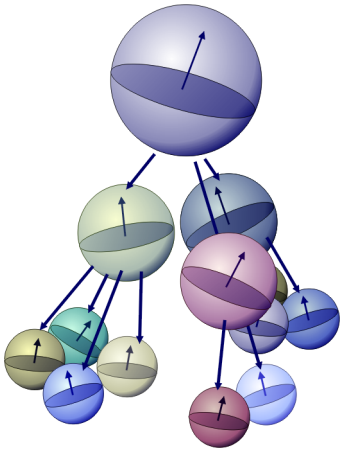
In [Lischinski and Rappoport 98], layered depth images are used for rendering global illumination effects. The proposal consists of two parts: First, a number of low resolution LDIs from different directions is computed in addition to a high resolution main LDI. These directional LDIs store precomputed secondary illumination and can be used to approximate glossy interreflection. Second, a raytracing technique is employed on the main LDI to render highly specular effects. The raytracer uses the base grid of the LDI as acceleration data structure and an interval search on the depth samples. The paper also proposes layered depth cubes (LDCs) that are a set of LDIs recorded from three orthogonal directions in order to ensure a regular sampling.

[Grossman and Dally 98] propose a "point sample rendering" technique: Point clouds are created from multiple orthogonal projections. A hierarchical z-buffer technique is used to resolve visibility: The point sample spacing is determined during the construction, guaranteeing a maximum point spacing on continuous surfaces. During rendering, a resolution in the z-buffer pyramid is chosen that matches the projected point spacing. For image reconstruction, a subsequent hierarchical push-pull algorithm is employed. In addition, backface culling is performed using normal cones for blocks of sample points. The approach aims at rendering of complex scenes. However, it does not employ a multi-resolution approach but only a constant sampling density. Thus, it is only efficient for a limited range of viewing distances.

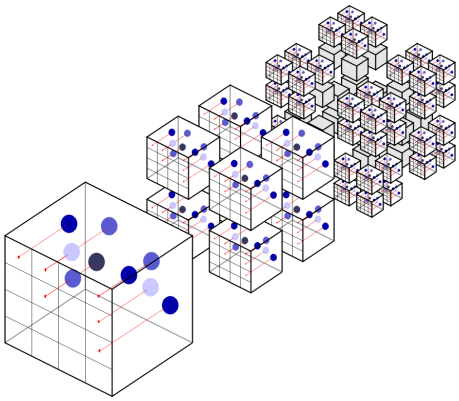
3.2.1.5 Volume Rendering

Volume data sets are representations of functions that assign optical properties to locations in space. Typically, they are represented as regularly sampled three-dimensional arrays that are the outcome of techniques such as finite differencing simulations or tomography scanners. Alternatively, volumetric data sets can also be represented as a mesh of volumetric primitives such as tetrahedra. A point cloud representation also represents a volumetric function that is somewhere in between these two approaches: It can be considered a sparse representation of regularly sampled volumetric data [Lischinski and Rappoport 98] as well as a set of one-dimensional primitives. Regularly sampled data is usually rendered by alpha-blending of textured slices [Akeley 93, Lacroute 94, Rezk-Salama 2000]. Mesh-based representations can be rendered efficiently by blending projections of the primitives [Shirley and Tuchman 91]. Alternatively, raytracing algorithms can be used [Kajiya 84] that allow for more flexibility (but usually at higher costs).

Westover suggests a rendering technique coined "*splatting*" that uses point primitives for rendering [Westover 90]: The volume is discretized into single sample points (in the paper, the technique is applied to regularly sampled data sets). A radial low pass filter function is associated with each sample point. The algorithm constructs the image by compositing several sheets in front to back order. For each sheet, the basis functions intersecting the sheet are rendered as "splats" and a continuous signal is reconstructed by additive blending. When a complete sheet has been reconstructed, it is combined with the image rendered so far by alpha blending. The usage of sheets allows a proper reconstruction of the volumetric function with basis functions with overlapping support. The splats can also be rendered directly into the image by alpha blending in depth sorted rendering order, omitting the sheet reconstruction. This technique runs faster but does not provide a properly antialiased reconstruction. A generalization of splatting to elliptical kernels is given in [Mao 96]: A generalized Poisson-disc sampling procedure is used to approxi-



(a) QSplat: Compressed bounding sphere hierarchy with averaged surface attributes in each node.



(b) Surfels: Hierarchy of LDCs. Prefiltering and mipmapping for antialiasing.

Figure 15: Schematic visualization of the QSplat and Surfels data structure.

mate a curve-linear volume with elliptical splats. Volume splatting has also been extended to hierarchical multi-resolution rendering in [Laur and Hanrahan 91]: An octree decomposition is used to adapt the splat density locally, allowing for the display of different levels of detail. A similar approach is used in [LaMar et al. 99]. LaMar's technique stores cubes of n^3 regularly sampled voxels in each node of the octree and renders them using textured slices instead of splats.

Volumetric representations can also be used to speed up the rendering of surface models. Chamberlain et al. propose the use of an octree hierarchy with each face of an octree node colored with the average color and opacity (as alpha-value) of an image of the contained geometry under orthogonal projection [Chamberlain et al. 95, Chamberlain et al. 96]. They prove a logarithmic running time for the case of equidistributed objects in the scene. The method works well for unstructured scenes such as landscapes, but the orthogonal projection approach leads to occlusion artifacts (false transparency) at continuous surfaces. [Neyret 96] uses volumetric representations to speed up raycasting of complex geometry. A parametric shading model based on normal distributions is fitted to the original geometry in each voxel to improve the display quality.

3.2.2 Recent Developments

3.2.2.1 Point-Based Multi-Resolution Rendering

Point-based multi-resolution rendering was introduced in 2000 by multiple authors: [Rusinkiewicz and Levoy 2000] describe a rendering system for large models from 3d-scanners dubbed "*QSplat*" (see Figure 15a). The algorithm uses a precomputed bounding sphere hierarchy that performs a hierarchical clustering of points that represent the original model. The inner nodes in the hierarchy store attributes (position, normal, color) that are averages of the children that have been collapsed into the cluster. A quantization scheme is used for a memory efficient encoding of the point hierarchy. The rendering algorithm traverses the hierarchy until the size of the bounding spheres (and thus the point spacing) is below a given splat size. Then, fixed size splats are drawn into a z-buffer to obtain a hole-free image.

[Pfister et al. 2000] describe a data structure coined "*Surfels*" (see Figure 15b): The scene is represented as a hierarchy of layered-depth-cubes (LDC) [Lischinski and Rappoport 98]. Each layered depth cube stores a point cloud approximation with fixed maximum sample spacing: The

original geometry is discretized by raytracing from three orthogonal directions and recording all intersection points. For textured surfaces, the surface attributes (especially surface color) are prefiltered by integrating over the texture footprint during preprocessing. The hierarchy is formed by an octree: Each node in the octree stores one LDC with a sample spacing that is a fixed fraction of the side length of the bounding box of the node. Thus, we obtain a hierarchy in which each child node represents the geometry with an increased precision (half sample spacing), leading to a multi-resolution representation. The rendering algorithm again traverses the tree until the projected sample spacing on the screen is below a certain splat size and draws points into the z-buffer. To delete occluded points that are visible through holes, a parallelogram approximating a tangent disc (with varying depth) is rasterized. It deletes all points with a larger depth. To fill the holes in the image, the authors propose simple splatting, hierarchical push-pull [Grossman and Dally 98] and Gaussian interpolation (see Section 1.3.2.2), leading to a trade-off between efficiency and image quality. For shading, the surface attributes from two adjacent levels in the hierarchy are interpolated linearly to avoid popping artifacts.

The randomized z-buffer algorithm, which is the basis of the techniques described in this thesis, was developed in parallel to the Surfels and QSplat approach [Wand 2000a]. It uses a hierarchy of summed area lists to perform a dynamic, randomized sampling of the scene geometry. Image reconstruction is performed by splatting, Voronoi diagrams or Gaussian interpolation, similar to the Surfels technique. The advantage of the randomized z-buffer approach is that it can render the original geometry with arbitrary precision, not limited by the initial, precomputed sampling. However, the dynamic sampling is more expensive in terms of rendering times than using precomputed sample sets. Thus, the two approaches can be combined, using a Surfels-like data structure to cache dynamic point sets [Wand et al. 2001]. The randomized z-buffer technique will be discussed in detail in Section 4.1 and Chapter 5.

3.2.2.2 Representation

Point-based representations are not always efficient: Large, flat areas of a scene can be processed much faster using a rasterization approach. Thus, the randomized z-buffer algorithm rasterizes the original triangles whenever the rendering costs of the point-based representation exceed the rasterization costs [Wand et al. 2001]. A similar technique was proposed concurrently by [Chen and Nguyen 2001] in the context of the QSplat system. In [Cohen et al. 2001], the idea is carried on one step further: The approximation hierarchy uses both triangle mesh simplification and point-based simplification techniques, depending on the local structure of the model. [Dey and Hudson 2002] propose a similar approach using a feature-based metric. Another option is the use of line primitives for simplification: [Deussen et al. 2002] use progressive sets of points, lines and triangles to display visualizations of complex plant ecosystems. The approach is based on [Stamminger and Drettakis 2001]: Here, a set of random points is rendered progressively (i.e. a prefix of the point list is rendered until no holes are visible anymore) to display complex objects. The progressive random point cloud technique lacks a hierarchical multi-resolution approach. Thus, [Stamminger and Drettakis 2001] use a second technique for extended parametric objects such as height fields. They perform a hierarchical subdivision in parameter space, similar to [Cook et al. 87, Freeman 96] but using an irregular “sqrt-5” subdivision pattern. Then, instances of the progressive point clouds are placed non-hierarchically in the scene, determining the sampling density on a per-object basis. The authors also apply their technique to speed up the display of animated scenes: The parametric sampling technique is efficient enough to perform resampling from scratch at every frame, thus allowing for animations of parametric objects. Non-parametric animated objects are rendered by moving the random point sets according to the dynamic deformation. However, the sampling density is not adapted to the deformations. Thus, only small de-

formations can be handled without creating holes on the surfaces. Another alternative to point clouds are clouds of billboards, i.e. small textured, partially transparent rectangles [Décoret et al. 2003]. Texture mapped primitives are usually rendered faster than unstructured point clouds as the rendering algorithm can utilize more coherence in terms of memory access and address calculations.

The expressive power of point-based representations can be improved by storing additional attributes at each sample point: [Kalaiah and Varshney 2001] use the second order surface curvature information along with point normals and positions to improve shading and to control the sampling density. Alternatively, statistical approximations based on a covariance analysis can be used [Kalaiah and Varshney 2003], similar to the technique proposed by [Schilling 2001] for the case of bump maps.

3.2.2.3 Improving Rendering Quality and Speed

The display of point clouds on raster graphics devices is subject to aliasing issues as this is the case for all discrete rendering techniques. [Zwicker et al. 2001a] propose a framework for anti-aliased rendering coined "*Surface Splatting*". The technique is an extension of [Levoy and Whitted 85]: Elliptical Gaussian splats around each sample point are projected to the screen and convolved with a unit Gaussian to restrict the splat size to one pixel. Occluded sample points are removed by visibility splatting [Pfister et al. 2000] with a small z-threshold to avoid deleting sample points from the same surface. The attributes in the image are reconstructed by a weighted sum of the attributes, which are re-normalized by dividing by the sum of weights (i.e. the projected Gaussians). Small weight values are interpreted as object borders. They are translated into alpha-values that are used during compositing. An a-buffer algorithm is used as rendering backend to support transparency and edge antialiasing. The surface splatting technique can also be adapted to volume rendering, as described in [Zwicker et al. 2001b].

To improve the rendering speed, surface splatting can be implemented using programmable graphics hardware [Ren et al. 2002, Botch et al. 2003]. Alternatively, splatting with a radial basis function controlling alpha blended compositing can be used, as proposed in [Rusinkiewicz and Levoy 2000, Coconu et al. 2002]. [Botsch et al. 2002] describe a hierarchical encoding scheme for software rendering that needs only 2 Bits to encode the position of a point in the hierarchy and allows efficient rendering from this representation. In [Dachsbacher et al. 2003] a multi-resolution rendering technique is proposed that runs completely on a contemporary programmable graphics processor: A QSplat-like hierarchy is stored in a linear list and the first and the last node needed for rendering are determined. Then, the corresponding interval in the linearized data structure is processed by the GPU. The drawback of this approach is the restricted adaptivity: Rendering a complete subinterval does not allow an efficient culling of subtrees in the hierarchy, effectively leading to a progressive rendering approach with a uniform sampling density (up to a constant factor) for the complete scene. As in [Stamminger and Drettakis 2001], the authors use instantiation of multiple objects to compensate for the restriction to progressive level-of-detail. An alternative to GPU-based implementations is to devise a specialized hardware for point-based rendering [Popescu et al. 2000, Amor et al. 2003]. However, the enormous computational resources offered by contemporary commodity graphics accelerators reduce the demands for a custom hardware solution.

3.2.2.4 Raytracing

Another direction for improving the image quality is a generalization of the rendering technique to global illumination techniques. [Agrawala et al. 2000] propose the usage of a LDI representa-

tion and raytracing for efficient soft shadow calculation, similar to [Lischinski and Rappoport 98]. [Schaufler and Jensen 2000] describe a technique for directly raytracing arbitrary point clouds based on a local surface reconstruction, allowing for full global illumination calculations. [Adamson and Alexa 2003] point out that the surface reconstruction is not necessarily unique, leading to artifacts when the surface is viewed under magnification. They propose an improved technique based on moving least square surfaces ([Alexa et al. 2001], see also next section) that circumvents this problem, but also needs larger computation times.

3.2.2.5 Points as Modeling Primitives

As already pointed out by [Szeliski and Tonnesen 92], the simplicity of point clouds (avoiding topology and connectivity issues) makes them a desirable representation not only for rendering but also for modeling. An additional motivation for point-based modeling techniques is the growing importance of three-dimensional surface scanners that create point clouds as native output format. Editing of such models directly based on point clouds is potentially more efficient and less prone to conversion inaccuracies than editing a primitive mesh created from the point cloud.

[Pauly and Gross 2001] propose a technique for surface filtering based on a patch-wise local parameterization and Fourier transformation. More general surface editing techniques based on global and local parameterization and resampling are presented in [Zwicker et al. 2002a]. To devise general surface editing algorithms, it is first necessary to define a unique surface described by a point cloud. [Alexa et al. 2001] propose a moving least squares (MLS) approach: The surface is given implicitly by defining a projection operator that projects nearby points in space to the uniquely defined surface: First, a local coordinate frame is created by a non-linear optimization process, according to the point to be projected. Second, the surface is reconstructed locally by fitting a polynomial height field using a weighted least squares approach with Gaussian radial basis functions around the point primitives as weights. The technique is extended to a progressive multi-resolution approach in [Fleishman et al. 2003]. [Pauly et al. 2003a] use the MLS technique to design algorithms for Boolean operations (such as union, difference, intersection) and local mesh deformations on surfaces defined by point clouds. A similar technique for Boolean operations is described concurrently by [Adams and Dutré 2003]. To optimize a point cloud, point cloud simplification techniques can be used similarly to mesh simplification techniques. [Pauly et al. 2002] suggest hierarchical clustering based on covariance matrices, point repulsion techniques [Turk 92] and point pair contraction with quadric error metrics [Garland and Heckbert 97]. An analysis of covariance matrices for point clouds can also be used for a feature extraction on point sampled surfaces [Pauly et al. 2003b].

3.2.2.6 Image-Based Rendering

Point cloud representations have also been used as data structures for the acquisition of three-dimensional models from photographs: [Matusik et al. 2002] capture geometry and reflectance behavior of objects using a visual hull algorithm and controlled lighting settings. The objects are represented as surface sample points with directionally varying reflectance properties. [Poulin et al. 2003] reconstruct three-dimensional scenes by matching random candidate points against multiple images. A three-dimensional real-time video transmission system based on sample points is described in [Würmlin et al. 2003]. Sample points with directional emission have also been used to build imposters in walkthrough applications [Wimmer et al. 2001].

3.2.2.7 Volume Rendering

The recent advances in point-based surface rendering have also led to an increasing interest in point-based multi-resolution volume rendering techniques. [Hopf and Ertl 2003] use a cluster hierarchy created by a hierarchical principal component analysis (PCA) to display large point clouds from particle-based numerical simulations. [Welsh and Mueller 2003] resample regularly sampled grids using elliptical basis functions for a set of sample points. They propose a Gabor wavelet decomposition to guide the resampling process. [Csébfalvi and Szirmay-Kalos 2003] compute x-ray projection images of large volume data sets by quasi-random importance sampling. The recent work on point-based surface and volume rendering might lead to the conjecture that both disciplines might eventually converge, employing point cloud representations associated with varying basis functions as uniform representation.

3.3 Components of the Point-Based Rendering Framework

3.3.1 Overview

In this thesis, we propose a set of novel approaches for rendering complex scenes. These approaches all utilize different point-based multi-resolution data structures to achieve output-sensitive running times. The following exposition is divided into three main parts: First, we describe data structures that provide different point-based multi-resolution representations of complex scenes. Second, we discuss different rendering strategies that make use of these data structures. Finally, an empirical evaluation is performed.

Two main data structures are proposed: The *dynamic sampling* data structures originally used in the randomized z-buffer approach [Wand 2000a, Wand et al. 2000b, Wand et al. 2001] allows a dynamic creation of random sample points in the scene according to an importance function based on the position and orientation of primitives in the scene. The approach has been developed independently, in parallel to the QSplat/Surfels approaches [Rusinkiewicz and Levoy 2000, Pfister et al. 2000] that are based on precomputed sample sets. It offers more flexibility, strongly reduced precomputation times and a better memory efficiency than the precomputed data structures. However, this comes at the expense of some computational efficiency. A combination of the two different approaches leads to a more efficient *static sampling* data structure. The data structure can also be extended to handle animated scenes (keyframe animations) [Wand and Straßer 2002]. The different data structures will be described in Chapter 4.

Based on these data structures, we will employ different rendering strategies: First, we will describe forward mapping algorithms that allow an interactive, real-time visualization of complex scenes (Chapter 5). Second, we describe a backward mapping (raytracing) algorithm [Wand and Straßer 2003a] that uses the multi-resolution representation to speed up the ray sampling process (Chapter 6).

Chapter 7 describes the implementation of our rendering strategies and provides an empirical evaluation.

In order to show the general applicability of the proposed techniques, we will also discuss briefly generalizations to other application areas in Chapter 8. We will consider the problem of rendering highly complex volumetric data sets, which can be solved using a similar multi-resolution approach as in the surface rendering case [Guthe et al. 2002]. We will also describe an

extension of the static sampling data structure to support out-of-core storage [Klein et al. 2002, Klein et al. 2004]. Additionally, we also describe how sample hierarchies can be employed for sound rendering in scenes with many sound emitters [Wand and Straßer 2003c] and how surface sampling can be used for real-time rendering of caustic effects in a global illumination system [Wand and Straßer 2003b]. We will conclude our discussion of point-based multi-resolution techniques in Chapter 9 with some ideas for future work.

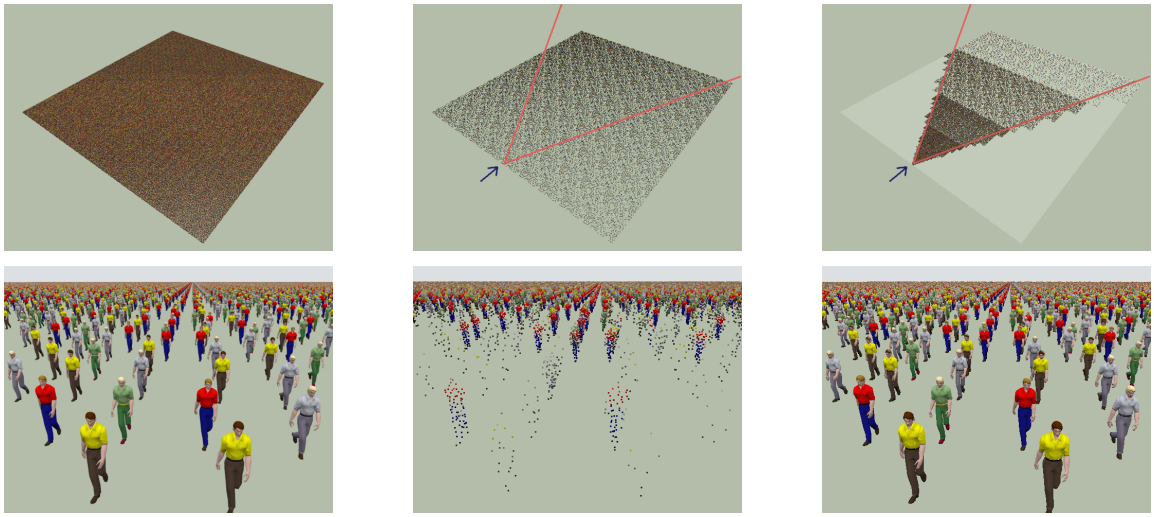
Chapter 4
Data Structures

In this chapter, we describe different data structures for surface sample set extraction from complex scenes. We will present two alternative approaches: The first technique generates sample points dynamically; the second one performs sampling already during preprocessing. In order to understand the consequences of different sampling strategies, we will perform a theoretical analysis of different strategies, proving upper bounds for the sample size and deriving confidence probabilities for the randomized parts of the algorithms.

Our goal is to reconstruct images of complex scenes from a small set of surface sample points. The key for doing this efficiently is importance sampling (see also Section 1.3.2.1): We should use a sampling density for regions of the scene according to their relevance for the image to be rendered. Additionally, a stratification of the sample sets can increase rendering efficiency. Nevertheless, the asymptotic efficiency mainly depends on the sampling probability distribution. Figure 16 shows an example: A complex crowd scene consisting of 575 million triangles is approximated by a sample set. A uniform sampling density fails to reproduce objects in the foreground faithfully and oversamples objects in the background. A sample set with a sampling density proportional to the perspective scaling factor allows an image reconstruction with uniform detail resolution throughout the image. If we tried to achieve the same quality with uniform sampling, a sample set with a size in the same range as the original scene description would be necessary, leading to non-output-sensitive running times.

In this chapter, we will present different data structures for the extraction of sample point sets according to viewpoint-dependent sampling density functions. The actual density function depends on the application: For simple perspective mappings (Chapter 5), a sampling density according to the perspective scaling in the image seems appropriate[8]. In the case of recursive ray-tracing with reflections and refraction, more general density functions have to be applied. In our case, we will use the ray density to control the sampling density (see Chapter 6). In order to unify the discussion of sampling data structures, we will assume that we are given a sampling density function for all parts of the scene. To create sample sets, we assume that it is possible to determine efficiently the maximum value of the density function for subsets of the scene geometry.

[8] Additionally, object features such as surface curvature or variation of texture can be considered to control the sampling density [Kalaiah and Varshney 2001, Dachsbacher et al. 2003]. This extension can be included easily in our data structures, as described later in the discussion.



(a) original image (crowd of walking people)

(b) uniform sample set / reconstructed image

(c) perspective importance sampling / reconstructed image

Figure 16: Sampling strategies. The first sample set (middle column) uses uniform sampling; the second sample set (right column) was created using importance sampling with perspective scaling as importance function. Sample sizes: upper row (overview images): 30K sample points, lower row (close-ups): 600K sample points, original scene: 575M triangles.

Using this information, we will extract sample sets with a sampling density that approximates the given density function in a conservative sense, i.e. the sampling density is guaranteed not to fall below the required density.

4.1 Dynamic Sampling

4.1.1 Overview

The goal of our first data structure is to extract *random* sample points from a collection of triangles with a probability density proportional to the given sampling density function. Three components will be used to provide this sampling mechanism: First, we use distribution functions according to the triangle areas. Second, we will arrange them in a spatial hierarchy in order to control the sampling density for different parts of the scene. Third, an additional global classification data structure will be employed to account for the orientation of the triangles and / or to exclude triangles from sampling that require too many sampling points.

4.1.2 Area-Based Sampling

First, we assume that the specified sampling density function is constant. Then, obviously, larger triangles should receive sample points with a higher probability than smaller triangles. Thus, the first step is to choose sample points with a probability according to the area of the triangles. To do so, we precompute distribution lists that allow an output-sensitive selection of sample points according to the triangle area:

A distribution list is a one-dimensional array. Each array element stores a reference to a triangle and a summed area value (Figure 17). The construction is straightforward: We initialize

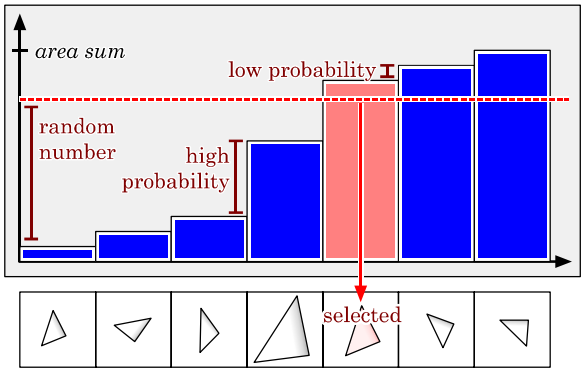


Figure 17: Distribution lists. A distribution function is computed by summing up the triangle areas. Then, triangles are chosen according to this distribution function by searching for the first triangle with a cumulative area value that exceeds a uniformly distributed random number. Each sample triangle is determined in O(log *n*) time for *n* triangles.

the current summed area value with zero and then iterate over all triangles in arbitrary order. In each iteration step, we add the area of the current triangle to the current summed area value and store this value in the array element along with a reference to the triangle. Obviously, this takes $O(n)$ time for n triangles. Using a precomputed distribution list, we can choose random triangles with probability proportional to the triangle area by binary searching [Press et al. 95]: First, we choose a random number from the interval $[0, 1]$ with uniform probability. Then, we multiply the value by the total summed area of all triangles. Then we perform a binary search for the first triangle with a summed area value above the random value. This procedure is equivalent to applying the inverse of the probability distribution function to a uniform random variable. This leads to random choices with probability proportional to the given probability distribution (see Figure 17). After we have obtained a triangle, we choose a random point on this triangle as a random linear combination of two of its sides. This yields a random point on a parallelogram. In order to obtain points on the triangle only, we mirror points located above the diagonal of the parallelogram. The whole sampling procedure takes $O(\log n)$ time per sample point.

At this point, we can incorporate a feature-based importance function. In order to increase the sampling density in areas of high surface curvature [Kalaiah and Varshney 2001], we can multiply the area values of triangles by a measure of surface curvature [Meyer et al. 2002] before constructing the distribution lists. Similarly, areas of high color variation can be assigned a higher sampling probability. It is also possible to support more general rendering primitives than triangles. The only requirement is that we are able to fetch a random point with (at least approximately) uniform probability from the surface of the primitive.

4.1.3 Spatial Adaptivity

Distribution lists can incorporate arbitrary importance functions. However, they are static. View-dependent sampling densities cannot be handled efficiently. If we included a view-dependent sampling density into the distribution lists, we would have to recompute the complete list every time the observer moves. This means that we lose the output-sensitivity of the approach, as the recomputation takes linear time. Thus, we need additional data structures to account for dynamically changing sampling densities.

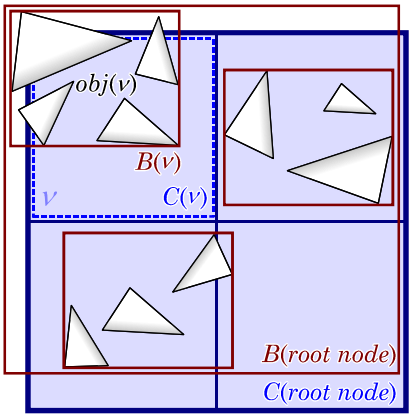


Figure 18: Definition of bounding boxes for octrees. $C(\nu)$ denotes the cube associated with a box due to splitting the root box, $B(\nu)$ is the bounding box of the associated triangles obj(ν).

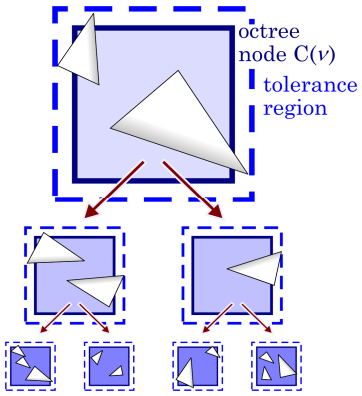


Figure 19: Limiting the octree bounding box overlap using tolerance regions. Triangles exceeding the tolerance region are stored in the corresponding parent node.

4.1.3.1 The Spatial Hierarchy

Usually, the desired sampling density for a certain viewpoint depends strongly on the spatial location. This dependence usually shows some coherency, i.e. we can expect the sampling density to be similar in neighboring areas of the scene. Thus, we can establish groups of triangles in close proximity and assign a similar sampling density (per unit area) to all triangles in such a group. The size (spatial extends) of the group should also be adapted to the desired importance function. For example, we can expect a stronger variation of the sampling density in close proximity to a viewer than for triangles that are farer away. In order to allow for groups of varying size, we employ a spatial hierarchy of groups. We use an octree [Samet 89] to define the spatial hierarchy of groups of triangles. First, we define some notation (Figure 18):

A *d-dimensional quadtree* (*octree* in case of $d=3$) for a scene $S \subseteq \mathbb{R}^d$ is a 2^d-ary tree that divides $\mathbb{R}^d$ into axis aligned bounding boxes as follows: We associate a d-dimensional bounding cube $C(\nu)$ with every node ν of the tree. The root node is given by a cube of minimal side length that contains all objects of the scene S. The cubes of child nodes are obtained by dividing the parent node into 2^d subcubes by splitting the parent cube in its middle in all dimensions.

Additionally, a set of geometric objects $obj(\nu)$ is associated with every node ν of the hierarchy. The smallest axis-aligned bounding box enclosing all objects associated with the current node and with all direct and indirect child nodes in the subtree below is denoted by $B(\nu)$.

To build an octree for the triangles $t_1,\ldots,t_n$ in our scene S, we first calculate a bounding cube for the scene. This will serve as cube $C(\nu_r)$ for the root node ν_r. Then we apply the following recursive construction procedure: If the current set of triangles contains less than a constant number of n_{max} triangles, we associate the set with the current node and end the recursion. Otherwise, we split the cube $C(\nu)$ of the current node ν into 8 subcubes. We go through the current triangle set and put each triangle in the construction list of that subcube that contains the center (arithmetic average of the vertices) of the triangle. Then the construction procedure is called recursively for all non-empty construction lists of the child nodes.

The data structure can now be used to speed up the identification of triangles from a certain region $R \subseteq \mathbb{R}^3$ (*range queries*, see also [de Berg et al. 97]): We recursively traverse the tree. If the bounding box B(ν) of the current node ν intersects with R, we report those triangles from

obj(v) that intersect with R and continue the traversal for all child nodes. Otherwise, we stop the recursion.

4.1.3.2 Improvements

Our triangle hierarchy still has some flaws: The first problem is overlapping of the nodes: As triangles are sorted into subcubes according to their center, they are allowed to exceed the octree cubes C. Thus, the bounding boxes B of the nodes in the tree can overlap. This can lead to performance penalties in geometric range queries because multiple octree nodes have to be visited to find objects lying in the same geometric range. In order to limit this effect, we modify the triangle selection procedure: Our goal is to limit the overlapping to a constant factor, i.e. the volume covered by bounding boxes B at each level in the hierarchy in the tree should only exceed the volume of the octree boxes C by at most a constant factor. This can be ensured by enforcing that the maximum side length of B-boxes must not be larger than a constant factor $(1+\delta)$ times the side length of the corresponding octree cube C. Typically, we allow a factor of 1.25, leading to at most an increase of volume of $1.25^3 \approx 2$. If the extends of a triangle exceed the extends of a child cube by more than $\delta/2$, it is not passed to the child node but stored in the parent node (see Figure 19).

This modification of the data structure is important in practice. Otherwise, the triangle hierarchy can easily degenerate and offer only a bad spatial classification. Besides this problem, there are two additional issues concerning memory requirements and construction time. However, these are more of theoretical interest:

The second problem is the limitation of memory requirements: Theoretically, the data structure might consume an arbitrary amount of memory. In order to guarantee that the construction procedure terminates, we have to assume that all centers of the triangles are disjoint. Otherwise, the recursive subdivision could continue endlessly. This is no severe restriction as it is obviously always fulfilled for non-degenerated (non-zero-area) triangles that are disjoint except of their edges. If we assume that every leave node contains only 1 triangle ($n_{max} = 1$), the depth of the octree is given by the logarithm in base 2 of the ratio of the side length of the root bounding cube to the smallest distance of two triangle centers in the scene [de Berg et al. 97]. However, this still means potentially unlimited memory requirements of the data structure. In order to enforce linear memory consumption, we have to introduce *shortcuts* [Bern et al. 90, Fischer et al. 98]: The large memory consumption is due to *useless splits*. A split into child nodes is useless if it does not divide triangles from one another but all triangles are stored in the subtree of one child node only. In this case, we do not store the split explicitly but shrink the subcube C. We calculate the bounding box of a subcube that enforces a split by calculating the bounding box of the child geometry and round the coordinates: Rounding means that we calculate the next-larger power of two corresponding to the side length of the box and then move the coordinates of the box outwards to the closest position in a grid of this power of two in the parent box. Then this box is subdivided and the corresponding child nodes are created. This data structure provides the same information as the full tree (the missing useless splits can be inferred) but needs only linear memory. This is easy to see: Every split divides the current list of triangles into (at most 8) non-empty pieces. In the worst case, the split divides at least two triangles from one another. This means that the data structure cannot contain more splits (i.e. octree nodes) than triangles. Thus, it will use linear memory.

The third issue is construction time: The algorithm presented above has a worst case $\Theta(n^2)$ running time. In the worst case, every node splits only one triangle from the others. Nevertheless, it needs linear time to inspect all triangles and sort them into the lists for the child nodes, leading to quadratic construction time. To avoid this behavior, we can employ a technique proposed by

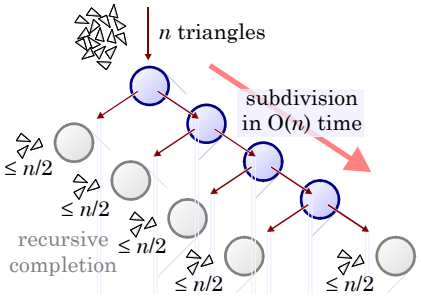


Figure 20: Asymptotically optimal octree construction in O($n$log n) time.

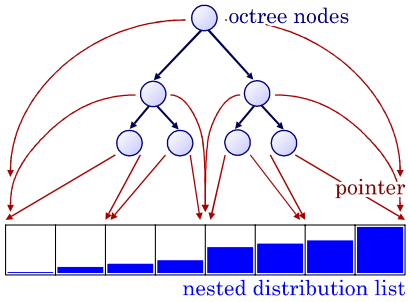


Figure 21: Nested storage of the distribution lists for the octree nodes.

[Callahan 95]: The problem is that one child node might contain most of the triangles while the others contain only a constant number of triangles. To avoid this situation, we continue to divide child lists into child nodes until each list contains less than $n/2$ triangles. Then we apply the algorithm recursively to all child lists that still contain more than n_{max} triangles (Figure 20). If we can perform this divide step in linear time, we will obtain O($n$log n) construction time: The length of the remaining lists will be less than half the size of the input so that O(log n) steps with O(n) running time each are necessary. How can we perform a linear time division step? To divide a point set (i.e. the centers of the triangles) into small sublists according to octree boxes, we first interpret an octree node as a binary tree of splits in the three coordinate directions (1 in x-, 2 in y-, and 4 in z-direction). In order to perform one such binary split efficiently, we start with three sorted doubly-linked lists, containing the triangles sorted by the x-, y-, and z-coordinates of their centers. For the root node, the lists are sorted explicitly in O($n$log n) time. For the recursive calls to the divide algorithm, they will be constructed on the fly. To perform one split in one of the three possible directions we first copy all three lists. Then, we go through the corresponding list in alternating front and back order. Thus, we can find the first entry that exceeds the splitting plane in the middle in time proportional to the size of the smaller of the two sets. The triangles corresponding to the smaller of the two sets are deleted from the lists. In the copy of the lists, the triangles are not deleted but a pointer to the corresponding child node is stored at the list entry. The remaining lists are still sorted and contain all triangles for the larger of the two halfs. Thus, we can apply the technique recursively until the remaining child list contains less than half the triangles of the original list. Afterwards, we must sort the constructed child lists. We do this by going through the three copies of the original triangle lists and follow the pointers to the child nodes in order to append the triangle to the corresponding child list. Then, we have the same situation at each unfinished child node as at the beginning and we can call the construction procedure recursively for each unfinished child.

Care must be taken to integrate the short-cut technique into this scheme. To do a short-cut, we must augment the splitting planes: Instead of using the middle of the current node, we have to calculate a potentially shrunk child node and use the middle of this node for splitting. The corresponding calculations can be done easily every time we start a new node: As we have sorted lists for the triangles in this node, we can determine the bounding box of the triangles in all three-dimensions and do the short-cut calculations as described above.

Using this technique, we can construct an octree with short-cuts in optimal O($n$log n) time using optimal O(n) space. The construction time is optimal because the data structure can be used to output a sorted list of triangle centers in one of the three coordinate directions. Sorting of a set of arbitrary real numbers is an $\Omega(n \log n)$ problem. Hence, the construction time cannot be improved asymptotically. The optimality of the space requirements is obvious.

```
Algorithm ε-sampling(Node ν):
 If maxSamplingDensity(B(ν)) > 0 Then
   If maxSamplingDensity in B(ν) is sufficiently uniform
   or ν is a leave node Then
     numberOfSamplePoints := maxSamplingDensity(B(ν))·area(ν)
     For i:=1 To numberOfSamplePoints Do
       chooseRandomPoint(ν)
     End For
   Else
     For each children c of ν Do
       ε-sampling(c)
     End For
   End If
 End If
```

Algorithm 1: Conservative sampling of an importance function using a spatial hierarchy.

The last two improvements cause a considerable implementation overhead. In practice, they are usually not necessary: Typically, the triangle sets are distributed fairly uniformly. Note that we need an exponentially decreasing spacing of triangle centers to run into problems with the size or the construction time of the data structure. Restricted irregularities do not matter as the size of the octree boxes shrinks exponentially even without short-cuts. Thus, the proposed improvements have not been implemented. They are just given to show theoretically optimal construction time and space of the proposed data structure.

4.1.3.3 Integrating Distribution Lists

In order to use the spatial hierarchy to control the sampling density, we have to incorporate distribution lists into our spatial hierarchy. A naïve approach would be to build a distribution list for each node in the octree, containing all triangles in the corresponding subtree. However, this would lead to best case $\Omega(n \log n)$ and worst case $\Theta(n^2)$ space requirements. To maintain linear memory complexity, we use nested distribution lists: A depth-first traversal of the octree yields a linear enumeration of the octree nodes and the stored triangles. The enumeration has the property that every subtree in the hierarchy appears as subinterval in the linearized list. Thus, we can use one distribution list for all octree nodes. We build this global distribution list by performing a depth-first traversal. At each node v, we first insert the triangles $obj(v)$, which are assigned to the node itself, into the distribution list. Then we perform the procedure recursively for all child nodes. Lastly, we store two indices in node v that mark the interval of the triangles of that node and the corresponding subtree. Using this information, we can chose sample points with uniform probability from all triangles contained in a node v using the same strategy as described in Section 4.1.2. The only difference is that we have to restrict the search to the interval defined by the two indices stored in v and that we have to scale the random numbers used for triangle selection to the according area interval.

4.1.3.4 The General Sampling Algorithm

Using this data structure, we can choose surface sample points according to general importance functions. We require two operations on the importance function: First, we must be able to de-

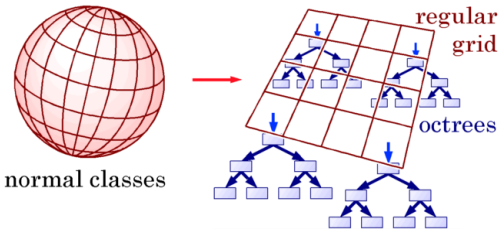


Figure 22: Classification by normal orientation using normal classes in polar coordinates.

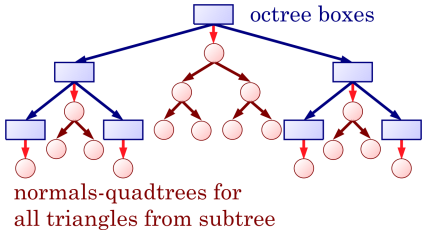


Figure 23: Theoretical alternative: two-level tree. For each spatial node, a quadtree in the orientational domain is stored, containing all triangles from the subtree of its node.

termine whether the sampling density is sufficiently uniform within a given axis aligned bounding box B. The choice of this criterion is a trade-off between approximation accuracy and running time. Second, we must be able to calculate the maximum sampling density that is found within such a bounding box in order to determine a conservative sample size. Using this information, we can employ the following algorithm to obtain a sample set with a sampling density that is a conservative approximation of the original one (Algorithm 1): We perform a depth-first traversal of the hierarchy and measure the ratio between the maximum and the minimum sampling density within the current node. If the sampling density does not vary significantly within the current node, a set of random sample points according to the area of the triangles is generated and the recursion is stopped (the area of the triangles in the box can be determined using the distribution list). Otherwise, the algorithm is applied recursively to the child nodes until the desired accuracy is met. The running time of the algorithm depends on the specified importance function and the approximation criterion. A detailed analysis for different perspective importance functions will be given in Section 5.2.3.

4.1.4 Orientational Adaptivity

For perspective projections, the sampling density depends mainly on the distance of a group of objects to the viewer and on the angle under which a viewing ray meets the surface of an object. Up to now, we have only performed a classification concerning the spatial location, i.e. we can only incorporate the distance into our importance function efficiently. To improve the sampling accuracy, we can augment our data structure in order to account for orientational properties of the sampled objects. The space of all surface orientations of a triangle can be represented as the unit sphere, containing all possible normal directions. Thus, we have a two-dimensional domain of possible surface orientations. A problem of importance functions depending on the orientation is that the surface orientation is independent of the spatial location of a triangle. For the spatial classification we have usually coherence among the three spatial coordinates: Surface points in close proximity have usually a similar importance (at least for simple perspective projections). If we include the orientation into our importance measure, the orientation of the triangles can be arbitrary, independent of the spatial location[9]. Thus, it is probably not efficient to incorporate the orientation as two additional axes into our spatial subdivision scheme, leading to a 5D-subdivision.

[9] unless we have to deal with simple smooth surfaces, which are not our primary target as they are handled well with mesh simplification techniques

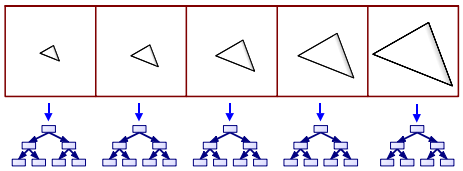


Figure 24: Classification by triangle area. The area value of the classes increases exponentially.

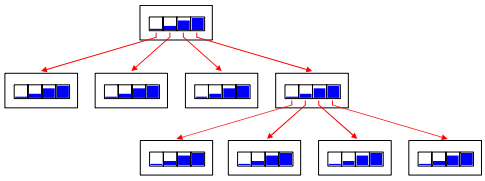


Figure 25: Using the octree as distribution tree. Local distribution lists are used to choose child nodes.

Instead, we just do a simple preclassification: We divide the domain of all orientations into classes of similar normals. In our implementation, we use a simple regular grid in polar coordinates. However, this might be improved by employing e.g. a regular subdivision of a Platonic solid to define the classes of similar normals. Afterwards, we build a spatial subdivision structure for each of the obtained *normal classes* (see Figure 22). The sampling algorithm is then applied to each of the classes, effectively multiplying the spatial classification effort by the number of normal classes. The calculation of the number of sample points in Algorithm 1 can use the additional orientational information to improve the estimation of the necessary sampling density.

An alternative to the preclassification approach could be a multi-level tree, similar to range trees [de Berg et al. 97]. In this data structure, a quadtree in the domain of all normal vectors is build for all triangles in the subtree of each node in the spatial hierarchy (Figure 23). This approach would allow a more adaptive classification by surface orientation than a simple regular preclassification. However, the data structure would consume O(hn) memory for n triangles and h being the height of the spatial tree[10]. This is probably much too much for large scenes. Additionally, this data structure does not allow for instantiation in order to encode complex scenes more efficiently. Thus, we have not examined this option more in detail.

4.1.5 Identifying "Large Triangles"

Sample points are only an efficient representation for rendering if the ratio between the number of triangles and sample points is large. Rendering of low-detail models with many sample points wastes computational resources. A criterion for efficient sampling is the number of sample points per triangle. If this number is larger than a small constant (say at most 1-3 points per triangle) sampling becomes inefficient. We call triangles that receive more sample points than the specified constant bound *large triangles*. To identify large triangles, we have to consider both the view-dependent sampling density and the actual size (more precisely the area) of the triangles. During the execution of the sampling algorithm (Algorithm 1), the actual sampling density employed for creating sample points is known exactly but the area of the triangles may be arbitrary.

In order to provide the missing information, we can employ preclassification by triangle area (*area classes*, see Figure 24): We consider the area values of all triangles in the scene and divide the corresponding area interval on the real line into classes of constant relative error, i.e. by an exponentially increasing spacing. If a_{min} is the minimum[11] and a_{max} the maximum area of a triangle in the scene and we employ a factor c to construct the classes, we need $\log_c (a_{max}/a_{min})$ different area classes. For each class, we construct a spatial data structure (possibly including normal classes, as described above). In the sampling algorithm, we multiply the sampling density

[10] In each of the h levels of the spatial tree, $\leq n$ triangles are stored in the orientational quadtrees.
[11] To employ the exponential spacing, we must assume $a_{min} > 0$.

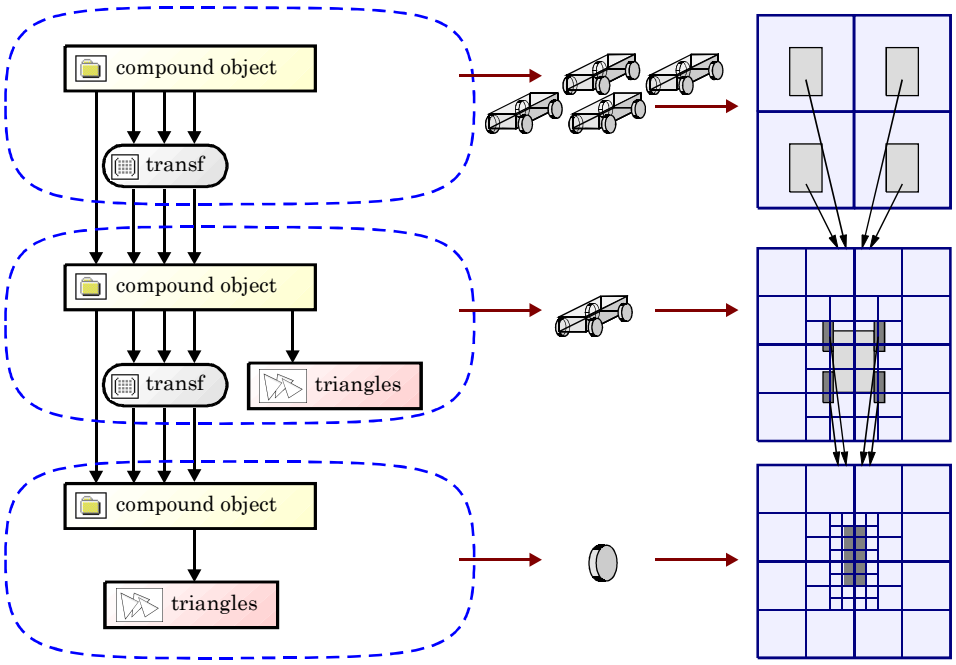


Figure 26: Scene graph based instantiation combined with spatial data structures for efficient sample point extraction. Spatial data structures can be assigned to groups of objects. A complete data structure is treated as compound object and inserted into a parent data structure. During traversal, the instantiation transforms have to be considered to retain the impression of one "large" octree.

by the minimum area of the triangles. If this value exceeds a certain threshold, we report all triangles of the subtree instead of generating sample points. The effort for the classification is multiplied by the number of area classes. Due to the exponential spacing, the number of area classes is usually fairly low.

The overhead of the introduction of area classes can be avoided in special cases: For certain sampling density functions and approximation criteria, the size of the bounding box and the sampling density are in a fixed relation if we fix the approximation accuracy a priori (see Section 5.2.3.2). Thus, we can directly store the large triangles in the octree node at which the sampling density becomes too large, creating too many sample points on the triangle.

4.1.6 Instantiation

As described in the Section 1.1.1, we would like to encode our scenes as scene graphs. This allows the specification of large amounts of geometry that would not fit into memory in an explicit representation. This scheme should also be adapted to the sampling data structure in order to allow processing large scenes.

The idea used for integrating instantiation is fairly simple: We extend our data structure to allow references to complete data structures to be handled in addition to simple triangles. Whenever a part of the scene is instantiated multiple times, a separate sampling data structure is build for that part of the scene. Then, pointers to this data structure are inserted into a global data structure along with a transformation matrix that describes the differences among the instances. This scheme can also be applied hierarchically, allowing for encoding arbitrary scene graphs (see Figure 26). The sampling algorithm can be easily modified to account for this instantiation technique: Whenever the sampling algorithm tries to step into an instantiated part of the tree during

recursive descent, the camera is just transformed by the inverse of the instantiation transformation[12] and the traversal is continued in the instantiated tree. The transformation matrix is recorded on a transformation stack. If sample points (or "large" triangles) are generated, these are transformed by all matrices on the transformation stack (last transformation first) in order to place them in the correct spatial position. The other case is that sample points are drawn from instances but the sampling algorithm did not descend into the corresponding tree. To deal with this case, we also insert a reference to the sub-data structure into the distribution lists, side-by-side with triangles. If the random sampling algorithm (Section 4.1.2) selects a data structure, the sampling is called recursively for the sub-data structure. Again, the sample points have to be transformed by the instantiation transform in order to put them at their correct place.

This scheme also works with preclassification by area and/or orientation: The instantiated data structures consist of different classes (by normal/area) that have to be inserted into the corresponding classes of the parent data structure. The only necessary restriction is that all classifications of all data structures must use the same classes (i.e. identical area intervals and identical normal sets). For a multi-level tree (Section 4.1.4), instantiation does not seem to be possible.

4.1.7 Dynamic Updates

So far, our data structure allows a rapid extraction of sample sets (and large triangles) for a given importance function, i.e. for given viewing settings. Thus, we can render walkthrough animations in which the user moves through a static scene efficiently. However, it is not possible to change the scene dynamically. Thus, we are neither able to edit the scene interactively nor is it possible to incorporate moving or deforming objects without rebuilding the data structures completely and invalidating the output-sensitivity.

In order to allow for dynamic changes, we need update operations that perform an efficient local modification of the scene. At least, this would permit handling small modifications of the scene efficiently. If large parts of a scene change dynamically, we need a priori knowledge of the future behavior to be able to construct a sampling data structure during preprocessing (see Section 4.3). Otherwise, an output-sensitive rendering strategy based on precomputed data structures does not seem to be feasible.

A dynamic version of our sampling data structure should support two operations: `insert()` adds a triangle (or a complete instance along with its individual transformation) into the data structure. `delete()` removes a triangle (or an instance) from the data structure. An `update()` operation that changes the shape of a triangle or the transformation of an instance can be easily constructed using a sequence of a `delete()` and an `insert()` operation.

Two tasks have to be performed for a dynamic modification: First, the spatial hierarchy has to be updated. Second, the distribution lists must be updated. For the delete operation, an additional location step might be necessary if no pointer to the object to be removed is known but only its geometry. The update of the spatial hierarchy is a standard task. For the insert operation, we first have to locate the node in which the object has to be inserted by descending the tree in O(h) time (h = maximum depth of the tree). Then we check whether the new node is a leaf node. And if so, we check if it will contain more than the maximum of n_{max} objects allowed. If this is the case, all triangles are removed from the leave node and a new subtree is constructed by recursive split-

[12] This is only possible for transformation matrices that can be mapped to a movement of the camera position. Thus, we allow only orthogonal transformations (rotations, mirroring), uniform scaling, translations, and combinations of these transformations as instantiation transformations.

ting using the same recursion as during the static construction of the tree. Otherwise, the new object is inserted into the object list of the node.

Deleting objects from the tree is done similarly: After (if necessary) locating the node of the object in O(h) time, we remove the object from the node list. Then, we check if the subtree of the parent node will contain no more than n_{max} triangles. This can be done efficiently by maintaining a counter of the number of contained objects in each tree node. The update of this counter can be done in O(h) time during the traversal of the tree. If the number of triangles in the subtree of the parent node is small enough, the subtree is collapsed into the parent node to avoid nodes with small sets of triangles.

After the update of the spatial hierarchy has been devised, we must now consider the update of the distribution list. The problem is that the distribution lists storing the accumulated area values in the static case cannot be updated efficiently. Therefore, we substitute a dynamically balanced search tree for the distribution list, e.g. a binary AVL-tree [22]: The leaf nodes of the AVL-tree store the area values of the triangles and the inner nodes store the sum of their child nodes. The entries in the tree are ordered by a depth-first traversal of the octree. Therefore, each octree node can mark up its region in the tree by two pointers to leaf nodes in the search tree, similar to the subintervals of the original distribution list.

When a dynamic insertion or deletion takes place, a leaf node is added or removed from the AVL-tree. The summed area values of the parent nodes must be corrected, taking O(log n) time and the tree must be rebalanced. The balancing operations of an AVL-tree preserve the order of the tree, so the intervals of the octree nodes are not affected by this step. The summed area values must be corrected for all nodes that are rotated during balancing, but this can be done locally in O(1) time per node. O(log n) balancing operations are necessary, so the overall update time is O(log n + h), the sample selection time remains O(log n) as in the static case.

To choose sample points, a similar search algorithm as in the static case can be used with the AVL-tree: The search algorithm that selects a sample point now first runs upwards in the AVL-tree from the leaf nodes marked by the pointers in the spatial hierarchy, summing up all area to the left and to the right of the search interval. This information is used to determine the area in the subinterval. A random number from the range [0,1] is taken and it is scaled to the computed interval. Then, the search algorithm starts from the root searching the leaf node: At each inner node the first child is chosen, if its area value is not smaller than the search value. Otherwise, the second child is chosen and the search value is decremented by the area value of the first child. Overall, we maintain a guaranteed O(log n) sample point selection time.

The implementation of a separated balance search tree can be avoided if the scene consists of objects that are uniformly distributed in at least one dimension. Then the octree itself is relatively well balanced and it can be used as distribution tree: In each leaf node, the area values of the triangles are stored (Figure 25). In each inner node, a list with the eight summed area values for the children is stored. The sample selection algorithm now starts at the octree node that was selected by the box selection algorithm and generates a random number between zero and the summed area value of the last child node. Then it descends in the tree, choosing the last child node with a summed area value below the current search value. The summed area value of the child node directly before that node is subtracted from the current search value. So the correct leaf node can be found in O(h) time. Dynamic updates can be done in O(h) time, too: After adding or deleting a node from the octree in O(h) time an additional traversal from the leaf affected to the root node is necessary to correct the summed area values.

For practical applications, the second variant is probably the more interesting one, because although the first variant is asymptotically faster, it introduces a considerable overhead. Furthermore, the implementation of the second variant is much simpler and as the height of the octree is often not very large for practical scenes, the increase of the running time of the sample selection algorithm to O(h) may be acceptable. Our implementation uses only the second variant. In practice, we encountered only a moderate performance penalty for the sampling algorithm.

4.2 Static Sampling

4.2.1 Overview

In this section, we propose a modification of the dynamic sampling data structure to trade off flexibility for speed. Motivated by the Surfels/QSplat approach [Pfister et al. 2000, Rusinkiewicz and Levoy 2000], we will use precomputed sample sets. This allows us to improve upon drawbacks of the original dynamic sampling data structure: Dynamic sampling is less efficient as precomputed sample sets for several reasons: First, it is of course faster to lookup precomputed values than generating them dynamically. The generation procedure uses a binary search for random values. This leads to a second problem: The sampling algorithm does not show memory locality but potentially accesses the complete scene data base. This leads to performance penalties on modern microprocessor architectures that use a cache-based memory hierarchy that provides at least an order of magnitude speedup for localized memory accesses. The situation is even worse for out-of-core storage: If we have to deal with scenes that do not fit into main memory, we will observe a fatal performance penalty as random accesses are very expensive for secondary memory such as harddiscs. The third, important problem is the structure of the sample sets. The dynamic sampling data structure always produces random sample sets. In order to avoid noise artifacts (or even holes in the images), we are forced to oversample (see Section 4.2.3 and 5.3.1 for a detailed analysis). The convergence speed for random sampling is only O($\sqrt{n}^{-1}$) (see Section 1.3.2.1). Thus, we need a considerable number of sample points to construct high-quality images. Using precomputed sample sets, the sampling pattern can be optimized. This reduces the sampling overhead considerably. Furthermore, for certain illumination models, prefiltering as proposed by [Pfister et al. 2000] can be applied to avoid the noise problems completely.

However, using precomputed sample points means that we must limit the resolution of the point representation a priori. In order to avoid approximating the original geometry, we include the original geometry in cases where a sampled representation becomes inefficient. Thus, we are able to retain linear memory consumption.

In the following, we will describe two variants of the data structure: The first one yields sample points similar to the dynamic data structure described in the preceding section. However, it allows for more optimized, stratified sampling pattern and strongly reduced random memory access. A nested sample storage scheme ensures linear memory consumption. The second variant stores sample points without hierarchical nesting, which could lead to slightly increased memory demands. However, this allows more involved preprocessing strategies for the sample sets such as prefiltering of color attributes. A detailed comparison of all the data structures proposed in this thesis will be given in Section 4.4, concluding this chapter.

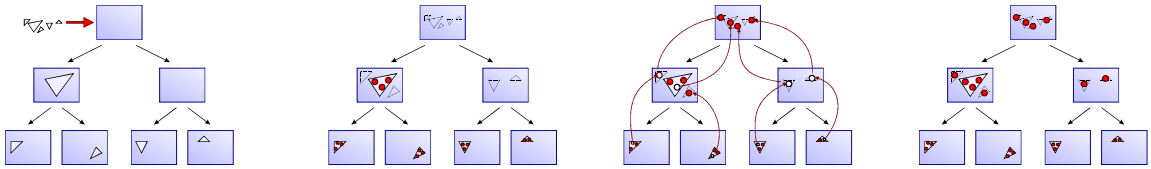
4.2.2 Precomputed Sample Sets

The main modification in comparison to the previous data structure is the usage of precomputed sample point sets instead of distribution lists. As proposed for the Surfels data structure [Pfister et al. 2000], we store precomputed sample sets in the nodes of the tree. Following [Pfister et al. 2000], we fix the ratio between the size of the octree boxes (side length of $C(v)$) and the sampling density. For each box, the spacing among sample points will be set to a constant fraction (typically 8 - 64) of the side length of the octree box. In each box, we store point clouds according to this sampling density. To extract sample points according to a given density function, we can employ Algorithm 1 with small modifications: Instead of stopping the traversal if the desired accuracy is met (i.e. the density function varies only by $1+\varepsilon$ the current bounding box), we stop the traversal if the demanded sampling density is reached (or exceeded). This strategy offers less flexibility than the dynamic data structure, which decouples the choice of regions of constant sampling density and their actual sampling density. However, we will show in Section 5.2.3 that this approach is equivalent to using a fixed approximation accuracy for perspective importance functions that demand a sampling density proportional to the squared inverse z-distance. In other words: We do not lose performance for perspective mappings, which is the most important case.

To construct the hierarchy, we build an octree similar to that described in Section 4.1.3: The octree splits triangle centers from one another until each node contains no more than a constant number of n_{max} triangles. Triangles that exceed the cubes of the octree nodes by more than a constant factor of the corresponding box side length are not propagated to child nodes but stored in the corresponding inner node. Additionally, we compute the expected number of sample points a triangle will receive at each hierarchy level. Triangles that exceed a small number p_{max} (usually 1-3) are stored in the current node and not propagated to child nodes. In the subtree below the current node, they are not considered during sample point generation. Only in nodes above they are represented by sample points. For the construction, we employ a simple divide-and-conquer algorithm that runs in $O(nh)$ time ($h \in O(n)$ being the height of the octree). The asymptotically optimal $O(n \log n)$ algorithm described in Section 4.1.3 cannot be employed because it does not permit sorting out large triangles efficiently. However, in usual computer graphics applications, the height h of the octree is a very small (typically $O(\log n)$) value so that precomputation efficiency is no issue in practice.

We start with a list of triangles and a bounding cube for the set of all triangles. If the list contains less than n_{max} triangles, we return a child node and terminate the recursive construction. Otherwise, we check whether the bounding cube can be shrunk in order to create a short-cut. Afterwards, the cube is split into 8 subcubes. Then we go through the list and look for large triangles. These are triangles with an area value that leads to an expected sample count of more than p_{max} points or triangles that exceed the tolerance zone of the child nodes (i.e. exceed the child cubes by more than a factor δ, see Figure 19). Finally, the remaining triangles are assigned to the subcubes according to their center (average of the three vertices) and the algorithm is called recursively to construct the subtrees.

After the octree has been constructed, sample points have to be generated (see Figure 27). To do this, we can employ two different strategies. The first is "*nested sampling*". It tries to optimize memory requirements and guarantees linear storage demands: First, sets of random sample points are created that cover the stored triangles with a sampling density as specified for the boxes in which the triangles are stored. Note that every triangle is stored in exactly one octree box with a depth in the tree corresponding to its area value. Next, a stratification algorithm is used that optimizes the positions of the sample points and removes superfluous points. Subsequently, sample points have to be created for the inner nodes. To do this, we do not compute addi-



(a) Creating the octree from a set of triangles.

(b) Creating sample point for triangles.

(c) Then: Propagation of sample points to higher octree levels.

(d) Alternative to (b), (c): full sampling at all octree levels.

Figure 27: Construction of a "static sampling" data structure: First, an octree is constructed for the triangles of the scene. Second, two different sampling strategies can be employed to create point sample representations. The first strategy creates sample points for the triangles in the octree and propagates them upwards in the hierarchy ("nested sampling"), creating a progressive point set. The second strategy creates new sample sets at every hierarchy level ("full sampling").

tional points but propagate points from child nodes to the parent nodes (depicted in Figure 27b,c). During hierarchy traversal, these points will be found by the traversal down from the root node and included in the output of the algorithm. This means that the points for parent nodes have to be subsets of the points used for the child nodes. We determine these subsets by just calling the stratification algorithm again for child point sample sets. The subset selected by the stratification technique is then moved to the parent node. This technique can also only be employed if the sample points are not modified during stratification. Some simple modifications like quantizing the point positions to a regular grid (see Section 4.2.3.3) can be applied on the fly, during hierarchy traversal. However, techniques like prefiltering of color attributes by averaging neighboring sample points cannot be used with nested sampling.

The nested sampling strategy obviously leads to linear memory requirements: For each triangle, only a fixed number of sample points are created and placed somewhere in the hierarchy. The hierarchy itself also needs only linear storage due to the shortcuts. To store points for different approximation accuracies even at shortcuts, we can use a simple progressive encoding: We run the stratification algorithm multiple times for the different inner nodes that have been left out and store the resulting sample sets in order of increasing density[13]. The resulting point sets are stored in order of increasing density so that the fraction necessary for approximating a left out inner node can be extracted efficiently.

The second variant of the data structure uses a "*full sampling*" strategy: It does not try to nest the sample sets but creates a completely new sample set at each node, independent of the child nodes (Figure 27d). This also means that we cannot use short-cuts, as we need all inner nodes to store point sets with a sampling density corresponding to the node size. Without nesting (i.e. reuse of sample points at different hierarchy levels), a progressive encoding at short-cuts in the tree is not possible. This leads to a (slightly) increased memory overhead: An octree without shortcuts can theoretically use arbitrary amounts of memory. However, this is only the case if the height of the tree is very large: Each "useless" split that would have been skipped by short-cuts in the tree shrinks the extends of the bounding box by a constant factor. An input scene can only lead to a large number of such useless splits if the scene consists of objects with very different scale (i.e. small objects with very small distance in a scene of large extends). The scale differences must grow exponentially to yield a linear increase of memory. Thus, this is rarely a problem in practice. The same arguments also apply to the number of sample points: The number of sample

[13] This increases the computation time from $O(hn)$ to $O(h'n)$, with h' being the height of the encoded tree rather than that of the short-cut tree.

points is proportional to a/d^2 with a being the area of the objects and d a constant fraction of the side length of the node bounding box $C(v)$ (in some cases also to $\Theta(a/d^2 \log a/d^2)$, see next section). Thus, the number of sample points again shrinks exponentially with decreasing depth in the tree, starting at a constant number of points per triangle. This does not guarantee linear memory demands in a strict sense but again it is very unlikely to observe problems in practice.

4.2.3 Sampling and Stratification

For creating the sample sets in the nodes of the spatial hierarchy, different sampling and stratification strategies can be applied. We will discuss different options in this subsection:

4.2.3.1 Random Sampling

The first sampling strategy is purely random sampling as already used for the dynamic sampling data structure (Section 4.1). The sampling technique is very simple: In order to create a sample set for a given sampling density, a set of surface points is chosen with uniform probability density on the surfaces of the triangles, independent of one another. In order to propagate a low resolution version to a parent node for nested sampling, a *sufficiently* large portion of the sample points is chosen randomly and moved from the child nodes to the parent. To perform random sampling successfully, we have to answer one question: How many sample points are necessary to satisfy a given sampling density? In the next subsection, we will analyze how many random samples must be taken from a given area fragment to guarantee a given sample point spacing. Using these results, we can decide how many points have to be included in the sample set and how many must be propagated to higher levels in order to ensure coverage of the surface.

Definition of *d*-sampling

The sampling density is informally given as maximum spacing among "adjacent" sample points on a continuous surface. To analyze different sampling approaches, we first have to make the notion of sampling density more precise. First, we denote a solid sphere of radius r around a point x by $B_{r,x}$, i.e. $B_{r,x} := \{y \mid \|y - x\| \leq r\}$. Next, we define the notion of an *interior* point within a piece of geometry: A point is called r_{max}-*interior*, if all spheres $B_{r,x}$ around point x with radius r contain at least a surface area of πr^2 for $0 \leq r \leq r_{max}$, i.e. the area measure $area(B_{r,x} \cap S)$ exists for all $0 \leq r \leq r_{max}$ and is no smaller than πr^2. Now, we define the notion of *d-sampling*: Let us assume we are given a surface $S \subseteq \mathbb{R}^3$ consisting of a finite set of triangles and a finite set $P \subseteq S$ of sample points from S. Then P shows a *sample spacing* d (*d-sampling* for short) if there is a sample point $p_i \in P$ within a distance $d/2$ from every $d/2$-*interior* point $x \in S$. This definition enforces a sampling with maximum sample spacing d on continuous surfaces ("interior points") within S. A coarser point spacing is only allowed at borders or if the surface contains holes so that the area condition is not fulfilled (see Figure 28). This is desirable because it is not necessary to cover regions in the scene that are fragmented or containing holes with the same sampling density as continuous surfaces. A reduced sampling density in such regions can lead to holes in the image reconstruction later on. This is acceptable as the original model itself also shows holes. It is also acceptable at borders, which cover a pixel (or a similar reconstruction area) only partially. However, at interior points, located within a continuous surface, this is not acceptable and avoided by our definition.

Formal Analysis: Sufficient Conditions for *d*-Sampling

Our question is now: How many random, uniform, independent sample points are necessary until we can be sure that a surface is sampled with sample spacing d? To answer this question, we perform in a rough sketch the following steps: In order to cover the surfaces with a maximum sample

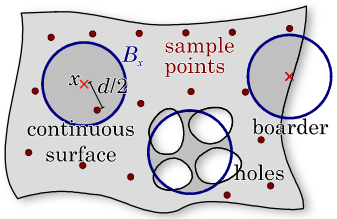


Figure 28: Definition of *d*-sampling. A sample point must be found within *d*/2 of each surface point if the area within *d*/2 around the point amounts to at least πr^2 for all $0 \le r \le d/2$.

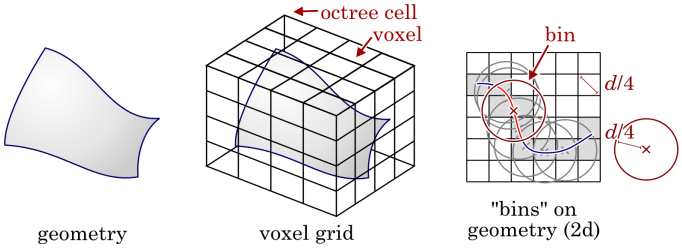


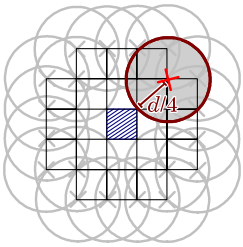
Figure 29: Dividing Geometry into bins. A grid divides the surface into fragments. Each fragment that contains a *d*/4-interior point corresponds to a bin, i.e. it should receive at least one sample point. The right image shows the overlapping bins schematically in 2d.

spacing of d, we divide the surface into symmetric cells of area of $\Theta(d^2)$. Then we make sure that every cell receives at least one sample point. This reduces our question to an occupancy problem (Section 1.3.3): For n equally sized cells, the expected number of random points that has to be chosen is approximately $n \cdot \ln n$.

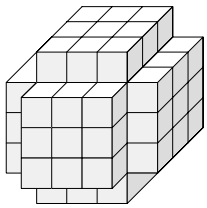
The main problem is the definition of symmetric cells on an arbitrary surface. To do so, we start with a regular grid with spacing $(\sqrt{3}/12)d$ that divides space into cubes with diagonal $d/4$. Then we make sure that every grid cell v that intersects with a piece of surface will receive at least one sample point in its $d/4$ neighborhood if the surface fragment comes from the interior of a closed piece of surface. Formally, if a $d/4$-interior point $x \in S \cap v$ exists, we guarantee a sample point in distance of at most $d/4$ to x (Figure 29). We will now first show that this enforces a sample spacing of d on S. Second, we will analyze the necessary sample size to satisfy these requirements.

To show that the conditions above yield a sample spacing of d on S, we consider an arbitrary $d/2$-interior point $x \in S$. To prove d-sampling, we must show that a sample point exists within a distance of $d/2$: The point x obviously lies within one of the grid cells constructed above. In this grid cell, there is at least one $d/4$ interior point x': As every r_{max}-interior point is also $r_{max}/2$-interior, at least x itself fulfills the requirements. By our prerequisites, we know that there is a sample point p in distance of at most $d/4$ to x'. Thus, in at most a distance of $d/4 + d/4 = d/2$ to x we find a sample point p, as required.

Next, we have to determine how many random sample points are sufficient to guarantee that every grid cell containing a $d/4$-interior point receives a sample point in distance of at most $d/4$ to such a point. To do so, we use an urn model, as discussed in Section 1.3.3. For each grid cell containing a $d/4$-interiour point, we form one "bin" that has to be hit by a randomly thrown "ball", i.e. it must receive a sample point. To estimate the sample size that fills all bins with at least one ball with a given probability, we have to count the number of bins and then determine the minimum probability of receiving a ball: We fix an arbitrary $d/4$-interior point x_i for each grid i cell that contains such a point. If no $d/4$-interior point exists, we also need no sample point for the grid cell, by definition. The "bin" of the grid cell is hit by a "ball" if a sample point from the area fragment $B_{d/4,x_i} \cap S$ is chosen. This area fragment has at least an area measure of $(\pi d^2/16)$. Consequently, a sample point is obtained with probability of at least $(\pi d^2/16)/A = \pi d^2/(16A)$ where A denotes the area of the surface to be sampled. To count the number of bins, we allocate an area fragment to each bin. Thus, we would obtain at most $16A/(\pi d^2)$ bins if all bins were disjoint. In fact, the bins can overlap so that we obtain a larger number. This increases the potential number of bins. Nevertheless, it is easy to see that the maximum overlap factor is bounded. Consider the area located in one voxel cell of the grid (see Figure 30). We now determine the maximum number



(a) 2d-case: at most 21 bins can overlap



(b) 3d-case: at most 81 bins can overlap

Figure 30: Limiting the overlap factor for the bins. Area contained in the innermost voxel cell (blue) can be overlapped by bins belonging to adjacent cells. In the 3d-case, an overlap factor of 81 is a conservative bound.

of bins from other cells that overlap with area in that cell. This yields an area overlap factor that is a conservative upper bound for the factor by which the number of bins can be increased. The surface area located in the considered cell can only overlap with bins from adjacent cells in which a point with a distance of at most $d/4$ to a point of the inner cell exists. In the three-dimensional case, these are at most 80 additional cells. Thus, we cannot have more than an overlap factor of 81, and thus, we cannot have more than $81 \cdot 16A/(\pi d^2) = 1296A/(\pi d^2)$ bins.

We want to make sure that we obtain a sample set with sample spacing d with a confidence probability of at least s (say s = 99%). According to equation (3) from Section 1.3.3, this means that it is sufficient to draw

$$k \geq \frac{\ln \frac{1296A}{\pi d^2} - \ln(1-s)}{-\ln\left(1 - \frac{\pi d^2}{16A}\right)} \tag{6}$$

random sample points, independent of one another, with uniform probability to guarantee a d-sampling with at least probability s. To obtain an asymptotic bound for small d, we replace $\ln x$ in the denominator by its first order Taylor expansion $\ln x \approx x - 1$, valid for values x approaching 1. Hence, we obtain a sample size of

$$\begin{aligned} k &\geq \frac{16A}{\pi d^2}\left(\ln \frac{1296A}{\pi d^2} - \ln(1-s)\right) \\ &\approx 5.1 \frac{A}{d^2}\left(6.03 + \ln \frac{A}{d^2} - \ln(1-s)\right) \\ &\in \mathrm{O}\left(\frac{A}{d^2}\left(\ln \frac{A}{d^2} + \ln f^{-1}\right)\right). \end{aligned} \tag{7}$$

with $f := 1 - s$ denoting the probability for the sampling procedure to fail, i.e. not to yield a d-sampling.

Sampling for Multiple Octree Nodes

For our "full sampling" data structure, we have to build sample sets for multiple nodes with different sampling densities. However, we want to make sure that we obtain a d-sampling in all nodes with a given confidence probability s. In order to guarantee this, we consider all nodes at once: Assume that we have given n nodes containing surface areas A_i that should be sampled with sampling distances d_i, $i = 1 \ldots n$. Then we obtain

$$m = \frac{1296}{\pi} \sum_{i=1}^{n} \frac{A_i}{d_i^2} \tag{8}$$

different bins. Each bin should receive sample points with (roughly) the same probability. Therefore, we perform a two step random selection process: First, we select the node v_i for which sample points have to be generated with a probability proportional to the area divided by the sample spacing squared (i.e. proportional to the required number of sample points), i.e.

$$\text{Prob}(v_i) = \frac{\frac{A_i}{d_i^2}}{\underbrace{\sum_{j=1}^{n} \frac{A_j}{d_j^2}}_{=:N}} = \frac{A_i}{N d_i^2}, \text{ with } N := \sum_{j=1}^{n} \frac{A_j}{d_j^2}.$$

In this formula, N is proportional to the number of "required" sample points, disregarding oversampling. After selecting the octree node, we choose surface sample points with uniform probability, as described in Section 4.1.2. Hence, each "bin" receives sample points with a probability of at least $\text{Prob}(v_i) \cdot (\text{minimum probability per bin}) = \pi d_i^2/(16A_i) \cdot A_i/(Nd_i^2) = \pi/16N$, which is a constant lower bound for all bins. Thus, we need

$$k \geq \frac{\ln m - \ln(1-s)}{-\ln(1-p)} = \frac{\ln\left(\frac{1296}{\pi} N\right) - \ln(1-s)}{-\ln(1-\frac{\pi}{16N})} \tag{9}$$

sample points to obtain a d_i-sampling in each node v_i, $i = 1 \ldots n$. For a large number of required samples N, we obtain an asymptotic sufficient sample size of

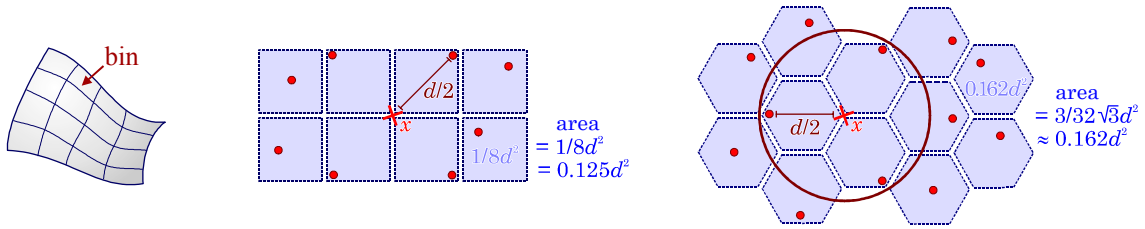
$$k \in \mathrm{O}\left(N(\log N + \log f^{-1})\right)$$

with f denoting the global probability that the sampling fails (i.e. any octree box does not obtain a d_i-sampling).

For nested sampling, the analysis above works with a small modification: We sum all area values of the inner nodes as in full sampling to calculate the oversampling factor and then place all of these points on the levels of the stored triangles only. Afterwards, a constant fraction is moved to nodes above in the hierarchy according to the area value and coverage is guaranteed as for full sampling. However, this strategy leads to a large number of sample points, violating linear memory requirements. Thus, it is not favorable to use purely random sampling with nested sampling. Instead, we should use the stratification techniques described in the next section (Section 4.2.3.2) to generate sample sets: We first create a random candidate set that covers the triangles in the hierarchy safely, disregarding upper levels in the hierarchy. Obviously, this set has size $\mathrm{O}(n \log n)$ for n triangles: As every triangle receives O(1) sample points, we have $N \in \mathrm{O}(n)$ in Equation 9. Then, the stratification algorithm is applied repeatedly to extract sample sets of coarser sample spacing. The stratification will work as long as we retain a safe coverage of the base triangles, which is ensured by our analysis.

A Bound for Flat Surfaces

A lower bound for the sample size can also be derived using heuristic arguments: We divide the surface to be sampled into equally sized cells of area $c \cdot d^2$ with regular structure, as depicted in Figure 31a. The constant c depends on the shape of the grid. The problem is that these cells cannot be easily constructed in a formally strict sense for arbitrary surfaces. Therefore, we assume that the area is flat so that a simple regular lattice can be used. We assume that each cell re-



(a) "bins" on locally flat geometry (square grid)

(b) counting the number of bins assuming a square grid – each bin has an area of $0.125d^2$

(c) counting the number of bins assuming a hexagonal grid – each bin has an area of about $0.162d^2$

Figure 31: Constructing bins on flat surfaces to obtain a tighter (heuristic) bound on the sample size

ceives at least one sample point. Then, the constant c can be determined by measuring the area of the cells if the maximum distance to a point in the plane is $d/2$. For a regular square grid, we obtain $c = 1/8 = 0.125$ (Figure 31b). The resulting bound can be improved slightly by assuming a hexagonal grid, yielding $c = (3/32)\sqrt{3} \approx 0.162$ (Figure 31c). In order to make sure that every cell receives at least one sample point, we again employ the statistical arguments from Section 1.3.3: As we have A/cd^2 cells that obtain a sample point with probability of cd^2/A, we need to choose

$$k \geq \frac{\ln\frac{A}{cd^2} - \ln(1-s)}{-\ln(1-\frac{cd^2}{A})} \approx \frac{A}{cd^2}\left(\ln\frac{A}{cd^2} - \ln(1-s)\right) \overset{\text{hexagonal grid}}{\leq} 6.16\frac{A}{d^2}\left(\ln\frac{6.16A}{d^2} - \ln(1-s)\right)$$

sample points in order to guarantee a d-sampling with probability of at least s. For sampling multiple octree nodes containing surface areas A_i and requiring sampling distances d_i, $i=1\ldots n$, we obtain

$$\begin{aligned} k &\geq \frac{\ln 6.16N - \ln(1-s)}{-\ln\left(1-\frac{1}{6.16N}\right)} \\ &\approx 6.16N\left(\ln 6.16N - \ln(1-s)\right) \\ &\approx 6.16N\left(1.82 + \ln N - \ln(1-s)\right), \text{ with } N := \sum_{i=1}^{n}\frac{A_i}{d_i^2}. \end{aligned} \qquad (10)$$

Implementation Notes

To implement the random sampling procedure, we must choose triangles and octree nodes with probability according to their area value. This can be done in logarithmic time per sample using distribution lists, as described in Section 4.1.2. However, this procedure is rather involved. In practice, a simpler and more efficient algorithm can be used: For each node in which sample points have to be generated, we go through all triangles and compute the expected number z of sample points for each triangle: We compute the number of sample points according to Equation (10) and multiply this number by the ratio between triangle area and the area of all triangles to be sampled. This yields the expected number of sample points for the triangle. According to the integer part $z_{int} := \text{floor}(z)$ of this number, we choose z_{int} random points from the triangle. Then, we choose an additional point with probability proportional to the fractional part $z_{fract} = z - z_{int}$. This procedure is not exactly equivalent to random sampling. It tends to distribute the sample points more evenly among triangles of similar area, as each triangle will always obtain the non-fractional part of its expected number of sample points. In practice, this does not lead to problems, as this tends to yield a more uniform sample point distribution, which is desirable. Addi-

tionally, the modified sampling scheme does not ensure to obtain exactly the desired number of random points. The desired number of sample points is only the expected value of the procedure. However, as the number of sample points is very large, the deviation can be neglected: Obviously, only the randomly chosen fractional parts can lead to a deviation to the desired sample size. Let Z_i denote the random variable that is one if a sample point is chosen with probability p_i and zero otherwise. Additionally, let $k := \sum_{i=0}^{n} p_i$ be the desired number of sample points. Then, the expected value is $\mathrm{E}(\sum_{i=0}^{n} Z_i) = \sum_{i=0}^{n} \mathrm{E}(Z_i) = k$. The variance can be computed similarly: $\mathrm{Var}(Z_i) = p_i(1 - p_i)$. As all Z_i are stochastically independent, we obtain $\mathrm{Var}(\sum_{i=0}^{n} Z_i) = \sum_{i=0}^{n} \mathrm{Var}(Z_i) = k - \sum_{i=0}^{n} p_i^2 \leq k$. Hence, we obtain a standard deviation of no more than $\sqrt{k}$ and thus a declining relative error of $\mathrm{O}(k^{-1/2})$. Therefore, the deviation vanishes for large k. As we use the algorithm only for very large k (typically several million sample points), it is save to use the modified sampling strategy in practice.

The runtime of the modified sampling algorithm is $\mathrm{O}(n+k)$ for n triangles and k resulting sample points. This is more efficient than the $\mathrm{O}(n + k \log n)$ procedure using distribution lists.

4.2.3.2 Stratification

Random sampling leads to considerable oversampling. As shown in the previous section, we obtain an oversampling factor proportional to $\ln n + \ln f^{-1}$ to guarantee a sample point in each of n cells with similar area on a surface with a failure probability below f. In practice, this often leads to oversampling factors of about 15-20 (say e.g. $\ln 1{,}000{,}000 + \ln 0.01^{-1} \approx 18.5$). This means that only every 15th to 20th sample point would be necessary if the sample points were chosen optimally. Thus, it is not favorable to use random sample sets directly for rendering. Instead, we should consider the random point cloud as a "candidate" set and employ a further stratification algorithm to remove superfluous points.

Our postprocessing approaches fall into two different categories: The first are grid stratification techniques that detect superfluous sample points by identifying sample points that are located in the same cell of a regular grid. The second conceptual approach is neighborhood-based point removal: Points can be deleted if their neighborhood contains enough points to satisfy the sampling requirements. Figure 35 illustrates the different possibilities.

4.2.3.3 Grid Stratification

A simple and effective stratification technique is grid stratification. We superimpose a three-dimensional grid and assign the sample points to the grid cells in which the points are located. If multiple points are found in one and the same grid cell, all points except one are deleted. Optionally, we can modify the position of the point in the cell. There are several options such as quantizing the points to the centers of the cells, computing the average of all points, choosing the point closest to the center, or just keep one of the random points (Figure 35b-e).

One Point per Cell

The simplest grid stratification strategy deletes all points in each grid cell except one. To obtain a d-sampling, we start with a random point set that guarantees an ε-sampling, $0 > \varepsilon > d/2$ (see preceding section). Then we superimpose a three-dimensional grid with diagonal $d/2 - \varepsilon$ on the point set. For each cell that receives more than one sample point, the additional sample points are deleted. It is easy to see that this procedure yields a d-sampling (Figure 32): Assume we are given a $d/2$-interior point x on the surface. Now, we have to show that a sample point exists in distance of at most $d/2$ to x. The random sample set originally contained a sample point in a distance of at most ε. This sample point is contained in one of the grid cells. If it was deleted by the stratifica-

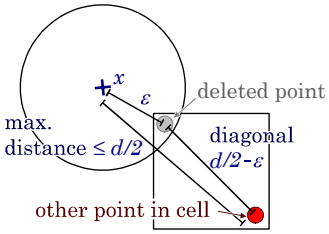


Figure 32: Grid stratification of random ε-sampling.

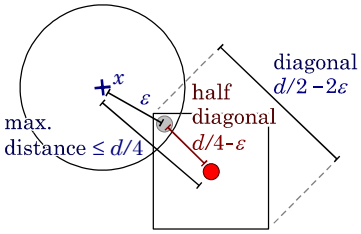


Figure 33: Quantization reduces the sample spacing by a factor of two.

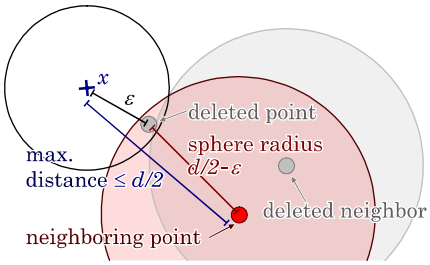


Figure 34: Stratification by removing points that are still covered by neighboring points.

tion algorithm, another point in the same grid cell was retained instead in a distance of at most the diagonal of the grid, i.e. $d/2 - \varepsilon$. Overall, the distance is no more than $d/2$.

Quantization

The variation in point spacing can be reduced by quantizing the point set to the centers of each grid cell, leading to a more uniform point distribution (Figure 33): We use a grid with diagonal $d/2 - 2\varepsilon$ and perform the same procedure as before. Afterwards, all remaining sample points are moved to the center of the grid cell in which they are located. This guarantees that a sample point is found in at most a distance of $d/4$ of a deleted point and thus yields a sample set with sample distance $d/2$, i.e. half the sample distance of that computed by the previous technique. If we use the same grid size as before (i.e. we must use half the sample spacing ε for the random candidate set), this means that we obtain a sample set with half the sample spacing using the same number of sample points. If we apply the sampling procedure to flat surfaces, this effectively reduces the number of necessary sample points by a factor of 4 (and thus usually increases the rendering performance by the same factor).

Averaging / Closest To Center

The drawback of the quantization technique is that points are moved to an arbitrary position in space, potentially away from the original surface. This leads to visible artifacts unless the projections of the sample points are no larger than a single pixel in the rendered image later on. To reduce these artifacts, we can employ a heuristic approximation: Instead of quantizing the points, we take the point that is closest to the center of the grid cell. Alternatively, we can also compute the average of all sample points in a grid cell to determine the sample point position. If we do this with a high density candidate set, we obtain representative point positions for each grid cell that resemble the original surface more closely than quantized points (see Figure 35 for a comparison). Nevertheless, the points still tend to lie close to the center of each grid cell. Thus, we obtain a sample set with roughly a sample spacing of $d/2$. Note that this is only a heuristic: Formally, we can only prove a sample spacing of d, not $d/2$. Nevertheless, the strategy works well in practice. If we apply the strategy but assume a sample spacing slightly below that of quantized points during rendering, we obtain images without holes but still have smaller sample sets than those obtained by simple grid stratification.

Implementation

We have several options for implementing the grid stratification techniques: The simplest, straightforward implementation uses a three-dimensional array of k^3 entries that represent the cells. Each entry in the array stores a list of sample points. The lists are filled by going through the sample set, calculating the array address and storing the point in the corresponding list. Af-

terwards, the points in the list can be processed to choose a representative point for each grid cell. This techniques needs $\Theta(n + k^3)$ time and space for n sample points. Often, we have to deal with flat surfaces, leading to $n \in O(k^2)$ sample points. Thus, the technique is not optimal. It should only be used for grids with small k.

We can improve upon this technique by replacing the array with a hash table [Fischer et al. 98]: We use the concatenation of the sample point coordinates quantized to grid cell centers as hash keys to simulate a large three-dimensional array. This techniques only needs linear $\Theta(n)$ space and can theoretically run in expected amortized $O(n)$ time using randomized perfect hashing (in practice, conventional heuristic hashing schemes are usually employed).

A third implementation technique is based on successive sorting and yields optimal time complexity for $k \in O(n)$. We start by sorting the point list by x-coordinates and form lists of sample points with the same quantized x-coordinate. This can be implemented efficiently using bucket sorting with k buckets for the k different quantized x-coordinates. Then, the process is repeated of the y-coordinates for each of the obtained sublists. The resulting sublists are processed again, now sorting by quantized z-coordinates. This algorithm computes lists of points that lie in the same grid cell in $\Theta(n + k)$ time ($\Theta(n \log n)$ with conventional sorting, as needed for large k).

We have implemented the first and the third strategy: The first strategy is simpler to implement and avoids the overhead of handling lists of lists. Additionally, it allows for more flexibility when computing point attributes (see Section 4.2.4) Thus, we think it is favorable for small k. For larger k, the third strategy can be used. Replacing the array by hash tables would only be necessary for very large grid sizes k. In our applications, we always have to deal with multiresolution hierarchies that contain a large number of different quantized point sets. Therefore, each of those point sets has to be relatively small. Thus, we did not encounter cases in which the grid size k became too large to be handled with either the first or the third strategy.

It should also be noted that it is possible to generate grid-based sample sets without using randomized sampling. [Pfister et al. 2000] perform raycasting from three orthogonal directions with a 2d-grid of rays to create sample points. Afterwards, the points are quantized by storing them in a single LDI. This technique creates similar results as our quantized grid stratification technique. However, the costs of raytracing are unnecessarily high if a conventional raytracing algorithm is employed. Thus, a more effective way is voxelization: A three-dimensional triangle rasterizer can be used to create quantized sample points directly from the triangles [Max 96, Coconu et al. 2002]. This creates a lot of intermediate sample points that have to be removed afterwards: Our goal is to create only a few sample points per triangle (usually not more that 1-3). Thus, we have mostly to deal with triangles that are much smaller than a single voxel. Nevertheless, sample points are created for all of these triangles and double points have to be removed afterwards by an additional grid stratification step. The algorithm uses optimal $O(n + t)$ time and space (n sample points, t triangles). Hence, it is slightly better than our $O(n \log n + t)$ technique based on a random candidate set. The advantage of the random sampling strategy is its generality: It can be generalized easily to cases such as animated scenes (see Section 4.3). This cannot be done straightforwardly using voxelization techniques.

4.2.3.4 Point Removal Stratification

Grid stratification is easy to implement and reduces the number of sample points necessary for a given d-sampling considerably. However, it still has some drawbacks: The regular grid structure can lead to aliasing problems during rendering unless involved image reconstruction techniques are applied. The ratio of point density and guaranteed point spacing is not yet optimal either (see

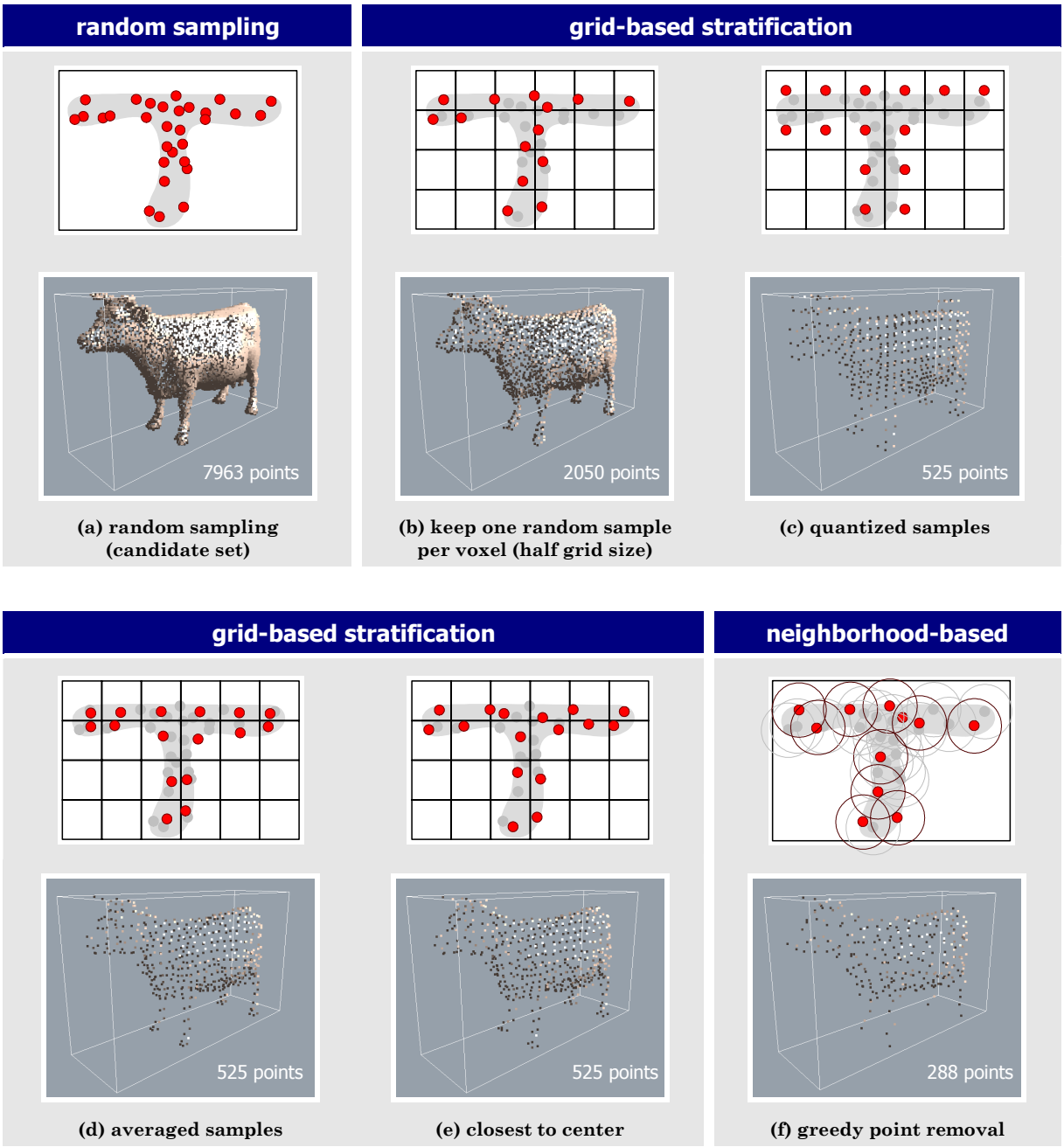


Figure 35: Comparison of sampling and stratification techniques: Random sampling causes the largest oversampling. Different stratification techniques can be employed to optimize random sample sets. The first row shows a schematic 2d illustration of the sample sets. The second row shows an example sample set on the surface of a 3d-model for a fixed sample spacing. Please note that the strategy (b) has to be employed with half the grid spacing (in comparison with c, d, e) in order to obtain the same maximum sample spacing.

next section for a detailed analysis). Additionally, it is not easy to generalize the technique to more general cases such as animated geometry.

As an alternative, we can use a grid-free approach to stratification. The basic idea is very simple: The random candidate set contains a lot of "*unnecessary*" points that can be removed without violating the conditions of d-sampling. A criterion for unnecessary points is neighborhood

coverage. A point can be removed if a neighbor in close proximity exists that preserves the sampling requirements[14]. More formally, we assume that we want to construct a sample set with d-sampling. We start with a random candidate set with sample spacing ε, $0 < \varepsilon < d/2$. Then we think of a sphere of radius $d/2 - \varepsilon$ around each sample point in the candidate set. We say that a point is *covered* by a neighboring point if it is contained in its $(d/2 - \varepsilon)$-sphere. Now we can delete all points that are (1) covered by a neighboring point and (2) are not the only point to cover a neighbor that has already been deleted. This condition assures a d-sampling (Figure 34): If we are given a point x in the interior of a sampled surface, we find a sample point in the original sample set within a distance of at most $\varepsilon < d/2$. If this point has been deleted, a neighbor in distance $d/2 - \varepsilon$ must exist that still covers the point because otherwise it would not have been deleted. Additionally, the neighbor would not have been deleted if it was the last point that covers the deleted point. Thus, we find a sample point in distance of at most $d/2$, proving d-sampling.

The implementation of neighborhood-based point-removal stratification is a bit more involved than that of simple grid stratification: First, we need a data structure that quickly retrieves all neighbors of a point of the candidate set within a distance of $d/2 - \varepsilon$. Our implementation uses a regular grid (a three-dimensional array of lists of points) with a grid spacing of about $d/2$ to accelerate the proximity query. We go through all sample points once in random order. For each point, we check our criterion and delete the point if possible. The performance of the stratification algorithm depends on the distribution of the sample points. The worst case bound is obviously $O(n_c^2)$ for n_c random candidate points if all points lie in close proximity to one another. However, this is not typically observed in practice. For a (locally) flat surface, we can expect $O(n_c/n_s)$ sample points to be reported by the proximity query algorithm where n_s denotes the size of the resulting, stratified sample set. With $\rho := n_c/n_s \in O(\log n_s)$ denoting the oversampling factor of the random candidate set, we obtain a runtime of $O(\rho n_c) = O(\rho^2 n_s) = O(n_s \cdot \log^2 n_s)$. Therefore, the technique is more expensive than grid stratification (which needs $O(n_c) = O(n_s \cdot \log n_s)$ time) but it leads to better results, as shown in the next subsection.

4.2.3.5 Oversampling

In order to compare the quality of the stratification algorithms, we try to quantify the *oversampling*, i.e. the number of "unnecessary" points that remain after stratification. In general, it is a non-trivial problem to quantify the oversampling. It might depend strongly on the concrete shape of the sampled surface. To simplify things, we measure oversampling using a simple model configuration: We assume that a flat surface of arbitrary orientation is sampled. This serves as an approximation for applications with locally flat surfaces (which are often found in practice). To quantify the oversampling, we assign a disc of diameter d (i.e. radius $d/2$) to each sample point that "covers" the surface around the sample point, i.e. fills the area of the (flat) test surface without holes. d should be as small as possible as long as coverage of the surface can be guaranteed (at least with sufficient probability). Obviously, a point sample set with sample spacing d will cover the surface completely[15]. Thus, we can use discs of diameter d for measuring the oversampling of sample sets with sample spacing d. Then, the ratio between the sum of the areas of the discs and the area of the surface to be sampled defines the *oversampling factor*. Ideally, this factor should be one. However, it is not possible to find a configuration in which circles cover a flat surface without overlap. The theoretically optimal configuration for a flat surface is a hexagonal grid

[14] Note that this approach is similar to Poisson-disc sampling [Glassner 95], in which a minimum distance between sample points should be retained.

[15] d-sampling means that a sample point will be found in distance of at most $d/2$ to each point in the interior of the surface. As this point is assigned a disc with radius $d/2$, this point (and thus all points in the interior of the surface) will be covered by such a disc.

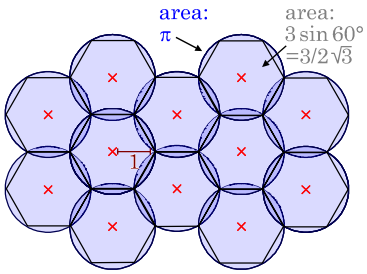


Figure 36: Optimal (hexagonal) sampling pattern for flat surfaces: a circle *covering* with minimum overlap. Oversampling factor $2\pi/9\sqrt{3} \approx 1.21$

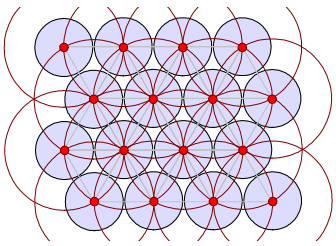


Figure 37: A (hexagonal) tightest *packing* of circles without overlap is the worst case configuration for neighborhood-based stratification.

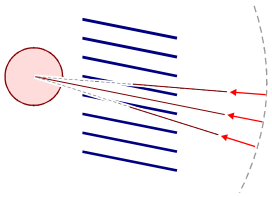


Figure 38: Venetian blind example [Chamberlain et al. 95]. Occlusion can be highly direction-dependent.

(see Figure 36), leading to an oversampling factor of $2/9\pi\sqrt{3} \approx 1.21$ [Williams 79]. To quantify the performance of stratification techniques, the oversampling factor should be compared with this optimal value.

For our evaluation, we determine bounds for the worst case oversampling analytically. Additionally we measure the average oversampling empirically: We create a high density sample set on a sphere. For small sample spacing d, the sphere appears to be locally flat and thus we obtain an approximate estimate for the estimated oversampling of a plane with random orientation. Table 1 summarizes the results.

Grid Stratification

To determine the worst case oversampling of the grid stratification method, we have to determine the configuration that leads to the largest density of sample points on a planar object. Obviously, the grid cells that intersect with the sampled object can receive one sample point each. In the worst case, it might be possible that every grid cell that intersects with the object receives a sample point. Thus, the question for the worst case oversampling is equivalent to finding the configuration under which a plane intersects with a maximum number of cells of a cube grid (per unit area). To construct an upper bound for this number, we consider the unit normal vector $n = (d_x, d_y, d_z)$ of a plane that intersects the stratification grid: Without loss of generality, we assume that the z-coordinate has the largest absolute value of all coordinate entries and that the grid cells have a side length of 1. Now we can express the height z of the plane as linear function $z(x,y) = ax + by + c$ of the x and y-coordinates. As $|d_z| > |d_x|$ and $|d_z| > |d_y|$, we also know that the slopes a and b have an absolute value of at most 1. Now we consider one grid cell in the x-y-plane: Within one such cell $[x_c, x_c+1]\times[y_c, y_c+1]\times\mathbb{R}$, the z value can vary by at most $1\cdot1 + 1\cdot1 = 2$. This means that the plane cannot intersect with more than three different cells in z-direction while staying in the same x-y-cell. Next, we have to find a lower bound for the area "used" for intersecting with these cells. As we cover a complete x-y-cell, we will have to use at least 1×1 area units, i.e. an area value of one[16]. Thus, it is impossible to obtain more than 3 sample points for 1 area unit.

For simple grid stratification, the worst case sample spacing d is given by twice the diagonal of the grid. Thus the diagonal $\sqrt{3}$ of the grid corresponds to $d/2 - \varepsilon$ (see Section 4.2.3.3). Using this scale factor, the side length of the grid corresponds to $(d/2 - \varepsilon)/\sqrt{3}$ and we obtain an upper bound of at most 3 sample points per $(d/2 - \varepsilon)^2 / 3$ area, i.e. $9 / (d/2 - \varepsilon)^2$ points per unit area. Each point is assigned a disc of area $\pi d^2/4$. Hence, we obtain an oversampling factor of

[16] Of course, our bound is not tight: If we want to intersect with more than 1 cell in z-direction, we are forced to use slopes of absolute value greater than zero, which increases the "used" area value.

$$9\pi\frac{(d/2)^2}{(d/2-\varepsilon)^2}\overset{\varepsilon\to 0}{\geq} 28.27\,.$$

For quantized grid stratification, the maximum sample spacing is only half as large. This means that the maximum oversampling factor is only a quarter of that value, i.e. 7.07 for $\varepsilon \to 0$.

The oversampling depends strongly on the orientation of the surface in respect to the quantization grid. An empirical measurement of the average oversampling on a spherical object (measured using a 32^3 quantization grid of the bounding box of the sphere) yields 13.4 and 3.45 for grid stratification without and with quantization, respectively. Thus, the average values are about halve as large as our upper bounds.

Point Removal Stratification

The next strategy that we have to analyze is neighborhood-based point removal. The algorithm deletes all points that are still covered by neighbors (and are not needed themselves to cover deleted neighbors). Therefore, the worst case configuration consists of a tightest packing of points that cannot be deleted, i.e. that are not within the $(d/2 - \varepsilon)$-disc of one another. Equivalently, we can form a $(d/2 - \varepsilon)/2$-disc around each sample point and demand that these discs are all disjoint. This means, that we are looking for a tightest packing of circles with radius $r := (d/2 - \varepsilon)/2 + \delta$ (with arbitrarily small $\delta > 0$), which is a well known problem. The solution is (again) a hexagonal grid [Williams 79] (see Figure 37). To calculate the number of sample points per unit area, we consider a single triangle in Figure 37 that connects three adjacent sample points. This triangle has an area of $\sqrt{3}r^2$ and accounts for half a sample point. In other words, we obtain $(1/2)/(\sqrt{3}r^2) = \sqrt{3}/(6r^2) = \sqrt{3}/(6\,((d/2-\varepsilon)/2+\delta)^2)$ sample points per unit area. Letting $\delta \to 0$ to establish an upper bound, we obtain $2\sqrt{3}/(3(d/2-\varepsilon)^2)$ sample points per unit area, which are assigned discs of area $\pi d^2/4$. Overall, we obtain an oversampling factor of

$$\frac{2}{3}\pi\sqrt{3}\frac{(d/2)^2}{(d/2-\varepsilon)^2}\overset{\varepsilon\to 0}{\geq} 3.63\,.$$

This value is considerably smaller than the bounds for the grid stratification techniques. However, these are only conservative estimates while the bound for neighborhood-based stratification is tight. To compare the oversampling under realistic conditions, we have also measured the average oversampling on a sphere (a unit sphere and $d = 1/32$). The measurements yield an average

sampling strategy	construction time	worst case oversampling	average oversampling (empirical)
random sampling:	$O(n) \subseteq O(N \log N)$	$\leq 4.84\left(\ln 6.16N + \ln f^{-1}\right)$	≈ 20-100 (depending on f)
grid stratification: one point per cell	$O(\varepsilon^{-2}n\ log\ \varepsilon^{-2}n)$ $[n \in \Theta(N)]$	$\leq 9\pi\frac{(d/2)^2}{(d/2-\varepsilon)^2}$ (28.3 for $\varepsilon \to 0$)	13.40
grid stratification: quantized	$O(\varepsilon^{-2}n\ log\ \varepsilon^{-2}n)$ $[n \in \Theta(N)]$	$\leq \frac{9\pi}{4}\frac{(d/2)^2}{(d/2-\varepsilon)^2}$ (7.1 for $\varepsilon \to 0$)	3.45
point removal stratification:	$O(\varepsilon^{-2}n\ log^2\ \varepsilon^{-2}n)$ $[n \in \Theta(N)]$	$\frac{2}{3}\pi\sqrt{3}\frac{(d/2)^2}{(d/2-\varepsilon)^2}$ (3.63 for $\varepsilon \to 0$)	1.61

Table 1: Comparison of sampling strategies for a large, flat surface of area A, neglecting borders. n denotes the resulting number of sample points, including oversampling. d is the sample spacing of the resulting sample point set. f is the failure probability for the random sampling step. To make the construction times more comparable, we also consider $N := A/d^2$ as complexity parameter, which is proportional to the minimum required number of sample points.

oversampling of 1.61, which is about half the value of that for quantized grid stratification. It already comes close to the theoretically optimal value of 1.21. As the outcome of random order point deletion is not deterministic, we repeated the measurements several times. The deviation from the average value was very small: For an average 6600 points sample set, we obtained a standard deviation below 1% after 10 trials. This outcome could have been expected: As a large number of points are handled mostly independent of one another, different point densities due to bad sample deletion orders should cancel out for large point sets.

Other Reasons for Oversampling

Another source of oversampling is the fixed factor between adjacent sample spacings in the hierarchy. Due to the octree structure, the sample spacing is always decreased by a factor of two between a node and its child node. This means that the sampling density on a flat surface is increased by a factor of 4 between hierarchy levels. Thus, in the worst case, we obtain an additional oversampling factor of up to a factor of 4 if we need a sample spacing of d but the closest node only provides $d + \varepsilon$ so that we are forced to use the child nodes. Even on the average, assuming that all sample spacings are needed with similar probability, we still obtain an expected oversampling factor of

$$\int_1^2 s^2 ds = 2\frac{1}{3}.$$

In order to reduce the oversampling due to discrete resolution steps, we must reduce the factors between levels of resolution in the hierarchy. However, the regular subdivision structure of the octree enforces a stepping of at least a factor of 2 between sampling distances (larger steps could be created by increasing the branching factor of the tree). Thus, we store multiple point sets per octree node[17]. For k different point sets, a factor of $r = \sqrt[k]{2}$ between the sample spacing is employed, leading to a worst case oversampling (due to resolution mismatch) of $2^{2/k}$. Obviously, this leads to a trade-off between storage costs and performance. For a single flat surface, we expect storage costs of

$$\#sample\ points \le \sum_{i=0}^{\infty} np_{\max}\left(r^{-2}\right)^i = \frac{np_{\max}}{1-r^{-2}} = \frac{np_{\max}}{1-2^{-2/k}}.$$

In practice, values of $k = 2$ or 3 lead to a significant performance enhancement at moderate additional storage costs (the number of sample points is increased by an expected factor of 1.5 and 2.02, respectively). Please note that the problem of discrete resolution steps does not apply to the dynamic sampling data structure but only to precomputed sample sets.

4.2.4 Point Representations and Point Properties

Up to now, we have created a hierarchy of triangles and surface sample points. In this hierarchy, each sample point represents a piece of geometry, ranging from a fragment of a single triangle to a large set of triangles. Now we have to decide which properties should be stored for every sample point to represent the substituted geometry.

[17] The technique can only be used for full sampling, not nested sampling.

4.2.4.1 Point Samples

A straightforward option is to use point samples in a strict sense: Each point primitive represents a single point of the sampled surface and thus reports only the local properties at that point. Besides the position in space, these are also the properties necessary to perform lighting calculation (material identifier, probably also a normal vector or texture colors). The dynamic sampling data structure (Section 4.1) always yields simple point samples. For static sampling (Section 4.2), we have the option to prepare and store additional and/or enhanced attributes for each sample point during preprocessing.

4.2.4.2 Prefiltered Samples

Obviously, simple discrete point samples can easily lead to aliasing (regular sample patterns) and noise (in case of irregular sampling) issues. [Pfister et al. 2000] propose employing a prefiltering approach during preprocessing to avoid these problems: A point cloud is a discretely sampled representation of a continuous function defined on a surface. Thus, the function has to be band limited with a suitable low-pass filter in order to avoid aliasing artifacts (see Section 1.3). For regular sampling patterns, such as those created by the quantized grid stratification technique, we can directly employ the well-known signal processing framework to derive a suitable filter kernel. A typical choice is a Gaussian function with a standard deviation in the range of half the grid spacing. For irregular sample patterns, we just substitute the maximum sampling distance for the grid spacing and use again a corresponding three-dimensional Gaussian filter kernel. Color properties on the surface are now not point sampled but a weighted average of adjacent colors is stored instead, using the Gaussian as weighting function. To determine this convolution integral numerically, we use a simple Monte Carlo integration approach: We increase the density of our random candidate set prior to stratification. Then, we select a subset of "representative" points with one of the stratification techniques. For each of those representative points, we determine all of the original points within the support of the Gaussian around the representative point using a grid search structure, as described in Section 4.2.3.4. The color attributes of these points are averaged and the result is stored for the representative point. In the case of grid stratification, the process can be speed up significantly: Instead of creating intermediate points and then searching for them later on, we directly write the color information (weighted by the Gaussian) into the cells of the stratification grid. In both cases, an additional sum of the weights is used to renormalize the calculated values later on. The Monte Carlo integration is not optimal; we need a relatively large sample set to avoid noise artifacts (oversampling factors of 100-300 yield good results). However, as this is done during preprocessing, the performance is not a critical issue. The technique could perhaps be improved by using more elaborate numerical integration techniques, applied directly to the underlying triangles. However, due to the possibly highly irregular structure of the represented geometry, these techniques can not easily provide a better convergence than stochastic sampling either.

4.2.4.3 Higher Order Surface Approximations

In addition to color attributes, we also often need to represent local geometric properties: For lighting calculations, we need e.g. at least a normal vector to compute the light reflected from light sources to the viewer. Of course, we would also like to do prefiltering for such attributes in order to avoid aliasing of surface properties as well. However, simple averaging does not work for higher order geometric properties. For example, the average of the normal vectors of surface with rough microstructure (such as waves on a water surface) yields a uniform normal vector, removing the roughness of the original surface [Schilling 2001]. The problem is that we need to represent the distribution of the normal vectors rather than an average direction. As the

representation of the exact surface normal distribution is usually too costly, we need a simple model: The simplest variant is just the average direction, which works well surfaces that are locally smooth on the corresponding scale. Even if this is not the case, missing roughness might still be better than aliasing artifacts. An improved representation was proposed by [Kalaiah and Varshney 2001]: A second order surface is fitted to the surface in a local coordinate frame and the directions of minimum and maximum curvature as well as the curvature values are stored. This allows the reconstruction of local curvature information during rendering, which can be used e.g. for improved shading [Kalaiah and Varshney 2001]. We use these differential properties to control the size of secondary ray cones in our point-based multi-resolution raytracing approach described in Chapter 6. However, second order surface representations are also just able to represent locally smooth surfaces. A more general model is described by [Schilling 2001] in the context of environment mapped bump mapping: In addition to surface curvature, roughness parameters are stored at each sample point. The roughness parameters are the two main axis of a two-dimensional Gaussian distribution that has been fitted to (a planar projection of) the normal vectors in the neighborhood of a sample point.

4.2.4.4 The Visibility Problem

An important aspect of a sample point is its occlusion: During rendering, the visibility of sample points has to be resolved by examining other sample points in front potentially occluding them. In general, the occlusion of a complex set of geometric objects can depend strongly on the viewing direction, which makes a representation by a simple set of point attributes infeasible: A simple example to illustrate the problem is a Venetian blind ([Chamberlain et al. 95], see also Figure 38): The structure is opaque for all angles but a very small angle interval in which it rapidly becomes transparent. Thus, [Chamberlain et al. 95] conjecture that no fixed size representation exists that represents the occlusion properties of an arbitrary piece of geometry exactly. Note that as long as no exact representation of occlusion is known, it is not possible to guarantee correct rendering results for point-based simplifications of complex models either. Therefore, we always have to deal in general with heuristic approximations of potentially unknown quality if complex geometry is substituted with constant sized pieces of information (i.e. “points” with attributes). All proposed solutions with bounded memory costs are just such heuristics. Nevertheless, good images can be rendered in practice as often the errors concerning micro-occlusion effects are not important to a human observer.

An elementary representation of occlusion could be to assume a solid sphere with the diameter of the maximum sample spacing around each sample point. If the surface is locally flat, a tangent disc can be used instead as a first order approximation, delivering more accurate results [Pfister et al. 2000, Rusinkiewicz and Levoy 2000, Zwicker et al. 2001a]. For more general surfaces (such as a set of small branches of a tree, now represented by a single sample point), this technique is not well-suited. For such cases, we use a conservative transparency heuristic: We assign an alpha value to each sample point that measures the percentage of occluded area. This value is view-dependent but as we want to avoid holes in the image later on, we use a conservative upper bound: For d-sampling, we compute the area in the $d/2$-neighborhood of the sample point and compare it to the minimum value of $\pi d^2/4$ for a closed surface. The fraction of measured and minimal surface area (clamped to 1) is treated as alpha (opaqueness) value. This ensures that closed surfaces never show holes, which is important as such artifacts are easily noticeable (this was an open issue of the technique proposed by [Chamberlain et al. 96]). During compositing, we do not know anymore the positions of the occluding surfaces. Thus, we should use the sum of alpha values (rather than alpha blending) to determine the occlusion of multiple sample points on one line of sight to retain a conservative visibility bound. However, it turns out in practice that

simple alpha blending is also satisfactory because our criterion always guarantees opaqueness for every single sample point belonging to a closed surface. An underestimation of opaqueness can only occur if multiple, disjoint layers of geometry interact at a micro-occlusion level. These cases are hard to distinguish for practical scenes so that alpha blending also yields good results (although neither being correct nor at least strictly conservative).

The computation of the alpha values can be integrated easily in our sampling framework: We use the same high density random sample set that was used for estimating color and curvature properties and count the neighboring points in the neighborhood in order to estimate the local area.

4.2.4.5 More Complex Representations

We could of course employ more complex representations than simple attributes like alpha or curvature values. An option for more sophisticated representations could be for example the usage of small light fields (storing view-dependent transparency, color, normal vector and depth information). This would allow the reconstruction of images at lower costs, using less sample points than those that are necessary with simple attributes. However, the storage costs also increase correspondingly. Obviously, choosing the point attributes is a trade-off between precomputation time and space and the number of sample points necessary for a certain image quality, i.e. the rendering time. Our goal was to create representations with low memory overhead. Therefore, we opt for not using expensive point representations beyond simple models with few parameters. Other techniques could be subject of future work.

4.2.5 Instantiation

Instantiation of static sampling data structures can be performed similarly as for the dynamic sampling data structure (Section 4.1.6): A complete octree with triangles and precomputed sample points is treated as an individual object and inserted into a higher level data structure. Instead the extends of triangles, the root bounding boxes of instantiated octrees are considered for sorting the object into the data structure. The depth at which the instance has to be stored is determined by comparing the sampling distance of the root node with that of the nodes in the hierarchy. If a leaf node is reached before the sampling distance is short enough, the sub-octree is inserted into that leaf node (similarly to a short-cut) in case of nested sampling. For full sampling, the intermediate nodes are generated until the sampling density matches, i.e. the sampling distance is not smaller than two times that of the leaf node. In order to limit the branching factor of the instantiated tree, it is also beneficial to limit the number of instances that are stored in one leaf node. Thus, beside the parameter n_{max} that controls the maximum number of triangles, a second parameter $inst_{max}$ is introduced to control the maximum number of instances in a leaf node until it has to be split. Typically, this value is much smaller than n_{max} (n_{max} ~ 50-5000, $inst_{max}$ ~1-8).

To generate sample points from instances, two techniques can be used: First, new sample points can be generated using a precomputed distribution list of all triangles in the instance. However, it is more convenient to just use the sample points from the root node (or the first few levels) in the hierarchy and use them for resampling.

If the precomputed sample points use prefiltered color attributes (or similar precomputed surface attributes), the set of applicable instantiation operators is more restricted than for simple points samples: It is still possible to perform geometric transformations, e.g. affine mappings. However, operations that change the appearance of the surface, such as exchanging materials by

identifiers to create differently colored objects from one base model, might lead to problems as the instantiation transformation cannot be applied any longer after averaging samples from multiple surface points. Thus, the increased image quality comes at the cost of less flexibility in modeling.

4.2.6 Caching

Dynamic sampling and static sampling both have certain advantages and disadvantages (see Section 4.4). For rendering real-time animations, the two techniques and their advantages can be combined by using a static data structure as cache for dynamically generated sample points. The same concept is also useful for other applications such as rendering from data stored out-of-core or rendering compressed volume data[18].

The idea is fairly simple: As the generation of sample points using dynamic sampling and distribution lists is rather expensive, it makes no sense to recreate all sample points from scratch during an animation. Instead, we can expect that the majority of the created sample points from one frame is still adequate for rendering the next frame. Only some points have to be recomputed. The validity depends on the relative movement: If the projected size on the screen has not changed by more than a small constant, we will usually still accept the point set. This change of the projected size depends on the movement of the viewer and on the position of the objects in the scene: Parts of the scene that are far away usually change less frequently than parts of the scene in close proximity to the viewer. Therefore, we need an adaptive caching strategy. This is achieved by associating the caches with the nodes of the octree: Each node in the dynamic sampling data structure carries a list of cached point clouds. If a point cloud with sample spacing d is requested, we first make a look up in the cache whether we already have an old point cloud with a sampling distance not larger than d and not smaller than $\varepsilon \cdot d$ (with $\varepsilon > 0$, typically $\varepsilon = 1/2$). If a cached version is found, it is returned to the rendering algorithm. Otherwise, a new sample set is created and stored in the cache prior to reporting it to the rendering algorithm. Note that this scheme also works with instantiation as we allow multiple resolutions to be stored for each node. To avoid memory overflows, we have to limit the size of the cached data. We do this using a simple least-recently-used (LRU) heuristic to delete cache entries that have not been used for the longest time.

In addition to just caching the sample sets, we can also process them further in order to increase the rendering performance. For example, we can apply a stratification algorithm to the random sample set. Grid stratification techniques are fast enough to be performed on the fly. We could also do prefiltering. However, we need fairly large sample sets to obtain low noise results so that this might be out of scope for real-time applications. Another processing option is optimization for the rendering process. This includes e.g. the creation of display lists for hardware accelerated graphic systems or the creation of incremental structures such as LDIs for efficient software rendering. Decompression from a more compact encoding is also an option, improving the memory efficiency of the data structure.

Caching makes even sense for precomputed sample sets: If hardware accelerated rendering is used, only a working set of recently used point and triangle data is downloaded as display list or vertex buffer to the graphics card. This is necessary as such graphics devices usually only have limited memory resources Often, this virtualization is already done in the graphics driver. However, we then still have duplicates of non-downloaded buffers in main memory, which can be avoided by manually controlled caching. An additional option is sorting: If we render alpha-

[18] The idea of hierarchical multi-resolution caching is not new, see e.g. [Shade et al. 96].

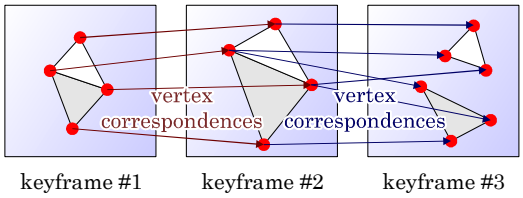


Figure 39: Input model for animated scenes: a sequence of keyframe meshes; the vertices are connected via correspondences.

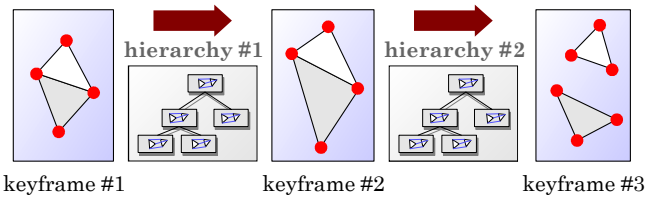


Figure 40: An "interpolating hierarchy" is placed between each pair of adjacent keyframes.

blended points (as proposed in Section 4.2.4.4), we have to sort the points in depth order. However, small deviations of the viewing direction usually do not demand for an immediate reordering but are invisible. Thus, we can cache sorted point lists until the viewing angle changes by more than a fixed value, e.g. by more than 20°. A (multi-level) caching concept will also be used for the volume rendering technique and the out-of-core data structure discussed in Chapter 8.

4.3 Animated Geometry

The data structures presented up to now can handle static scenes only. In many applications we have to deal with animated content, not only static scenes. Typical examples are crowd animations: Large crowds of animated humans or animals are moving dynamically within a static scene. Our goal is to speed up the rendering of the animated content as well so that complex scenes such as large crowds can be displayed in output-sensitive time.

4.3.1 Input Model

To devise a general technique for multi-resolution rendering of animated content, we first need a model for the input. A lot of different modeling techniques for animations are known in literature, often highly specialized for certain tasks such as facial animations, smoke effects and others. A general technique to specify animated surface models is the usage of keyframes: A keyframe animation consists of a sequence of triangle meshes of arbitrary topology and connectivity. For each pair of consecutive keyframes, correspondences between the vertices of the triangles must be specified, i.e. every vertex of a keyframe must be assigned a matching vertex in the other keyframe. During animation, the positions (and all other vertex attributes, like normal or color) are interpolated between the keyframe values. In our case, we will restrict the discussion to linear interpolation; a generalization to higher order interpolation should be unproblematic. Triangles can be created or deleted by blending from one vertex position to three different positions and vice versa. The specification of vertex correspondences is part of the input to the algorithm, i.e. they are not established automatically but they must be specified by the user during modeling. Usually, these correspondences are known to the modeling algorithm that creates the input scene.

4.3.2 Hierarchy Creation

4.3.2.1 Organization

To define a multi-resolution hierarchy, we consider pairs of adjacent keyframes in the animation sequence. For each such pair, we construct one "*interpolating hierarchy*" that represents all possi-

ble configurations for timesteps in between the two keyframes. If each such interpolating hierarchy uses O(n) space (n being the number of triangles in the two keyframes), we obtain linear memory usage for a simple linear sequence of keyframes[19].

This organization reduces the problem: Our task is now to provide a multi-resolution hierarchy for sets of triangles with vertices moving on a linear path. Topological changes happen implicitly at keyframe transitions by dropping or introducing zero area triangles for the next keyframe pair, respectively.

4.3.2.2 Interpolating Hierarchies

To create a spatial hierarchy for these deforming triangle sets, we use a simple heuristic: We create a conventional octree for the triangles at the start of the time interval, as described before. We keep this partition of the input scene over the time interval of the keyframe pair and compute the bounding box of the corresponding triangle groups at the end time. The bounding box at any time in between the two keyframes is then constructed by interpolation:

4.3.2.3 Bounding Box Interpolation

To determine the extends of a set of triangle moving over time, we use a simple conservative estimate: Due to the linear paths, we can just interpolate the bounding boxes linearly and obtain and upper bound of the volume in which the triangles may be located at any position in time. This interpolated bounding box is used as bounding box of the interpolated octree at any time in between the two keyframes. The proof of this property is simple: We consider a set of points moving on linear path (the bounding box of a set of triangles is the same as the bounding box of their vertices). Without loss of generality, we consider only the lower bound of the x-coordinate of the vertex set. Let $x_{min}(t)$ be the minimum of all x-coordinates at time step t and $x_i(t)$ be the coordinate of point i ($i = 1\ldots n$) at time t. Now we have to show that $x_{min}(t) \geq (1-t)\cdot x_{min}(0) + t\cdot x_{min}(1)$. This can be shown by considering the definition of the $x_{min}(t)$:

$$
\begin{aligned}
x_{min}(t) &= \min_{i=1..n} \underbrace{\left[\underbrace{(1-t)}_{\geq 0} \cdot \underbrace{x_i(0)}_{\geq x_{min}(0)} + \underbrace{t}_{\geq 0} \cdot \underbrace{x_i(1)}_{\geq x_{min}(1)}\right]}_{\geq (1-t)\cdot x_{min}(0) + t\cdot x_{min}(1)} \\
&\geq \min_{i=1..n} \left[(1-t)\cdot x_{min}(0) + t\cdot x_{min}(1)\right] \\
&= (1-t)\cdot x_{min}(0) + t\cdot x_{min}(1)
\end{aligned}
$$

Every x-coordinate in the minimum function is a convex combination of the keyframe coordinates. By substituting these by lower bounds $x_{min}(0)$, $x_{min}(1)$, the sum can at least become smaller. Thus, it is always not larger than the original minimum.

Similar arguments can be applied to the other coordinate directions and to the upper bounds. This shows that the interpolated axis aligned bounding box (which is the Cartesian product of three real intervals, i.e. six bounds) is always a conservative bounding volume for the complete time interval.

4.3.2.4 Limitations

The interpolated hierarchies rely on a heuristic. They only work if we have some temporal coherence. If all vertices were moved to random positions between two keyframes, the hierarchy would

[19] If a more complex graph of keyframe transitions is used, up to O(k^2) hierarchies can be necessary for k keyframes.

be destroyed soon after the start time. However, such a situation is rarely found in applications. Typically, triangles that are located in close proximity to one another will probably move in a similar direction. If this is not the case because the group is too large, we can perform subdivision to improve the spatial locality. In practice, the worst case is that groups of triangles move in opposite directions, for example parts of the two legs of a walking human. In this case, the bounding volume grows if we move forward in time. If it becomes too large, the box will be subdivided during rendering. The spatial subdivision will then (perhaps in one or two steps) divide the moving parts from each other. The coherence could be improved by a preclassification by motion vector, similar to area groups or orientational groups (Section 4.1.4/4.1.5). However, to our experience, the simple hierarchy interpolation heuristic works well in practice, such classification techniques do not seem to be necessary. The most important reason for that is probably that the hierarchy interpolation is only applied in between two adjacent keyframes. Each keyframe interval usually represents only a small portion of the complete movement sequence. At every keyframe position, the hierarchy is rebuild so that distorting developments can only evolve within small time steps.

4.3.3 Sampling

For handling animated geometry, we generalize the static sampling data structure. The construction of the point hierarchy is analogous to the static case, just substituting interpolated hierarchies for static hierarchies. Two modifications of the static hierarchy creation algorithm are necessary: First, we should determine large triangles during octree creation. These are triangles with either an area corresponding to more than p_{max} sample points or triangles exceeding the node tolerance zone. To do so, we compute the maximum area and bounding box of each triangle during the animation and the minimum extends of the node bounding box. If the x-, y-, or z-interval of a triangle exceeds the minimum size of the start and end intervals of the nodes bounding boxes by more than a factor δ, the triangle is stored at the parent node. The maximum number of sample points is determined by dividing the maximum area by the sampling spacing squared, multiplied with a constant accounting for the expected oversampling of the stratification technique. The sample spacing is always determined by the start bounding box (a constant fraction of the side length of the cube of the octree node). This assures that the sample spacing is decreasing by a constant factor of two between two octree levels[20].

The second addition we need is a sampling strategy for animated geometry. It has to create point sets that cover the surfaces safely over time, i.e. provide a d-sampling for a given sample spacing d, although the geometry is deforming. Considering the sampling strategies discussed in Section 4.2.3, we see that grid-based sampling strategies are not suitable for processing animated scenes: The geometry would move independently of the spatial grid so that the sample sets would not be well defined anymore (unless we do a steady resampling, creating large numbers of redundant sample points over time). A suitable strategy is random sampling supplemented by neighborhood-based point removal stratification. In the following, we discuss how to adapt this stratification strategy for use in animated sampling.

4.3.3.1 Candidate Set

Again, we first have to create a candidate set that can be processed by the stratification algorithm. We use the same random sampling technique as described before in Section 4.2.3, with minor modifications: The original sampling algorithm uses the area of a triangle to determine the

[20] We can of course also use multiple sample spacings with a factor 2^{-k} per node to improve oversampling properties, as discussed in Section 4.2.3.5.

number of sample points to be placed on each triangle. More precisely, the ratio of the triangle area and the sum of all triangle areas is the probability for receiving a sample point. For animated scenes, the area changes dynamically over time. In order to be conservative, we compute the maximum area value for each triangle during the animation. We then use this value to compute the number of random sample points the triangle should obtain. This leads to two questions: First, we need an algorithm for computing the maximum area. Second, we need to assure that the strategy guarantees a save coverage of all triangle area through the keyframe period.

Area Computation

To perform the sampling algorithm, we first need to determine the maximum area of a triangle during a time interval. For linear interpolation, this can be done analytically: Let $a(t)$, $b(t)$, $c(t)$ be the vertices of a triangle over time. At any time t, we can express the square of the area of the triangle by the scalar of the cross products of two side vectors with itself:

$$\begin{aligned} A^2(t) &= \frac{1}{4}\big((b(t)-a(t))\times(c(t)-a(t))\big)^2 \\ &= \frac{1}{4}\big(((1-t)b(0)+tb(1)-(1-t)a(0)-ta(1))\times((1-t)c(0)+tc(1)-(1-t)a(0)-ta(1))\big)^2 \\ &= \frac{1}{4}\left(\begin{bmatrix}(1-t)b_x(0)+tb_x(1)-(1-t)a_x(0)-ta_x(1)\\(1-t)b_y(0)+tb_y(1)-(1-t)a_y(0)-ta_y(1)\\(1-t)b_z(0)+tb_z(1)-(1-t)a_z(0)-ta_z(1)\end{bmatrix}\times\begin{bmatrix}(1-t)c_x(0)+tc_x(1)-(1-t)a_x(0)-ta_x(1)\\(1-t)c_y(0)+tc_y(1)-(1-t)a_y(0)-ta_y(1)\\(1-t)c_z(0)+tc_z(1)-(1-t)a_z(0)-ta_z(1)\end{bmatrix}\right)^2 \end{aligned}$$

This expression obviously yields a 4th degree polynomial in t (the cross product creates quadric terms, which are squared again by the scalar product). As the resulting function of t is C^∞ differentiable, a global maximum is either found at the interval limits $t = 0$, $t = 1$ or at a local maximum with a derivation $dA^2(t)/dt$ of zero. The values at the interval borders are easy to compute, all potential local minima can be found by computing the derivative (which is a 3rd order polynomial) and solving analytically for $dA^2(t)/dt = 0$. This can still be done with a closed formula. This yields the time with maximum area.

Coverage

To analyze the coverage properties of the sampling algorithm, we first fix an arbitrary position in time t. As we now have a static triangle mesh, we can apply our analysis from Section 4.2.3.1: The conservative area estimate can only increase the probability of receiving sample points for each triangle. Thus, the probabilities for the "bins" receiving "balls" in the proof of Section 4.2.3.1 can only increase and thus the proof still holds, for any static frame from the animation.

The problem is that the mesh is potentially deforming over time. It is still uncertain how we can obtain the same failure probability throughout the complete animation sequence. To ensure save coverage, we make a very conservative estimate: We subdivide the time interval into k small time steps Δt. We demand that the vertices of the triangles are not moving by more than ε, with ε being a small fraction of the sample spacing d. If we can guarantee a d-sampling somewhere in each time interval, we are sure to retain a $(d + \varepsilon)$-sampling overall. The number N of time intervals depends on the movement of the triangles. This number is typically not very large: For the following analysis, it is sufficient to consider only movements relative to the center of the interpolated bounding box. Then, the maximum movement is bounded by the maximum diagonal extend of the bounding box divided by ε (typically being a constant fraction of d). The maximum extends usually differ from the extends of the start bounding box only by a small factor. If the ratio is very large, we probably lose the spatial locality of our hierarchy, which is a much more

serious problem than local coverage problems. Thus, limited movement relative to the scale of the octree node might be a realistic assumption for practical applications.

This consideration defines N different frames with potentially slightly different geometry. Now we make the very conservative assumption that all frames show a totally different geometry, such as moving all triangle vertices to random positions, independent of one another but still conserving the computed area values for the triangles. If this was the case, each submesh will still show a d-sampling with probability s (i.e. failure probability $f = 1 - s$). As the computed point set is a random one, independent of the mesh, it will cover any triangle mesh obeying to the assumed area constraints with probability s, even if it was chosen randomly. If all submeshes were chosen stochastically independently of one another, we would obtain a failure probability of $f' = Nf$. However, the meshes at the different time steps are not independent of one another. Indeed, they are highly correlated. Nevertheless, the sum of probabilities is still a conservative bound for the probability of the union of the events (i.e. at least one failure in time) [Motwani and Raghavan 95]. Thus, we have $f' \leq Nf$. This means, we must choose $f := f'/N$ in order to retain a maximum probability of f' for the sampling process to fail. This increases the oversampling factor: The oversampling factor according to Equation 10 is $\ln 6.16n + \ln f'^{-1}$ for n "bins" (i.e. $\Theta(n)$ sample points after stratification). This factor is now enlarged: Instead of $\ln f'^{-1}$, we have to add $\ln Nf'^{-1} = \ln N + \ln f'^{-1}$ to the $\ln n$ term in the oversampling. Thus, the oversampling is increased by the logarithm of the number of time steps. Even for very large numbers of N (say several million), the logarithm is still in the range of the logarithm of the target number of sample points $\ln n$ and thus the performance of the sampling algorithm is not severely effected (even in that exaggerated case it just doubles). We can also measure N during sampling according to a user defined ε and control the sampling process in order to obtain an upper bound for the sample size.

The consideration above is of course very conservative. Due to the strong stochastic dependence of coverage probability of the different submeshes on one another, we do not expect to need a strong additional oversampling for animations in practice.

4.3.3.2 Stratification

Now that we have a candidate set of sample points that guarantees a d-sampling over time for any given $d > 0$, we can employ a stratification algorithm to improve on the oversampling factor of the sample points. Neighborhood-based point removal can be extended easily to animated scenes: The algorithm deletes points that are still covered by neighbors and that are not needed as being the last to cover a previously deleted neighbor. In the static case, covering means being located in a $(d/2 - \varepsilon)$-sphere. For the animated case, we just have to extend the notion of coverage: An animated point is covered by its neighbor if it is located within its $(d/2 - \varepsilon)$-sphere during the whole time interval.

To compute this extended coverage information, we consider the distance between two animated sample points. We will show that two points cover each other if (and obviously only if) they cover each other at the start and at the end of the time interval. To do so, we first consider a single sample point on a triangle and show that it will move on a linear path: Let again the triangle t consist of vertices v_i ($i = 1, 2, 3$) with coordinates $v_i(t) = v_i(0) + t \cdot (v_i(1) - v_i(0))$. The sample point $p(t)$ can be expressed as linear combination of the vertices with constant weights λ_i:

$$
\begin{aligned}
p(t) &= \sum_{i=1}^{3} \lambda_i v_i(t) \\
&= \sum_{i=1}^{3} \lambda_i \left(v_i(0) + t\left(v_i(1) - v_i(0)\right)\right) \\
&= \underbrace{\sum_{i=1}^{3} \lambda_i \left(v_i(0)\right)}_{\substack{\text{constant vector} \\ =: p_0}} + t \underbrace{\sum_{i=1}^{3} \lambda_i \left(v_i(1) - v_i(0)\right)}_{\substack{\text{constant vector} \\ =: \Delta p}} \\
&= p_0 + t\Delta p
\end{aligned}
$$

Hence, $p(t)$ is a linear function with base and direction vectors being weighted averages of those of the vertex trajectories. Next, we consider the squared distance $d^2(t)$ of two points $p(t)$, $q(t)$:

$$
\begin{aligned}
d^2(t) &= [p(t) - q(t)]^2 \\
&= [p_0 + t\Delta p - q_0 - t\Delta q]^2 \\
&= [\underbrace{(p_0 - q_0)}_{=: d_0} + t\underbrace{(\Delta p - \Delta q)}_{=: \Delta d}]^2 \\
&= [d_0 - t\Delta d]^2 \\
&= \Delta d^2 t^2 + 2 d_0 \Delta d t + {d_0}^2
\end{aligned}
$$

The result is a quadric function in t, with positive coefficient for the t^2 term. Thus, the distance is a convex parabola[21]. Thus, the distance within the time interval cannot exceed the value at the endpoints. Hence, it is sufficient to compute the distances at start and end time to determine whether animated points cover one another throughout the time interval.

Using this technique, we can employ the stratification algorithm from Section 4.2.3.4 and just modify the computation of the coverage information. This yields a stratified sample set with guaranteed sample spacing.

4.3.3.3 Time Subintervals

The oversampling of the animated version of the sampling algorithm can be larger than that of the static version. It is possible for the triangles to change their area strongly over time. In a bad input scene, the area can grow by $O(t^2)$ (square root of a 4th degree polynomial), e.g. by scaling the object linearly over time, thus increasing the area quadratically. This means that the average area

$$
\overline{A} = \int_0^1 A(t)dt = \int_0^1 ct^2 dt = \left[\frac{1}{3}ct^3\right]_{t=0}^{t=1} = \frac{c}{3}
$$

is three times smaller than the maximum area c. In that case, we obtain an average oversampling factor of 3. To reduce this effect, we can employ *time subintervals*: We divide the time between two keyframes into k time subintervals and apply the stratification algorithm to each subinterval separately, using the same candidate set. For each time interval, unnecessary points can be marked and excluded from rendering. For efficient rendering, we maintain a list of point indices for each subinterval. This increases the memory demands. However, for small k, the additional storage costs might be acceptable, especially if expensive attribute sets are used for the points. For small k and a large number of points, it is also possible to store 2^k-1 different lists for points

[21] In case of $\Delta d = 0$, the distance is constant so that our claim is trivially true.

that are present in each possible combination of time subintervals. During rendering, only those groups of triangles are rendered that are visible at the current time. For small k, this is the most efficient technique both in terms of rendering time ($O(n+2^k)$ for n visible points) and storage costs ($O(n')$ for n' overall sample points). For storage (e.g. on harddisc, not during rendering), a simple k-bit bitfield per point can also be employed as compact representation.

4.3.4 Instantiation

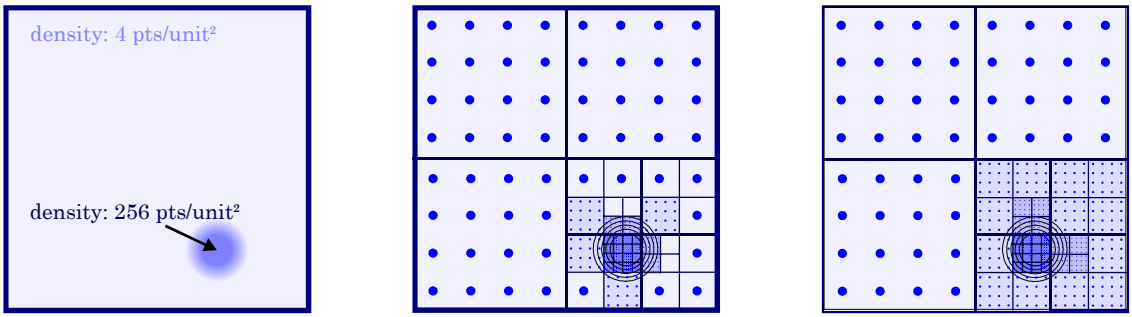
To build large scenes, we can again use instantiation, as before. In addition to a local coordinate system, we can also assign each instance a local time. In addition, all instantiation operations (affine transformations, change of materials, deformation etc…) can be made time-dependent.

For hierarchical instantiation, there are some issues that have to be taken care of: We must precompute the higher level hierarchy. Thus, the global animation has a fixed complexity, as every possible global motion must be known in advance and split into keyframes in order to compute the higher levels in the hierarchy. If we control the instantiation manually for each instance, we can handle all instantiation parameters dynamically without being forced to precompute. This leads to more freedom for the design of the global behavior of the animation. However, in this case, the complexity of the scene is limited. Every entity that is controlled manually can be a complex model (also using hierarchical instantiation, without dynamic control of the instantiation parameters). However, the total number of entities being controlled is limited by the at least linear run time of the global control algorithm.

4.4 Comparing the Data Structures

In this chapter, we have proposed three different data structures for a point-based multiresolution representation of three-dimensional surface models: *Dynamic sampling* and *static sampling* with a *nested* and a *full sampling* strategy. In this concluding subsection, we are going to examine the differences of these proposals and their consequences.

An important aspect is the type of sample points that is obtained from the different data structures: Dynamic sampling and nested sampling data structures always use point samples in a strict sense: Every sample point represents the properties of an infinitesimally small surface point. Full sampling data structures can also provide prefiltered sample points (beside strict point samples) that contain information about the neighborhood of the sample point, such as prefiltered colors, estimates for the transparency, surface curvature or the distribution of normal vectors. Both approaches are useful in certain applications: Prefiltering eliminates noise and aliasing artifacts without requiring many sample points per pixel. Thus, the strategy provides the best ratio of image quality and rendering times. However, the application of prefiltering techniques is also restricted: For general illumination models, we often have to deal with large sets of varying surface attributes such as multiple texture layers or procedural shaders. In general, it is not clear how general attribute sets can be processed during prefiltering. Solutions are only known for special cases (normals, color). Often, these are only approximations: For example, it does not seem to be possible to represent the normal distribution or the transparency under different viewing angles faithfully using a fixed amount of information (Section 4.2.4). Thus, the usage of strict point samples may be an option if real-time performance is not the primary goal but flexibility in shading is mandatory.



(a) specified sampling density (strongly varying)

(b) piecewise constant approximation using dynamic sampling (after stratification)

(c) piecewise constant approximation using precomputed sample sets

Figure 41: Comparison of the approximation accuracy of dynamic and static sampling. (Note that this is a schematic figure, the sample points are only depicted to visualize the sampling density. Usually, they are not stratified on a regular grid, at least not in case b.)

A second, related aspect is the flexibility in modeling: Point-based rendering is useful for highly detailed scenes. Usually, a scene in which point-based rendering leads to a significant speed up does not fit into main memory as a plain set of triangles with simple, linear, uncompressed encoding. In this thesis, we consider hierarchical instantiation techniques to encode complex scenes. The degree of freedom for an instantiation operation also depends on the data structure: All data structures allow geometric transformations such as affine transformations. In addition, we could also handle some more general transformations that do not lead to a non-constant scaling of surface area or destroy the locality of the spatial hierarchy. Changing the surface properties is only possible with simple point samples that consist only of surface position, orientation, and a material identifier at the sample point. Here we could think of exchanging material identifiers by id-numbers or applying procedural shader scripts to the surface. For prefiltered samples, discrete attributes on the surface such as material ids are lost and thus cannot be used for instantiation.

Besides flexible instantiation, we could also think of extending point-based rendering to more general procedural modeling techniques. We expect that randomized dynamic sampling could be generalized more easily than static sampling techniques as these require the precomputation of all sample points in advance. This probably abandons the advantages in terms of memory efficiency of procedural modeling techniques.

In terms of performance, full sampling provides the least run-time costs as this technique allows the usage of all stratification techniques and only uses precomputed sample points. The drawback is memory usage: We cannot strictly prove linear memory costs. However, this is of little relevance in practice (as discussed before). Nested sampling is second best in terms of rendering costs. Prefiltering cannot be used. This usually increases the necessary sample size for rendering low-noise images significantly. In addition, only some stratification techniques can be employed. However, in terms of memory usage, the approach yields an optimal, linear behavior. The least efficient technique is dynamic sampling. It computes an unstratified sample set of simple sample points dynamically, during rendering. Thus, the size of a sample set that is sufficient for high image quality is fairly high. However, this technique offers the most flexibility for modeling. It also needs linear storage only. The construction time of $O(n \log n)$ is optimal, better than $O(h\,n)$ for static sampling (h being the height of the octree).

A subtle advantage of the dynamic sampling strategy is sampling accuracy (Figure 41): The technique can always ensure a strict ε-approximation of a given sampling density function: Each

bounding box is subdivided until the density function does not vary by more than a constant within the bounding box. This is not the case for static sampling: If a box is subdivided, the sampling density of all sibling boxes is increased, too, possibly without need. This leads to increased oversampling. However, to our experience, the general performance advantages of static sampling dominate such effects in most applications.

Chapter 5
Forward Mapping

In this chapter, we describe an efficient forward mapping rendering algorithm that makes use of the data structures of the preceding chapter. The algorithm provides interactive to real-time frame rates for highly detailed scenes. It relies on z-buffering, similarly to real-time rendering techniques for conventional, triangle-based models. The rendering algorithm consists of two steps: First, we perform a hierarchy traversal to collect a set of sample points for all objects in the view frustum. Second, we reconstruct an image from the sample points. This process again consists of two logical steps: First, all invisible sample points have to be removed from the sample set. Second, the remaining points are fed into a scattered data interpolation algorithm that reconstructs a continuous image. The chapter consists of three sections: First, we examine properties of the perspective projection. Then we use these observations to devise an efficient rendering algorithm and analyze its cost. The third section describes several techniques for image reconstruction.

5.1 Perspective Projection

We use the point-based multi-resolution data structure to implement an importance sampling approach: Our goal is to cover the on-screen projections of the objects of the scene uniformly with sample points. This allows a reconstruction of an image with uniform resolution all over the screen. Put in other words, we need a sampling density for each surface fragment that is proportional to the projection factor under a given (dynamically changing) perspective projection. The projection factor is defined as the scaling factor that is applied to an infinitesimally small surface fragment when it is projected onto the screen. To derive the projection factor, we consider an infinitesimally small surface fragment f. First, we assume that the fragment f is parallel to the image plane. With g denoting the projected size of f, z being the orthogonal distance from f to the center of projection and d being the distance of the image plane (i.e. a constant that describes the scaling factor of the rendered image), we obtain (Figure 42):

$$\frac{f}{g} = \frac{h_2}{h_1} = \frac{z}{d}, \text{ and thus } \quad g = f\frac{d}{z}$$

For two-dimensional surface fragments, the scaling factor has to be squared so that we obtain

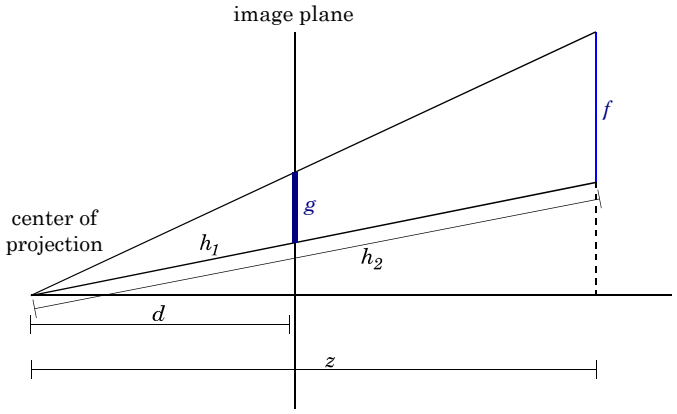


Figure 42: Projection of surface fragments parallel to the image plane.

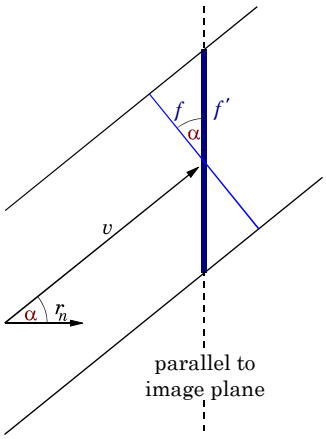


Figure 43: Projection of surface fragments orthogonal to the vector to the center of projection.

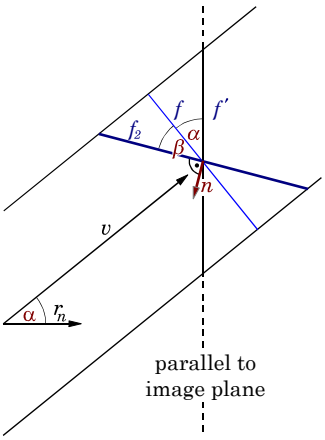


Figure 44: The general case.

$$A_g = A_f \cdot \frac{d^2}{z^2}$$

Next, we derive the projection factor for surface fragments that are orthogonal to the vector v that connects the center of f and the center of projection. As f is assumed to be infinitesimally small, the choice of a "center" point for f has no influence on the result. We can assume that the center of projection is infinitely far away. To compute the projected area, we rotate f so that it is parallel to the image plane, yielding a fragment f'. With α denoting the angle between view direction r_n and vector v, we obtain (Figure 43):

$$f' = \frac{f}{\cos\alpha}, \quad \text{and thus} \quad A_g = A_f \cdot d^2 \cdot \frac{1}{\cos\alpha} \cdot \frac{1}{z^2}$$

Lastly, we allow an arbitrary orientation of the surface fragment (fragment g_2 in Figure 44). We first rotate this fragment until it is perpendicular to the view vector v, the we apply the preceding formula, yielding

$$A_g = A_f \cdot d^2 \cdot \cos\beta \frac{1}{\cos\alpha} \cdot \frac{1}{z^2}$$

with β denoting the angle between surface normal n and v. If we assume backface culling (i.e. invisible back sides), we also have to clamp the cosine of β to the range [0, 1]. Overall we obtain the following formula for the projection factor that scales the area of a surface fragment f:

$$prj(f) = \frac{d^2 \left|\cos\beta\right|_{[0..1]}}{z^2 \cos\alpha} \tag{11}$$

In vector notation, with v denoting the difference vector between the center of projection and (the center of) f, v_n the normalized version of vector v, and r_n the normalized view direction (i.e. normal of the image plane), we obtain:

$$prj(f) = d^2 \frac{\left|\langle v_n, n\rangle\right|_{[0..1]}}{\langle v_n, r_n\rangle \langle v, r_n\rangle^2} \tag{12}$$

The projection factor consists of three distinct terms: the *depth factor* z^{-2}, the *orientation factor* $\cos\beta$ and the *distortion factor* $\cos\alpha$. An additional constant factor d^2 describes the scaling to pixel coordinates. The depth factor accounts for perspective foreshortening with increasing distance of the object. The distance is measured orthogonal to the image plane (i.e. as scalar product with the normalized view direction vector r_n) because the image is created as projection on a plane. The orientation factor accounts for the fact that the projected area of an object is reduced if it is seen under a small angle. The distortion factor accounts for a slight increase in projected area if an object is projected onto an outer area of the screen instead of its center.

Our goal is to create sample points that are distributed uniformly on the projections of the objects in the image plane. Thus, an optimal sampling density function would be proportional to the projection factor within the view frustum and zero outside. However, this ideal sampling density cannot be used efficiently in practice. Even a strict ε-approximation of the sampling density usually voids output-sensitive rendering times, as discussed in the next section more in detail. Consequently, we use a simplified projection factor to determine the sampling density. The most dominant factor is the depth factor. For scenes with larger extends and close viewpoints, it varies strongly and usually should not be neglected. The orientation factor is only a factor between zero and one. However, if it is close to zero, neglecting the orientation factor can still lead to strong overestimations of the projection factor. The third factor, the distortion factor is always in the interval $[\cos^{-1}\alpha_{max}, 1]$ with α_{max} being the maximum angle between an object on the screen and the central viewing direction. This factor is usually quite small and can only lead to a constant overestimation if being neglected. For example, a diagonal view angle of 90° leads to a maximum deviation of 45° and thus to $\cos^{-1}\alpha_{max}$ of 1.41. In the case of 45° diagonal view angle, we obtain 1.08.

It is also interesting to see how the different factors can be bound: The depth factor and the distortion factor can be bound by restricting the location of potentially projected objects: The depth factor requires a restriction of the depth interval and the distortion factor a restriction of the angle to the central view direction. In the following, we are going to use octree subdivision for spatial localization. Thus, both factors are bound at the same time. In contrast, the orientation factor is independent of the spatial location. It depends only on the orientation of the surface. This means that any spatial group of triangles can potentially contain a large number of different orientations that require a separate classification, orthogonal to spatial classification. In addition, an oriented surface also demands for anisotropic sampling patterns for an optimally uniform distribution of the sample points in the image plane. To avoid this additional effort, we usually neglect the orientation factor and consider only the depth and distortion factor during rendering.

```
Algorithm dynamicPerspectiveSampling(Node v, Camera c)
 result := ∅
 If B(v) ∩ viewFrustum(c) ≠ ∅ Then
   If (v is a leave node) Then
     result := all triangles in v
   Else
     If (depth factor in B(v) varies by at most 1 + ε) Then
       A   := areaOfTriangles(v)
       df  := maxDepthFactor(v,c)
       of  := maxOrientationFactor(v,c)
       xf  := maxDistortionFactor(v,c)
       prj := df·of·xf·oversampling·screenScaling
       n   := A·prj
       If (averageTriangleArea·prj > p_max) Then
         result := all triangles in subtree(v)
       Else
         result := n random sample points from v
       End If
     Else
       For each (child k of v) Do
         result := result ∪ dynamicPerspectiveSampling(k)
       End For
     End If
   End If
 End If
 Return result
```

Algorithm 2: Conservative perspective sampling using a dynamic sampling data structure.

5.2 Hierarchy Traversal

5.2.1 The Rendering Algorithm for Dynamic Sampling

To render the scene, we first fix an on-screen sampling spacing d', which is a user defined constant. Then we define an object space sampling density that guarantees the specified on-screen sample spacing. We employ an object space sampling density proportional to the depth factor of the projection factor with a constant that leads to a sample spacing of d' pixels on the screen. Then we apply an algorithm similar to the ε-approximation algorithm (Algorithm 1, Section 4.1.3) to this sampling density function. In order to consider the other two factors (orientation, distortion) as well as the view frustum as well as in order to perform the detection of large triangles, we need to modify the basic algorithm slightly:

We start traversing our data structures by examining the preclassification lists (Section 4.1.4 and 4.1.5) with orientation and area classes, if applicable: If preclassification is used, we have to deal with separate spatial hierarchies that only contain triangles of similar area (area classes) or triangles with similar orientation (orientation classes) or a combination of both. The

```
Algorithm staticPerspectiveSampling(Node v, Camera c):
 result := ∅
 If B(v) ∩ viewFrustum(c) ≠ ∅ Then
   df  := maxDepthFactor(v,c)
   of  := maxOrientationFactor(v,c)
   xf  := maxDistortionFactor(v,c)
   prj := df·of·xf·oversampling·screenScaling
   If (samplingDensity(v) ≥ prj) Then
     Return (triangles and sample points in v)
   Else
     For each (child k of v) Do
       result := result ∪ staticPerspectiveSampling(k)
     End For
     result := result ∪ triangles in v
     If (nestedSampling) Then
       result := result ∪ sample points in v
     End If
   End If
 End If
 Return result
```

Algorithm 3: Conservative perspective sampling using a static sampling data structure. In case of stratified sampling, *maxOrientationFactor* has always to be set to 1.

latter case means that the list of area classes again contains lists of orientation classes[22]. For each spatial hierarchy found in this lists, we store the maximum and minimum area values as well as the average normal direction along with the maximum angle by which a normal from that class can deviate from the average. These values are used later on to refine the sampling decisions.

Then we start with the traversal of the spatial hierarchy (see Algorithm 2 for a pseudocode description): For each node, we first check whether it is contained in the view frustum. If this is not the case, the recursion is stopped. Otherwise, we check whether the spatial extends of the current octree node (i.e. its bounding box) already allow an ε-approximation of the depth factor. If this is not the case, the algorithm is applied recursively to the child nodes. If the approximation is already good enough, we compute an *upper* bound for the projection factor and use this bound to determine the number of sample points necessary for conservative sampling. If the average number of sample points per triangle exceeds the predefined constant p_{max}, we do not perform sampling but report all triangles of the node instead. The average area per triangle used in this computation is obtained from the area classes[23]. Otherwise, the sample points are generated as described in Section 4.1.2. To determine the number of sample points, the stored normal cone of the current orientation class is considered. In conjunction with a bounding sphere of the current octree node, we can determine the minimum angle β between a surface normal and the view direction and reduce the sampling density accordingly.

[22] Due to the strong increase of classification efforts, this strategy is not used in practice.
[23] It seems that we could also use *A/#triangles* as average triangle area, not needing area classes. It is possible to use the value as alternative method for computing the number of sample points per triangle. However, we would still need a classification by triangle area. Otherwise, the size of the triangles could vary strongly so that we would also report large numbers of small triangles once a few sufficiently large triangles are contained in the same node. This is of course less efficient than treating triangles of similar size only.

5.2.2 The Rendering Algorithm for Static Sampling

The rendering algorithm for static sampling data structures (Algorithm 3) is very similar to that for dynamic sampling. The algorithm performs a hierarchy traversal until the sampling density reaches or exceeds the specified on-screen sampling density. It also stops if a node is outside the view frustum. When the desired sampling density is found, the precomputed sample points are reported. All triangles that are found on the way down the hierarchy are reported, too. In case of nested sampling, we also have to report all sample points in the inner nodes of the traversal tree. The algorithm computes an estimate for all three factors in the projection factor. In case of the orientation factor, we need again a preclassification by orientation to estimate the factor. In addition, it can only be used for random sampling. If stratification is used, we cannot decrease the sample pattern any longer for triangles seen under a small angle. In that case, the on-screen sampling pattern would become anisotropic and would not cover the area safely.

5.2.3 Analysis

The two algorithms given above obviously return a conservative sample set for the scene under a given perspective projection. The question is now: How efficient are these sampling approaches? Two different aspects determine the overall efficiency: First, we need to find out how many nodes of the hierarchy are traversed by the algorithm. Second, we need to determine the approximation accuracy. The better the conservative sample sets match the target sampling density necessary for image reconstruction, the shorter will be the rendering times. Obviously, these two questions are intervened. A better approximation accuracy will probably also cause a larger traversal effort, leading to a trade-off between traversal and oversampling overhead.

In this section, we will perform a formal analysis to determine the efficiency of our proposal. We will start with an analysis of the traversal costs for an ε-approximation of the depth factor using dynamic sampling (Algorithm 2). It will turn out that the costs depend only weakly on the scene so that even large scenes can be handled efficiently. Second, we will show that a similar result also holds for static sampling. The next topic to consider is view frustum culling. Here we will show that our algorithm performs an approximate culling with sufficient accuracy for average cases. As we have already bounded the influence of the distortion factor a priori, it then remains two consider the influence of the orientation factor. It will turn out that our techniques do not allow a strict ε-approximation of the sampling density for arbitrary scenes. However, we will show that we obtain only a constant oversampling on for “average” scenes so that this is no restriction in practice.

5.2.3.1 Traversal Costs for an ε-Approximation of the Depth Factor with Dynamic Sampling

Algorithm 2 determines a set of octree boxes from our spatial hierarchy so that the projection factor varies only by a factor of ε in every box. In addition, all boxes in the set intersect with the viewing frustum; boxes outside are skipped. How expensive is this traversal? This question is essential for the performance of the whole rendering approach.

To answer this question, we first need some notation: For given camera settings and an observer position p, let $z_{near}(p) > 0$ denote the minimum and $z_{far}(p) > z_{near}(p)$ denote the maximum depth of an object in the view frustum. The depth of the near clipping plane is always a lower bound for $z_{near}(p)$. We use z_{near} (without dependency of the observer position p) to denote the depth of the near clipping plane. Let $\tau(p) := z_{far}(p) / z_{near}(p)$ denote the *current relative depth range* of the

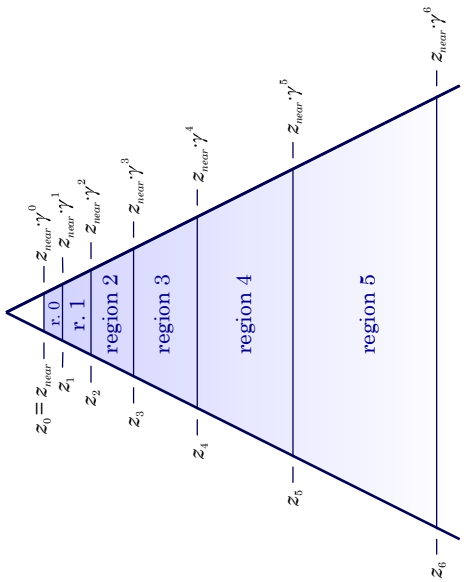


Figure 45: Dividing the view frustum into depth regions.

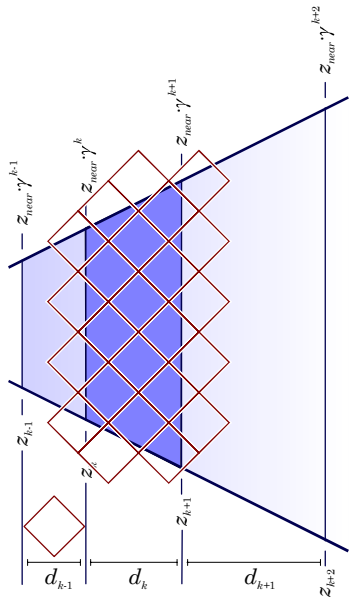


Figure 46: Covering one region with boxes with a diameter not larger than that of the previous region.

scene for the observer position. In general, $\tau(p)$ cannot be larger than the diameter of the scene (e.g. measured as the diagonal of the smallest axis aligned bounding box of the scene) divided by the z-value of the near clipping plane. We use τ to denote this maximum value, which we refer to as *maximum relative depth range*.

Now we will show that Algorithm 2 will chose O(log τ) octree boxes using O(h + log τ) time (h denoting the height of the octree) if we neglect the orientation factor and the distortion factor (setting them to one) and do not account for the sampling costs (which depend on the surface area and thus are potentially unbound). The proof consists of several steps: First, we divide the view frustum into regions of similar depth factor. Then, we show that each region can be covered with a constant number of octree boxes and that the number of regions is in O(log τ). Afterwards, we show that all regions can be covered with O(log τ) boxes from a single octree by considering transition effects at the borders of the regions. Lastly, we will bound the traversal costs to proof O(h + log τ) running time. The general approach of the proof is similar to that of [Chamberlain et al. 96]. In that paper, the authors show an O(log n) running time for their approximate rendering strategy (which is also based on a spatial hierarchy) under the assumption that n objects are uniformly distributed in the scene.

We start the proof by dividing the scene into regions of similar depth factor. To do so, we need a condition to bound the deviation of the depth factor. Our goal is to retain an ε-approximation of the depth factor, i.e. the depth factor may not vary by more than $(1 + \varepsilon)$ within a group of objects. With z_{min} denoting the minimum and z_{max} the maximum depth of a surface point within a group of objects, we obtain the condition

$$z_{max}^{\ 2} / z_{min}^{\ 2} \le 1+\varepsilon, \quad \text{i.e.} \quad z_{max} / z_{min} \le \sqrt{1+\varepsilon} \quad \text{or} \quad z_{max} \le \underbrace{\sqrt{1+\varepsilon}}_{=:\gamma} z_{min}.$$

In the following, we write γ for $\sqrt{1+\varepsilon}$ to simplify the notation. We start the division of the view frustum into regions of similar depth factor at the near clipping plane $z_0 := z_{near}$. The first region (region 0) ends at $z_1 := z_{near} \cdot \gamma$. The second (region 1) starts at that depth, reaching up to $z_2 :=$

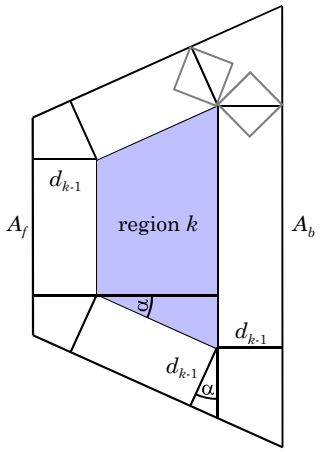


Figure 47: Depth region k, extended by the diameter d_{k-1} of a covering box.

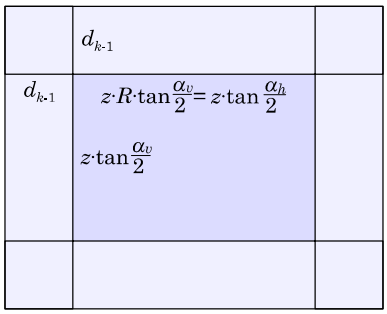


Figure 48: Cross section of the (extended) view frustum at depth z.

$z_{near} \cdot \gamma^2$. The k-th region r_k starts at $z_{k-1} := z_{near} \cdot \gamma^k$ and ends at $z_k := z_{near} \cdot \gamma^{k+1}$ (see Figure 46). This procedure defines $k_{max} := \lceil \log_\gamma \tau \rceil \in O(\log \tau)$ different regions. Within each region, the depth factor obviously varies by at most $(1 + \varepsilon)$.

Next, we show that each of these regions can be covered by a constant number of octree boxes without violating the conditions for the ε-approximation. The octree consists of a hierarchy of non-overlapping cubes $C(v)$. Each cube is associated with a bounding box $B(v)$ for the geometry stored in the corresponding subtree. The bounding box $B(v)$ of the node v might be larger than the cube $C(v)$ by a constant factor of $(1+\delta)$. For simplicity, we first only consider the non-overlapping cubes and account for the overlapping bounding boxes B later. This means that (for now) the octree nodes correspond to a hierarchy of three-dimensional, regular cube grids with side lengths shrinking by a factor of two at each hierarchy level. Within each level, the different cubes of the grid are disjoint. To cover a region r_k safely, without violating the conditions for the ε-approximation, we use cubes with a diagonal smaller than the depth interval $d_{k-1} := z_k - z_{k-1} = z_{near} \cdot (\gamma - 1) \cdot \gamma^{k-1}$ of the previous region (Figure 46). If we use cubes with a diagonal not larger than d_{k-1}, we can safely cover region k: Each cube that intersects with region k cannot intersect with region $k - 2$ or lower. The largest cube diagonal found in the hierarchy that fulfills these requirements is always in the interval $[d_{k-1}/2, d_{k-1}]$ because the cubes are available with side length varying by a factor of two.

The idea for counting the number of such cubes is to compare the volume of the region and the minimum volume of a cube. As we also have to deal with cubes that intersect only partially with the depth region, we cannot directly argue based on the volume of the region but we use an extended region as conservative estimate:

Figure 47 shows depth region k, extended by d_{k-1} in all orthogonal directions. Any set of boxes with diameter d_{k-1} that intersects with region k must lie completely within the extended region. If a part of the box was located outside the extended region, it would not be able to reach the inner region itself and thus would be superfluous. Hence, the number of disjoint boxes covering the inner region is bounded by the volume of the extended region divided by the minimum volume of the covering cubes, which is $V_{cube(k-1)} := (d_{k-1}/(2\sqrt{3}))^3 = \sqrt{3}\, d_{k-1}^3 / 72$. To determine the volume of the extended region, we first compute the cross-sectional area of the extended view frustum at a given depth z: The cross-sectional area of the view frustum at depth z is given by

$$height(z) \cdot width(z) = \left(z \cdot \tan\frac{\alpha_v}{2} \right) \cdot \left(z \cdot \tan\frac{\alpha_h}{2} \right) = z^2 R \tan^2\frac{\alpha_v}{2}$$

with α_v being the vertical and α_h the horizontal view angle of the view frustum and R being the aspect ration, i.e. the ratio between height and width of the view frustum. The cross-sectional area of the extended view frustum is given by (see Figure 48):

$$\begin{aligned} A_{ex}(z) &= 4{d_{k-1}}^2 + 2d_{k-1} \cdot height(z) + 2d_{k-1} \cdot width(z) + height(z) \cdot width(z) \\ &= 4{d_{k-1}}^2 + 2d_{k-1}\left(z\tan\frac{\alpha_v}{2} \right) + 2d_{k-1}\left(zR\tan\frac{\alpha_v}{2} \right) + z^2 R\tan^2\frac{\alpha_v}{2} \\ &= z^2 \underbrace{R\tan^2\frac{\alpha_v}{2}}_{A} + zd_{k-1}\underbrace{\left(2\tan\frac{\alpha_v}{2}(1+R) \right)}_{B} + 4{d_{k-1}}^2 \end{aligned}$$

We substitute symbolic names A, B for the constant terms in this formula so that the cross-sectional area can be expressed as:

$$A_{ex}(z) = A z^2 + B z d_{k-1} + 4{d_{k-1}}^2$$

Substituting $z_{near} \cdot (\gamma - 1) \cdot \gamma^{k-1}$ for d_{k-1}, this can be expressed as:

$$A_{ex}(z) = A \cdot z^2 + B \cdot z \cdot z_{near}(\gamma - 1)\gamma^{k-1} + 4{z_{near}}^2(\gamma-1)^2\gamma^{2k-2}$$

The volume of a frustum with start area of A_f, end area A_b and depth d is $1/2(A_f + A_b)d$. Thus, we obtain a volume of the extended region of:

$$\begin{aligned} V_k &= \frac{A_f + A_b}{2} d \\ &= \frac{A_{ex}(z_k - d_{k-1}) + A_{ex}(z_{k+1} + d_{k-1})}{2} \cdot \left(d_k + 2d_{k-1} \right) \end{aligned}$$

Now we substitute $z_{near} \cdot (\gamma - 1) \cdot \gamma^{k-1}$ for d_{k-1} and $z_{near} \cdot \gamma^k$ for z_k and obtain:

$$\begin{aligned} V_k &= \frac{1}{2}\left(A_{ex}(z_{near}\gamma^k - z_{near}\gamma^{k-1}(\gamma-1)) + A_{ex}(z_{near}\gamma^{k+1} + z_{near}\gamma^{k-1}(\gamma-1)) \right) \cdot \left(z_{near}\gamma^k(\gamma-1) + 2z_{near}\gamma^{k-1}(\gamma-1) \right) \\ &= \frac{1}{2}\left(A_{ex}(z_{near}\gamma^{k-1}) + A_{ex}(z_{near}\gamma^{k-1}(\gamma^2+\gamma-1)) \right) \cdot \left(z_{near}\gamma^{k-1}(\gamma^2+\gamma-2) \right) \end{aligned}$$

and applying the formula for A_{ex} yields:

$$\begin{aligned} V_k &= \frac{1}{2}\begin{pmatrix} A \cdot (z_{near}\gamma^{k-1})^2 + B \cdot (z_{near}\gamma^{k-1}) \cdot z_{near}(\gamma-1)\gamma^{k-1} + 4{z_{near}}^2(\gamma-1)^2\gamma^{2k-2} \\ + A \cdot (z_{near}\gamma^{k-1}(\gamma^2+\gamma-1))^2 + B \cdot (z_{near}\gamma^{k-1}(\gamma^2+\gamma-1)) \cdot z_{near}(\gamma-1)\gamma^{k-1} \\ + 4{z_{near}}^2(\gamma-1)^2\gamma^{2k-2} \end{pmatrix} \cdot (z_{near}\gamma^{k-1}(\gamma^2+\gamma-2)) \\ &= \frac{1}{2}\begin{pmatrix} A \cdot {z_{near}}^2\gamma^{2k-2} + B \cdot {z_{near}}^2\gamma^{2k-2}(\gamma-1) + 8{z_{near}}^2(\gamma-1)^2\gamma^{2k-2} \\ + A \cdot {z_{near}}^2\gamma^{2k-2}(\gamma^2+\gamma-1)^2 + B \cdot ({z_{near}}^2\gamma^{2k-2}(\gamma^2+\gamma-1) \cdot (\gamma-1) \end{pmatrix} \cdot (z_{near}\gamma^{k-1}(\gamma^2+\gamma-2)) \\ &= \frac{1}{2}{z_{near}}^3\begin{pmatrix} A\gamma^{2k-2} + B \cdot \gamma^{2k-2}(\gamma-1) + 8(\gamma-1)^2\gamma^{2k-2} \\ + A\gamma^{2k-2}(\gamma^2+\gamma-1)^2 + B\gamma^{2k-2}(\gamma^2+\gamma-1) \cdot (\gamma-1) \end{pmatrix} \cdot (\gamma^{k-1}(\gamma^2+\gamma-2)) \\ &= \frac{1}{2}{z_{near}}^3\gamma^{3k-3}\left(A + B \cdot (\gamma-1) + 8(\gamma-1)^2 + A(\gamma^2+\gamma-1)^2 + B(\gamma^2+\gamma-1) \cdot (\gamma-1) \right) \cdot \left(\gamma^2+\gamma-2 \right) \end{aligned}$$

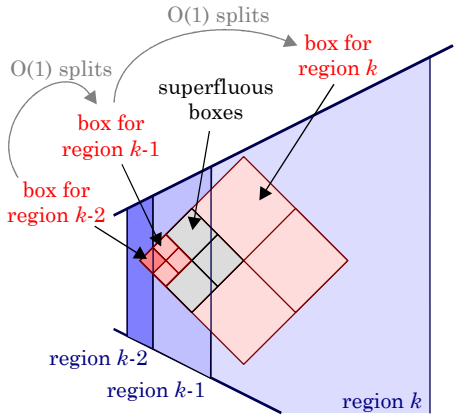


Figure 49: Bounding the number of "superfluous" splits for boxes taken from an octree grid.

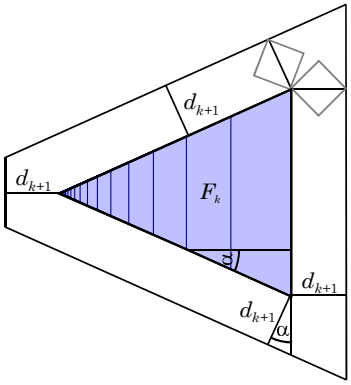


Figure 50: Cumulative depth regions $F_k := \bigcup_{i=1..k}$ (region k), extended by d_{k+1}.

Now we determine the ratio between V_k and $V_{cube(k\text{-}1)} = \sqrt{3}\, d_{k\text{-}1}{}^3 / 72$ to obtain an upper bound for the number of octree boxes necessary to cover region k. First, we can express $V_{cube(k\text{-}1)}$ as:

$$V_{cube(k-1)} = \frac{\sqrt{3}}{72}\left(z_{near}(\gamma-1)\gamma^{k-1}\right)^3 = \frac{\sqrt{3}}{72} z_{near}{}^3 \gamma^{3k-3}(\gamma-1)^3$$

Then, the ratio can be computed as:

$$\frac{V_k}{V_{cube(k-1)}} = \frac{z_{near}{}^3 \gamma^{3k-3}\left(A + B\cdot(\gamma-1) + 8(\gamma-1)^2 + A(\gamma^2+\gamma-1)^2 + B(\gamma^2+\gamma-1)\cdot(\gamma-1)\right)\cdot\left(\gamma^2+\gamma-2\right)}{\frac{2\sqrt{3}}{72} z_{near}{}^3 \gamma^{3k-3}(\gamma-1)^3}$$

$$= \frac{72}{2\sqrt{3}} \frac{\left(A + B\cdot(\gamma-1) + 8(\gamma-1)^2 + A(\gamma^2+\gamma-1)^2 + B(\gamma^2+\gamma-1)\cdot(\gamma-1)\right)\cdot\left(\gamma^2+\gamma-2\right)}{(\gamma-1)^3}$$

$$\in \Omega\left(\frac{\gamma^8}{\gamma^3}\right) = \Omega(\gamma^5) = \Omega\left((1+\varepsilon)^{\frac{5}{2}}\right)$$

Overall, the ratio between the volume of the extended region k and the volume of an octree box of maximum size that allow a save coverage of region k is a constant that only depends on the approximation accuracy ε and the parameters of the perspective projection. Thus, every region can be covered with a constant number of boxes. Please note that we are considering an upper bound. The result above does *not* mean that the traversal effort will increase with increasing ε (indeed, they are decreasing in practice). Only the conservative upper bound increases. The reason for this is that we using boxes fitting into region k-1 to cover region k. As the ratio between these two regions increases with γ, we also obtain an upper bound that increases with γ.

Our next task is to show that not only a single region can be covered with a constant number of cubes from the octree but that this is also true for all regions if they have to be covered simultaneously with a selection of cubes from the tree. If we want to cover a region with boxes of uniform size and adjacent regions with boxes of smaller size, it can happen that the large boxes used for large regions have to be subdivided in order to obtain the smaller boxes that are needed for smaller regions. This leads to more boxes of intermediate size that also have to be counted (Figure 49). We consider a box for a region k that has to be subdivided because it also overlaps with smaller regions k-1, …, j ($j < k$). Note that it is impossible to overlap a region k and j and not the regions k-1, …, j+1 in between. To bound the number of additional nodes that are output by

the algorithm, we count the number of additional nodes caused by cubes overlapping with region k-1, which causes boxes of region k to be split. This also counts further splits: the boxes in region k-2 account for splits of boxes from k-1 and so on. Thus, it is sufficient to consider only the costs caused by one box from region k-1 that leads to splitting boxes from region k. First, we already know that only a constant number of boxes are needed to cover region k-1. Additionally, the costs for each box are bounded, too: As boxes needed to cover region k are only larger by a factor of γ than boxes covering region k-1, we need at most $\lceil \log_2 \gamma \rceil \in \mathrm{O}(1)$ splits within the corresponding node from region k to shrink the node accordingly, leading to only O(1) additional boxes output by the algorithm.

Up to now, we have only considered the cubes of the octree for covering the regions. Indeed, we are forced to use the (extended) bounding boxes. These boxes might have side lengths that are larger by a factor of $(1+\delta)$ (for a user defined constant $\delta > 0$), i.e. their volume can be larger by a factor of at most $(1+\delta)^3$. This increases the number of boxes needed for covering the volume because a deeper subdivision level is necessary for larger boxes. However, as the volume is increased by a constant factor only, the number of boxes is also increased by a constant factor only. In the worst case, $\lceil \log_2 (1+\delta) \rceil$ additional splits are needed for every box.

Overall, this shows that the algorithm outputs at most O(log τ) octree nodes: We have constructed a set of nodes of size O(log τ) at which the traversal will stop in any case. If the algorithm terminates the recursion earlier for some subtrees, this does not increase the number of output nodes as the potential output is monotonically increasing with the recursion depth.

Now it remains to show, that the traversal costs are also small. We prove a bound of O(h + log τ) for an octree height of h. It is obvious that the traversal costs are O(log τ) if the bounding box of the scene lies completely within the view frustum. In this case, the all children of inner nodes in the traversal tree are forced to be traversed. The branching factor is at least two as they would be short-cut otherwise. Therefore, the number of nodes in the tree at increasing levels forms a geometric sum with an exponent of at least two, which means that the tree contains O(L) nodes if L is the number of leafs of the recursion tree, which is in O(log τ).

Problems can only occur if only a part of the tree is located inside the view frustum. Then it is possible that only one child node is visited during traversal for some of the nodes. To bound the number of processed nodes in this case, we consider cumulative regions F_k that are the union of regions 1…k. Then, we consider all nodes from the tree that have a diagonal in the interval $(d_k, d_{k+1}]$. These nodes are only subdivided if they intersect with F_k. Nodes that are smaller than d_0 are never subdivided due to the ε-approximation of the depth factor. Nodes with a diameter larger than $d_{k_{max}+1}$ are accounted for later; for now we only consider nodes with a diagonal below this size. Now we show that at most O(1) boxes exists that intersect with F_k and have a diagonal in the given interval. To do so, we again form an extended region by increasing F_k by d_{k+1} in all orthogonal directions. All nodes of the given size that intersect with F_k must lie completely within this region. Thus, the ratio of its volume and the minimum volume of an octree cube bounds the number of such nodes. Again, we consider only the disjoint cubes. Using overlapping cubes enlarged by at most $(1+\delta)$ will increase the number of boxes by at most a constant factor.

The volume of the extended regions is no larger than

$$V_{F_k} \leq \frac{A_{ex}'(z_{k+1} + d_{k+1}) + A_{ex}'(0)}{2}\left(z_{k+1} + 2d_{k+1}\right)$$

with A_{ex}' equal to

$$A_{ex}'(z) = z^2 A + z d_{k+1} B + 4 {d_{k+1}}^2$$

with constants A, B as defined above. This leads to a volume of:

$$\begin{aligned} V_{F_k} &\le \frac{1}{2}\left((z_{k+1}+d_{k+1})^2 A + (z_{k+1}+d_{k+1})d_{k+1}B + 8{d_{k+1}}^2\right)\cdot\left(z_{k+1}+2d_{k+1}\right) \\ &= \frac{1}{2}{z_{near}}^3\gamma^{3k+3}\left((1+(\gamma-1))^2 A + (1+(\gamma-1))(\gamma-1)B + 8(\gamma-1)^2\right)\cdot\left(1+2(\gamma-1)\right) \\ &= \frac{1}{2}{z_{near}}^3\gamma^{3k+3}\left(A\gamma^2 + B(\gamma^2-\gamma) + 8(\gamma-1)^2\right)\cdot\left(1+2(\gamma-1)\right) \end{aligned}$$

The minimum volume of a box with a diagonal in the interval $(d_k, \ldots, d_{k+1}]$ is

$$V_{cube(k)} \ge \frac{\sqrt{3}}{72}{z_{near}}^3\gamma^{3k}(\gamma-1)^3$$

Thus, the ratio is bounded by

$$\begin{aligned} \frac{V_{F_k}}{V_{cube(k)}} &\le \frac{\frac{1}{2}{z_{near}}^3\gamma^{3k+3}\left(A\gamma^2 + B(\gamma^2-\gamma) + 8(\gamma-1)^2\right)\cdot\left(1+2(\gamma-1)\right)}{\frac{\sqrt{3}}{72}{z_{near}}^3\gamma^{3k}(\gamma-1)^3} \\ &= \frac{36}{\sqrt{3}}\frac{\gamma^3\left(A\gamma^2 + B(\gamma^2-\gamma) + 8(\gamma-1)^2\right)\cdot\left(1+2(\gamma-1)\right)}{(\gamma-1)^3} \in \mathrm{O}(1), \end{aligned}$$

which is a constant value, independent of k and z_{near}. Up to now, we have shown, that only a constant number of octree boxes are visited during traversal with a diagonal in the interval $(d_k, d_{k+1}]$ for each k from 0 to k_{max}, with $k_{max} \in \mathrm{O}(\log \tau)$. Boxes with a diagonal smaller than d_0 are not used in the traversal. It remains to bound the number of larger boxes. A box with side length larger than d_{k+1} cannot be used to cover a view frustum region. Thus, it is subdivided because an indirect child node with side length $(d_{k_{max}-1}, \ldots, d_{k_{max}}]$ is used for covering $F_{k_{max}}$. As shown before, only O(1) such nodes can exist. Thus, they cannot have more than $\mathrm{O}(h)$ parent nodes. This leads to an overall bound for the runtime of $\mathrm{O}(\log \tau + h)$.

These costs are rather small. They depend only logarithmically on the maximum depth range and linearly on the height of the tree. In practice, we can expect that these two values are quite small. Note that the costs also depend on the approximation accuracy ε and the viewing parameters. Intuitively (omitting a formal proof), we could expect a growth of the number of selected octree boxes of $\Theta((\gamma-1)^{-3})$ for $\gamma \to 1$ because the initial box size in the first region shrinks with $(\gamma-1)^3$. Therefore, all boxes, which have a size that is a multiple of this size, also grow and shrink by this factor. Additionally, the cross section of the view frustum is proportional to $R\tan^2 \alpha_v$. Thus, we conjecture that we have a dependence of $\Theta((\sqrt{1+\varepsilon}-1)^{-3} R\tan^2 \alpha_v)$ on the remaining parameters.

5.2.3.2 Static Sampling

In the previous subsection, we have derived an asymptotic bound for the traversal cost of the dynamic sampling algorithm. In this subsection, we will show that the same bound also applies to static sampling. To proof this bound, we could perform a similar proof as outlined in the previous subsection. However, this is not necessary. Instead, we show that the sample spacing of octree boxes used in dynamic sampling is a constant fraction of the boxes side length. Thus, the results of the preceding section can be applied directly and we obtain the same bound. Additionally, we also get a connection between the number of sample points per octree box and the approximation accuracy for the depth factor.

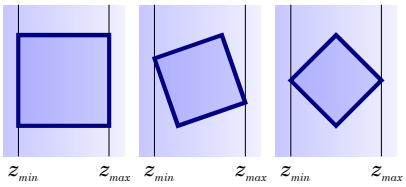


Figure 51: Octree boxes with different orientations in respect to the *z*-axis of the viewing coordinate system.

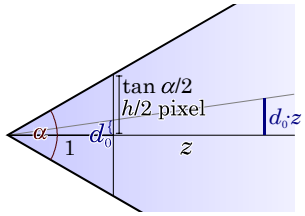


Figure 52: Relationship between sampling distance and screen resolution.

The dynamic sampling algorithm performs a depth-first search that terminates when the depth-factor of the perspective projection varies by no more than $(1 + \varepsilon)$ in the bounding box of the octree node. Let z_{min} be the minimum and z_{max} be the maximum depth of a point in such a bounding box. Then, we know[24] that

$$z_{max} = c_1\gamma \cdot z_{min} \quad \text{with} \quad c_1 \in [\tfrac{1}{2}, 1],\ \gamma = \sqrt{1+\varepsilon}$$

as the octree boxes match up the desired size up to a factor of 2. This means, that the depth of the octree box is $z_{max} - z_{min} = (c_1\gamma - 1)z_{min}$. The orientation of the box in respect to the z-axis of the viewing coordinate system may vary arbitrarily (Figure 51). Thus, the side length $size(z)$ of such a box is given by:

$$size(z) = c_2\left(c_1\gamma - 1\right)\cdot z \quad \text{with} \quad c_2 \in [\tfrac{1}{3}\sqrt{3}, 1] \tag{13}$$

Thus, the size of the octree boxes is (roughly) proportional to z. We compare this size with the sample spacing $d(z)$ required in a distance of z. Our goal is to create sample sets with a uniform spacing d_0 in the image plane. If we consider the depth factor only, this means that we need a sample spacing of $d_0 \cdot z$ in distance z to the image plane. Usually, d_0 corresponds to the size of a pixel. If we assume a vertical viewing angle of α_v and a display resolution of h pixels vertically, we obtain (Figure 52):

$$d(z) = d_0 \cdot z = \frac{2\tan\alpha_v/2}{h} \cdot z$$

The required sample spacing at distance z and the size of the approximating boxes of the dynamic sampling algorithm at distance z are both proportional to z; only a small variation due to uncertain orientation and quantization to powers of two is possible[25]. Thus, the hierarchy traversal will use the same asymptotic running time if we search for nodes in a static sampling data structure. In these nodes, the sample spacing is a fixed proportion $1/k$ of the box side length. This yields a sample spacing of $size(z)/k$. Comparing the constants in the proportionality with z, we obtain a relation between the approximation accuracy and the number of samples per box side length:

$$d(z) = \frac{size(z)}{k} \quad \text{means that} \quad \frac{2\tan\alpha_v/2}{h} = \frac{c_2\left(c_1\gamma - 1\right)}{k}$$

[24] It is also possible that the recursion stops earlier because a leaf node is reached before the desired accuracy is met. However, in this case we obtain fewer nodes in our selection. Thus, we do not need to account for such cases. It is also possible that the criterion is not met if the root bounding box is already smaller than necessary to fulfill the requirements of the approximation. In this case, our bounds are also fulfilled trivially so that we do not need to consider it either.

[25] An additional variation by a factor of $(1+\delta)$ can be caused by the node tolerance zones: The sampling distance is determined according to the side length $C(v)$ while the box $B(v)$ might be slightly larger. However, this also increases the uncertainty by a small constant factor.

$$\text{so that we obtain } \gamma = \frac{1}{c_1}\left(\frac{2k\tan\alpha_v/2}{c_2 h}+1\right),$$

$$\text{i.e. } \varepsilon = \frac{1}{{c_1}^2}\left(\frac{2k\tan\alpha_v/2}{c_2 h}+1\right)^2 - 1\,.$$

The oversampling due to the quantization of the sampling density in powers of two can be reduced by storing multiple resolutions per node (cf. Section 4.2.3), reducing the variation of c_1. c_2 is $1/3\sqrt{3}$ in the worst case. Hence, the approximation accuracy of static sampling is given by:

$$\varepsilon(k) = \left(2\sqrt{3}k\frac{\tan\alpha_v/2}{h}+1\right)^2 - 1$$

Table 2 shows examples for typical viewing conditions. It can be seen that moderate numbers of sample points per box side length only lead to a quite small maximum deviation of the depth factor. Thus, it is no problem in practice to use nodes with a quite large number of sample points (say 16-64 points per box side length). This is favorable as a block-wise processing of sample points can often increase the performance of rendering algorithms (cache coherence, burst transfers, swapping from secondary storage devices etc…).

In summary, we should note that static sampling and dynamic sampling are equivalent in terms of asymptotic hierarchy traversal costs necessary to achieve a fixed approximation accuracy of the depth factor. The approximation accuracy depends on the number of sample points per box side length (relative to screen resolution and viewing angle). For moderate numbers of sample points per box side length, good approximation accuracies are achieved.

5.2.3.3 View Frustum Overestimation

Another source of overhead is the overestimation of the extends of the view frustum. Obviously, our hierarchy traversal strategies are not able to extract an exact set of objects located in the view frustum but they compute a superset (Figure 53).

This means that the worst case overhead can be very large: In an unfavorable case, no geometry could be located in the view frustum but only in the invisible region border region that is also reported by the hierarchy traversal. This can lead to an overestimation of the sample size by an infinite factor. However, such a worst case analysis is not very meaningful for practical applications where the viewer moves around steadily. It is very unlikely that large parts of the geometry fall into the border region for many frames in such a situation. In practice, an average case analysis is more relevant: We assume that geometry falls in all regions of the view frustum and its border region with similar probability. Then, the performance degradation is given by the in-

points per box side k	2	4	8	16	32	64	128	256
depth-fact accuracy ε	1.67%	3.36%	6.8%	13.8%	28.4%	60.4%	135%	327%
view frust. overest. VF	2.54%	5.11%	10.3%	21.2%	44.4%	96.8%	225%	579%
dist-fact accuracy DF	0.42%	0.84%	1.68%	3.39%	6.89%	14.2%	29.6%	63.0%

Table 2: Overestimation of the depth factor, the projected view frustum area and the distortion factor in dependence of the number of sample points per box side length. We assume typical rendering settings: a resolution of 640×480 pixels and 60° vertical viewing angle. The table shows upper bounds, on the average, the values are smaller.

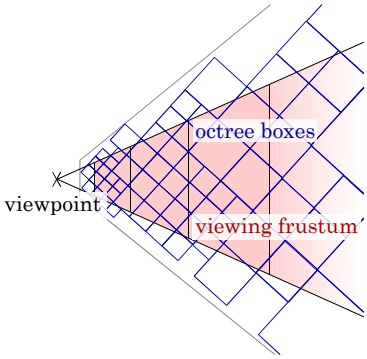


Figure 53: Covering the view frustum with octree boxes. The cross sectional area is overestimated by a constant factor.

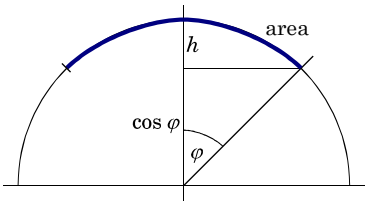


Figure 54: Area on the unit sphere in dependence of the angle φ.

crease of the cross-sectional area of the view frustum projected onto the screen. As shown in the previous subsection, the size of octree boxes in an ε-approximation of the depth factor is not larger than $(\gamma - 1)\cdot z$ at a depth of z. At the same depth, the view frustum has a cross-sectional area of $z^2 \cdot R \cdot \tan^2 \alpha_v/2$ with α_v being the vertical viewing angle and R being the aspect ratio of the image (*width* / *height*). The overestimated area is $z \cdot R \cdot \tan^2 \alpha_v/2 + 2(\mathrm{R}+1)(\gamma - 1)\cdot z^2 \cdot \tan \alpha_v/2 + 4z^2 \cdot (\gamma - 1)^2$. The projected area is obtained by dividing the cross-sectional area by z^2. We do this for both the cross-section of the view frustum and the overestimated area and obtain a constant ratio of:

$$VF \le 1 + \frac{2(\gamma-1)\cdot\left((R+1)\tan\alpha_v/2 + 4(\gamma-1)\right)}{R\tan^2\alpha_v/2}$$

For static sampling, the overestimation of the view frustum in terms of projected area can be bound more easily using a simple observation: The projected size of the sample spacing is always no larger than the on-screen sample spacing (typically the pixel size). Thus, the border of around the visible portion of the screen is not larger than $\sqrt{3}\cdot k$ times the on-screen sample spacing (i.e. typically k pixels). This yields a bound of

$$VF \le 1 + \frac{2\sqrt{3}(kh + kRh) + 12k^2}{Rh^2}$$

for a screen height of h samples and aspect ratio (*width* / *height*) of R. Table 2 shows example values for typical parameters. Note that this is an upper bound. On the average, we can expect smaller values. For dynamic sampling, we can also improve the approximation of the view frustum by enforcing a deeper subdivision level at the border of the view frustum. For static sampling, this also leads to an increased sampling density so that we do not obtain savings in terms of the expected number of sample points.

5.2.3.4 Bounds for the Distortion Factor

As it limits the spatial extends of groups of objects, our spatial subdivision strategy also bounds the distortion factor. An upper bound for the deviation of the distortion factor can easily be deduced by considering the projection of the boxes in the image plane. Again, we place the image plane in a distance of 1 to the center of projection so that we obtain a screen height of $\tan(\alpha_v/2)$ and width of $R\tan(\alpha_v/2)$ (see Figure 52). This leads to a diagonal angle of $\arctan(\sqrt{1+R^2}\,\tan(\alpha_v/2))$ for the image. The strongest variation of the distortion factor is found at the borders of the image (the derivative of $\cos^{-1}\alpha$ increases with α for $\alpha < \pi/2$). Thus, the worst case is an octree box that exceeds one of the diagonal corners of the image by its projected size of $\gamma - 1$. This leads to an

angle α of not more than $\arctan(\sqrt{1+R^2}\tan(\alpha_v/2)+\gamma-1)$. This yields a worst case bound for the overestimation of the projection factor of

$$DF \le \frac{\cos\left(\arctan\left(\tan(\alpha_v/2)\sqrt{1+R^2}\right)\right)}{\cos\left(\arctan\left(\tan(\alpha_v/2)\sqrt{1+R^2}+(\gamma-1)\right)\right)}.$$

Some example values are again given in Table 2. The deviation is relatively small in comparison with the other factors. For moderate viewing angles (in this example: vertical viewing angle 60°), the approximation of the distortion factor is only a minor problem.

5.2.3.5 The Influence of the Orientation Factor

The last factor in the projection factor that we have not yet considered is the orientation factor. We have two options for dealing with this factor. First, we can just ignore the orientation of the triangles and always use sample patterns that would be dense enough for covering the surfaces even for orthogonal viewing. For stratified, static sampling, this option is always employed because otherwise, we were forced to use anisotropic stratification patterns, which are currently not supported by our algorithms[26] (Section 4.2.3). Second, we can employ orientation classes that group triangles with similar orientation to obtain an estimate of the orientation factor. We will consider both alternatives in the following subsections. It turns out that the first, rather simplistic, is usually sufficient. Orientation classes only yield at most moderate performance improvements. The reason for this result is that ignoring the orientation leads only to small oversampling in average cases while the additional effort for an effective adaptation to the orientation factor is quite high.

Ignoring Orientation: Average Overestimation

If we ignore the influence or the orientation, we obtain a larger projected area and thus a larger sample size. In the worst case, the overestimation can result in an arbitrarily large factor: Think for example of a set of triangles parallel to the viewing direction. The required sample size is zero but the algorithm nevertheless employs several sample points, leading to an infinite approximation factor. Again, the worst case behavior is not very relevant in practice: Usually, the viewer moves around and examines objects from different viewing directions. Additionally, we can expect varying orientations of objects in the scene. Therefore, worst case configurations are usually only found for small subsets of the scene over time.

As a simple, abstract model for a "typical" scene, we assume that the orientations of the normals (as seen from the viewer) are distributed randomly, uniformly on a unit sphere. Then we compute the expected value for the orientation factor for that normal distribution: A uniform distribution on the unit sphere means that the probability that a normal falls into a surface fragment of area A is $A/4\pi$. Now let the random variable φ denote the angle between view vector and a random normal. The probability for φ being in the interval between 0 and φ_0 is thus given by

$$P(\varphi \le \varphi_0) = \frac{2\pi h}{4\pi} = \frac{1-\cos\varphi}{2}$$

[26] Our experiments with orientation classes for dynamic sampling showed that the technique usually provides only limited performance benefits. Therefore, we did not consider the usage of this technique for the (more recent) static sampling techniques anymore.

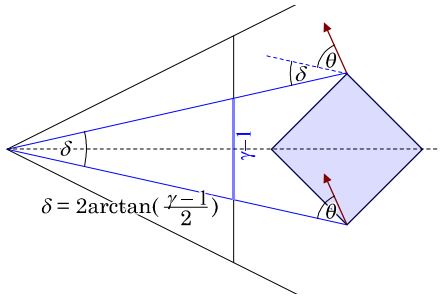


Figure 55: Orientational deviation due to spatial deviation.

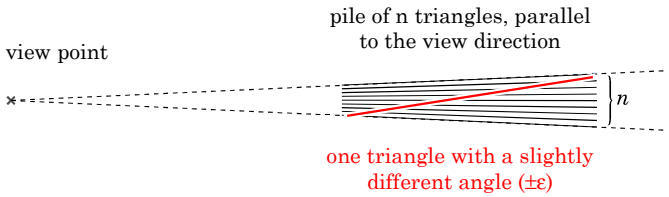


Figure 56: Worst case example for estimating the orientation factor. The projected area is overestimated by an arbitrary factor.

with h denoting the height of the cap of the sphere (Figure 54, the area formula is taken from [Bronstein et al. 97]). The probability density function is given by the derivative of the distribution function so that we obtain:

$$p(\varphi_0) = \frac{d}{d\varphi_0} P(\varphi < \varphi_0) = \frac{\sin\varphi}{2}$$

Hence, we obtain an expected value for the orientation factor of

$$\begin{aligned} \mathrm{E}\left(\left|\cos\beta\right|_{[0,1]}\right) &= \frac{1}{2}\int_0^{\pi/2} \sin\varphi \cdot \cos\varphi \, d\varphi \\ &= \frac{1}{2}\left[\frac{1}{2}\sin^2\varphi\right]_{\varphi=0}^{\varphi=\pi/2} \\ &= \frac{1}{4} \end{aligned}$$

if we account for backface-culling by setting the orientation factor to 0 for angles β of more than $\pi/2$. If we do not employ backface culling but display backfaces of triangles in the scene as well, we obtain an expected value of $\mathrm{E}(|\cos\beta|) = 1/2$. This means that the average overestimation for ignoring the depth factor (i.e. assuming a constant value of one) is a factor of two or four, respectively.

Orientation Classes

To improve the accuracy of the sampling density estimation in respect to the orientation factor, we can use orientation classes: triangles of similar normal direction are grouped together and a separate spatial data structure is build for each group of such triangles (Section 4.1.4). To analyze this strategy, we should first note that the angle between a surface normal and a vector to the viewer also depends on the spatial location of the corresponding piece of surface and not only on the normal direction. Within a spatial bounding box of projected size $\gamma - 1$, the angle to a normal may vary by $2\arctan((\gamma - 1)/2)$ (Figure 55). Therefore, we should restrict the resolution of the normal classes to a maximum angular deviation of that size. Otherwise, the accuracy of the estimate of the orientation factor is dominated by the spatial deviation.

This observation shows that the effort for restricting the deviation of the view angle to a value of $\pm\varepsilon$ leads to runtime costs of typically $\Omega(\varepsilon^{-4})$: To decrease the normal deviation we have to decrease the maximum angular deviation in the orientation classes as well as the spatial extends of the selected octree boxes. Both must be decreased proportional to ε: For the orientation classes, this is obvious. For the spatial bounding boxes, this can be seen by employing the Taylor ap-

proximation $\arctan(x) \approx x$ for $x \to 0$. Decreasing the size of the orientation classes to a fraction of ε leads to costs of $\Omega(\varepsilon^{-2})$. Decreasing the diameter of the spatial bounding boxes to a fraction of ε also leads to an effort of $\Omega(\varepsilon^{-2})$ for locally flat scenes ($\Omega(\varepsilon^{-3})$ for volume filling scenes). The two factors have to be multiplied, leading to a large effort for bounding the orientation factor.

Even after restricting the deviation of the orientation factor to a value of less than $\pm\varepsilon$, we still do not achieve a strict approximation of the orientation factor by a constant factor: For values of $\beta \geq \pi/2$, the orientation factor is zero. This means that we cannot guarantee a constant worst case overestimation factor for all groups of triangles with an interval of viewing angles overlapping $\pi/2$. In this case it is possible to obtain a arbitrary large overestimation even for arbitrarily small ε. Figure 56 shows an example: By adding triangles parallel to the view direction into a group of triangles with a maximum orientation factor > 0, we can generate arbitrarily large overestimations of the projected area.

As a consequence, we do not attempt to perform a strict ε-approximation of the orientation factor. Instead, we have to rely on a good average behavior for scenes with average normal distribution.

5.3 Image Reconstruction

The traversal algorithms described in the last section compute sample sets with fixed maximum point spacing in the image plane. Now, the next task is to reconstruct an image from the set of sample points. This process consists of two conceptual steps: First, we have to delete all invisible sample points; second, we have to fill the empty space among the remaining visible points using a scattered data interpolation technique.

5.3.1 Reconstruction of Occlusion

The reconstruction of occlusion, i.e. the removal of invisible points, is based on the z-buffer algorithm: We process all sample points linearly and points close to the viewer discard other points in their neighborhood that are farer away.

5.3.1.1 Grid Reconstruction

A simple implementation of this technique uses a regular grid of pixels. As in conventional z-buffer rendering, we use two arrays storing color and depth values. The values are initialized to background color and maximum depth. Then all points are projected onto the screen and written into the corresponding pixel if the stored depth value is larger than the new one. Triangles are handled analogously: They are projected and rasterized similar to conventional z-buffer rendering.

To avoid holes, we need sample sets that cover surfaces with sufficient density to avoid holes in the pixel grid. For static sampling, we demand a projected sample spacing of at most d_s = 1 pixel: For a z-buffer grid with a height of h pixels and vertical viewing angle of α_v, this corresponds to an object space sample spacing of (Figure 52):

$$d = \frac{2\tan\alpha_v/2}{h} \cdot z \,.$$

This approach ensures a safe coverage of continuous surfaces: The projected sample set shows a d_s-spacing on the screen, i.e. we will find a projected sample point in distance of at most $d_s/2$ to

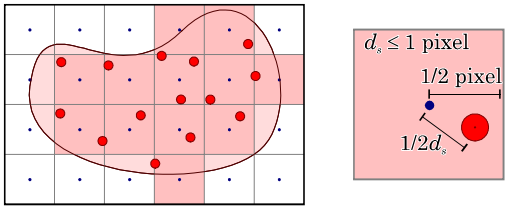


Figure 57: An on-screen sample spacing of d_s guarantees a safe coverage of closed surfaces.

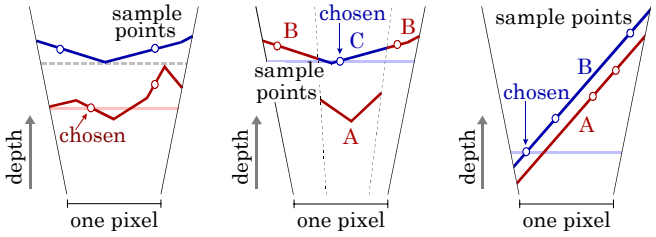


Figure 58: Different subpixel occlusion situations: (a) non-overlapping depth, (b) overlapping depth at object boundaries, (c) overlapping depth at parallel surfaces.

every point in the interior of a closed surface. This means that a projected sample point is found in distance of at most 1/2 to the center of each pixel, thus falling onto its area and covering the surface (Figure 57).

For dynamic sampling, we can calculate the necessary sample spacing d in object space according to Section 4.2.3.1. However, we can derive a tighter[27] bound by considering directly the projections of the random samples in objects space. Let $\bar{a}$ denote the projected area of the objects on the screen as computed by the hierarchy traversal algorithm (i.e. including overestimations). We measure $\bar{a}$ in pixels. Additionally, let k denote the number of sample points. If the image contains no occlusion, this means that we have to fill $p \le \bar{a}$ pixels by randomly choosing one of $\bar{a}$ pixels by k trials. p may be smaller than $\bar{a}$ because of the overestimation of the projection factor. However, the probability for choosing pixel p is always no less than $1/\bar{a}$ due to the conservative approximation. For filling these p pixels with probability of at least s, it is sufficient to use a sample size of

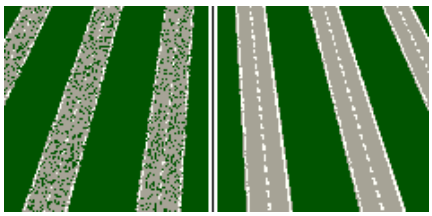
$$k = \bar{a}\left(\ln p - \ln(1-s)\right) \le \bar{a}\left(\ln \bar{a} - \ln(1-s)\right)$$

sample points (according to Equation 4, Section 1.3.3). In the case of occlusion, we can think of additional "hidden" pixels contained within the $\bar{a}$ pixels of projected area that are also filled but do not appear on the screen. Thus, a sample size of $\bar{a}(\ln \bar{a} - \ln(1 - s)) \in O(\bar{a}\log \bar{a})$ is sufficient for filling all pixels with probability of at least s. As $\bar{a}$ is known during rendering, we can compute the necessary oversampling factor (for a given security parameter s) and use it to control the sampling process. For stratified static sampling, the logarithmic factor disappears because the superfluous points created by the randomized sampling approach are already removed during preprocessing. Thus, the sample size is in $O(\bar{a})$ in this case.

Up to now, we know that pixels of continuous surfaces will be safely covered by sample points with any given probability. Now we must examine whether this is sufficient to generate correct results, i.e. the visibility reconstruction algorithm outputs only sample points that are visible from the current viewpoint. To answer this question, we consider the geometry and the sample points for a single pixel in the image. We can distinguish two cases:

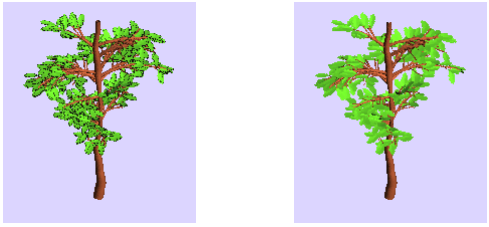
Non-overlapping depth: If we assume that a pixel is completely covered by foreground geometry and that the depth interval containing the depth values of the visible foreground is disjoint from the depth interval of the occluded background, we always obtain correct results (with high probability, according to s). As the foreground covers the complete pixel, at least one sample point must be located within the pixel area (with high probability). As the foreground sample

[27] In Section 4.2.3.1, we first divide the surface into cells in object space that are then again projected to pixel cells in image space. This leads to an overestimation of the necessary sample space by a constant factor.



(a) small offset to the ground (b) larger offset

Figure 59: Incorrect reconstruction of occlusion for polygons with small depth offset (cf. Figure 58b)



(a) dark backsides leaking through (b) heuristic solution: backface culling

Figure 60: A second example for small depth offset problems and a heuristic solution.

points all have a smaller depth value than the background points, one of them will be selected by the z-buffer algorithm. This case is illustrated in Figure 58(a).

Overlapping depth: If the depth intervals of foreground and background are not disjoint, or the foreground does not cover the pixel completely, it can happen that a background sample point exists with a larger depth than all foreground points. In this case, the background point is falsely classified as visible. Figure 58(b) shows a typical example: a small foreground fragment occludes a sample point that is falsely output as visible. Such errors are usually hardly noticeable in practice. An example that is more likely to cause problems in practice is a configuration with two parallel surfaces at very small distance. If the distance is small in respect to the pixel size scaled by the depth, the occluded surface might shine through with up to 50% probability. If static sampling with prefiltering is used, we obtain a similar problem: Typically, we obtain a mixture of both colors instead of the color of the visible surface. A simple heuristic to overcome such problems is to use a tangent plane (based on the normal vector) instead of a constant depth for depth comparison (as suggested by [Pfister et al. 2000]). During prefiltering, a similar effect could be achieved by projecting all geometry fragments on a plane in average normal direction and discarding fragments with a larger depth in normal direction. Such heuristics can improve the accuracy of the image reconstruction. However, they do not generally yield correct results. Point-based representations only reveal the properties of the geometry up to the sample spacing. Everything on a sub-sample sized level is uncertain. If occlusion effects on that level are crucial for correct rendering, we cannot expect correct results in general (see also Section 4.2.4). In such cases, we only have the option to increase the sampling resolution until the missing details are represented faithfully. Two examples of artifacts observed in practice are shown in Figure 59 and Figure 60: The first figure shows incorrect reconstructions of occlusion due to surfaces with very small depth offsets. The second figure shows leakage of backsides for thin objects (the leaves of the tree). In this case, simple backface culling can be applied to avoid the problem.

5.3.1.2 Neighborhood Reconstruction

In some cases, the simple pixel grid reconstruction technique is too slow for real-time applications. To obtain reconstructions at reduced sampling densities, we can process a complete neighborhood ("splat") of pixels for each sample point: For every pixel of the neighborhood, we perform the z-test to determine its visibility. The neighborhood is a square of d^2 pixels in the simplest case. However, more general neighborhoods are possible, such as a sphere of fixed diameter or a projection of a tangential disc, yielding an ellipsoid [Rusinkiewicz and Levoy 2000, Zwicker et al. 2001a]. It is also possible to use a varying sample spacing (i.e. varying neighborhood shapes), e.g. according to the surface curvature [Kalaiah and Varshney 2001].

(a) grid reconstruction **(b) neighborhood reconstruction**

Figure 61: Two options for reconstructing occlusion for sample sets with fixed on-screen sample spacing. The image shows how a random sample set is distinguished from the background. The neighborhood-based technique provides a better utilization of the available information.

We have two options for an implementation: A simple option is to use a z-buffer and perform the calculations on a per-pixel basis. Another option is to use a two-dimensional search structure containing the sample points in a continuous domain (usually a 2d-grid of lists of sample points). We store all points in that data structure. Then we determine all sample points in the neighborhood of each sample point and delete those sample points that are farer away than the current point. To make optimal use of the sample set, we can also use a small tolerance region in which sample points "behind" a new point are not deleted[28]. The second approach is usually slower than the first but offers a bit more flexibility. It is for example possible to use very large sample sets, allowing visibility computations at a subpixel level without being forced to use a high display resolution.

The sample spacing can be determined as before: If we use neighborhoods ("splats") of diameter d_s in screen space, we have to employ a sampling distance in object space that corresponds to a screen space sample spacing of d_s. For the correctness of the output, similar considerations hold as for grid reconstruction: In areas where foreground and background fall in different depth intervals, correct reconstructions are obtained. In other cases, problems can occur. If a depth tolerance zone is employed, special care must be taken to avoid leakage of background points. A good (heuristic) choice is to use a tolerance zone proportional to the sample spacing in object space. Additionally, backface culling should be used to delete background sample points at transitions between background and foreground areas. This avoids artifacts due to the simple depth tolerance interval.

Figure 61 compares the two options for a reconstruction of occlusion: grid-based and neighborhood-based reconstruction. The neighborhood-based technique is able to reconstruct more precise contours from the same data set while the grid-based technique suffers from quantization artifacts. However, this problem is only visible under magnification, as shown in our example scene. For a sampling density according to the pixel spacing, the artifacts are not relevant.

5.3.2 Scattered Data Interpolation

After we have determined the visible portion of the sample set, we reconstruct an image by interpolating the color values that are obtained by shading the sample points according to the given material and lighting model. To do the interpolation, we have again several options (Figure 62):

[28] This also prevents cascaded deletion effects: Without tolerance region, it might happen that a sequence of sample points with close depth and slowly varying position on the screen is completely deleted (except the foremost point).

(a) per pixel reconstr. (b) splatting (c) Voronoi regions (d) Gaussian filtering

Figure 62: Comparison of interpolation techniques at a constant on-screen sampling density.

5.3.2.1 Per-Pixel Reconstruction

The most elementary technique is to just color every pixel according to one of the sample points located within the pixel on the screen. This technique is usually used in conjunction with grid-based visibility reconstruction: Both steps are performed at once by just "painting" the projected points into a color buffer with additional z-buffer test (Figure 62a).

5.3.2.2 Splatting

To speed up the process, we can also draw larger "splats", e.g. small squares of $d \times d$ pixels or circles with d pixels diameter. If we use sample sets with on screen sample spacing not larger than d, we obtain a hole-free, piecewise constant image reconstruction (Figure 62b).

5.3.2.3 Voronoi Regions

An alternative to splatting is the use of Voronoi regions. Instead of drawing fixed-sized splats, the Voronoi region of each sample point is determined and filled with the color of the sample point. The computation of Voronoi regions in 1-norm can be performed efficiently by a region growing algorithm: Each pixel is inserted in a queue. Then the first element of the queue is picked repeatedly and the pixel is filled with the color of the sample point. Then, its 4 direct neighbors are considered. If the neighbor is still empty, it is inserted into the queue and marked as filled. This procedure computes the discrete 1-norm Voronoi regions in linear $O(p)$ time. However, the image quality is only improved slightly in comparison to simple splatting. Thus, the technique is probably not worth the additional effort (Figure 62c).

5.3.2.4 Averaging

Whenever we use sample sets without prefiltering, we have to deal with noise and aliasing issues. In the case of dynamically generated random sample sets, we mostly obtain noise artifacts. These can be eliminated by calculating several images, independent of one another and compute their arithmetic average. This technique can strongly improve the image quality. However, due to the $O(\sqrt{n}^{-1})$ convergence speed, several images are necessary to obtain noise free solutions. As a rule of thumb, 10 images yield an acceptable quality while several hundred are necessary for a truly noise-free solution (cf. Section 1.3.2). For static sampling without prefiltering, we also need antialiasing but cannot perform independent renderings for averaging. Instead, we can use higher resolution sample sets (deeper hierarchy levels). This yields multiple sample points per pixel, which can be used for averaging.

5.3.2.5 Gaussian Reconstruction

Averaging several images is only a heuristic. We can still obtain aliasing effects. One reason for this is the "depth-dominance" of the z-buffer-based reconstruction. Whenever several sample points fall onto one pixel, only the foremost is taken. Usually, a lot of points fall onto one pixel: Random sampling with safe coverage needs e.g. about 20 sample points per pixel on the average. This leads to aliasing effects very similar to those known from conventional z-buffer rendering. The problem can be avoided by using neighborhood-based visibility reconstruction with continuous point sets (as described in Section 5.3.1.2). The depth tolerance of the enhanced visibility reconstruction avoids depth dominance and thus z-buffer aliasing. After all occluded points have been deleted, an image grid is defined and a reconstruction low-pass filter kernel is placed at every sample pixel of the image grid. The search structure of the visibility reconstruction step is then queried to determine all remaining, visible points that overlap with the reconstruction kernel. Then, the weighted average of the corresponding color values, using the function value of the reconstruction filter as weights, is computed to determine the color values for the pixels. This technique is of course more expensive than simple z-buffer-based image reconstruction but it yields a superior image quality with little aliasing effects.

5.3.2.6 Advanced Splatting Techniques

It is possible to modify the simple splatting algorithm to produce images with low aliasing. Two options have been published in literature that are useful for different applications:

"Surface Splatting" [Zwicker et al. 2001a]

[Zwicker et al. 2001a] propose an implementation of reconstruction with Gaussian filter kernels using splatting: For each sample point, a Gaussian splat with a standard deviation proportional to the pixel spacing is drawn into the framebuffer using additive blending. In addition to the color buffer, a weight buffer is also used in which the weight sums of the kernel functions are stored, too, for later renormalization. The technique can be modified to support more general elliptical splat primitive: A two-dimensional Gaussian splat is associated with each sample point and projected to the screen. By limiting the main axes of the projected ellipse to 1 pixel, aliasing is avoided. A depth buffer and a depth test with a small depth tolerance can be used to perform visibility reconstruction in one pass. The technique is superior to our Gaussian reconstruction algorithm [Wand 2000a] described before both in terms of performance and generality.

Alpha-Blending [Rusinkiewicz and Levoy 2000]

Another reconstruction technique from literature is using alpha-blended Gaussian splats. The splats are drawn in front-to-back order, performing occlusion and image reconstruction in one step. This technique does not perform correct antialiasing for surface objects. However, it yields better results for volume filling scenes, such as point sampled trees with many small leaves and branches. We have implemented the technique and modulated the Gaussian splats with a conservative opacity estimation computed from the number of sample points per unit area, as described in Section 4.2.4.4. For unstructured scenes such as a forest of trees and bushes, the technique yields good results. For surfaces with structured patterns, aliasing artifacts occur as alpha blending does not compute a correct averaging of the color values. Probably, both approaches (additive reconstruction with basis functions and alpha blending) could be combined in future work: First a *volume* of attributes (transparency, normals, color) is reconstructed at a subpixel scale by adding radial basis functions around sample points. Then the volume rendering integral for the "ray volume" is computed using successive alpha blending.

5.4 Overall Efficiency

In this subsection, we summarize the results concerning performance of forward mapping rendering: We distinguish three cases: Dynamic sampling, static sampling with random sample sets and static sampling with stratified sampling.

In all three cases, we need $O(h + \log \tau)$ time for hierarchy traversal with h denoting the height of the octree and τ the maximum relative depth range of the scene. Then, we have to choose sample points from the nodes. Let $\bar{a}$ denote the projected area as estimated by the hierarchy traversal algorithm. We assume that the scene shows uniform normal distribution and a uniform probability for geometry to be located anywhere within the extended view frustum, as reported by the traversal algorithm. Then, on the average, $\bar{a}$ is proportional to the exact projected area a of the scene on the screen: The depth factor is approximated up to a constant factor $1 + \varepsilon$, the distortion factor is bound by a constant, the view frustum is overestimated by a constant factor, and the average orientation factor is also overestimated by at most a constant factor. Thus, we have $\bar{a} \in O(a)$.

The sample size depends on the sampling pattern: For random sampling, we need $O(a \log a)$ sample points, for stratified sampling we need only $O(a)$ sample points. The costs of determining the sample points also depend on the data structure: For static sampling, each sample point can be chosen in $O(1)$ time, for dynamic sampling we need $O(\log n)$ time where n is the number of triangles in the scene. Overall, we obtain the following performance characterization:

The *dynamic sampling* algorithm needs $O(h + \log \tau + a \cdot \log a \cdot \log n)$ rendering time. The static *sampling algorithm* with *random sample sets* needs $O(h + \log \tau + a \cdot \log a)$ time. And the static *sampling algorithm* with *stratified sample sets* needs $O(h + \log \tau + a)$ time to render a scene. Note, that the projected area a also contains occluded geometry, it is not bound by the screen resolution. In order to reduce the dependence of the running time on the occlusion density in the scene, we have to combine the rendering technique with an occlusion culling algorithm. Most techniques described in Section 2.2.5 could be employed. However, this is beyond the scope of this thesis.

An interesting special case is “disc-like” scenes. Here we assume that the geometry is distributed uniformly on a disc of Radius R within a small constant height H above the disc. Such scenes are often found in applications. For example, scenes of cities or landscapes roughly fit into this model. In this case, we have $n \in O(R^2)$ primitives in the scene. The projected area (again assuming uniform distribution of normals) can be estimated by integrating the depth factor over the disc (assuming the viewer to be located in the middle, which is the worst case):

$$a(R) = \int_{z_{near}}^{R} \frac{O(H)r}{r^2} dr = \int_{O(1)}^{R} O(1/r) dr = O(\log R) = O(\log n)$$

This means that we expect a rendering time of $O(h + \log \tau + \log^2 n \cdot \log \log n)$ for dynamic sampling, $O(h + \log \tau + \log n \cdot \log \log n)$ for static random sampling and $O(h + \log \tau + \log n)$ for stratified static sampling, without any occlusion culling. Of course, the constants in the O-notation depend on the occlusion density in the scene.

Chapter 6
Backward Mapping

In this chapter, we examine the usage of point-based multi-resolution rendering techniques in the context of backward mapping (raytracing) algorithms. We describe a multi-resolution raytracing algorithm that makes use of point hierarchies. In contrast to the forward mapping techniques that usually run at interactive framerates, the raytracing algorithm targets at offline-rendering applications. The algorithm supports rendering of more general global illumination phenomena, specifically classic distributed raytracing effects such as depth-of-field, blurry reflections, soft shadows with full antialiasing. In contrast to conventional raytracing algorithms, the point-based multi-resolution algorithm permits an approximation of these effects at the cost of one primary ray per pixel.

6.1 Motivation

In the last chapter, we have shown that using a point-based multi-resolution hierarchy can drastically speed up the rendering of complex scenes with forward mapping (z-buffer) rendering techniques. Hence, it is an obvious question whether point-base multi-resolution data structure can also be applied to accelerate backward mapping (raytracing) rendering techniques. In this chapter, we are going to examine this topic more in detail. First, we should note that the performance characteristics of raytracing are different to that of simple forward mapping: Raytracing is usually implemented using a spatial hierarchy as auxiliary data structure to speed up the ray queries. Thus, the algorithm already shows a strong output-sensitive time complexity for most scenes in practice (cf. Section 2.3.2). A typical configuration is a ray hitting a locally flat surface that is composed of many small triangles. If the ray hits the surface at a large (orthogonal) angle, the search time is usually logarithmic in respect to the number of fragments the surface is composed of (see Figure 13b in Section 2.3.2). If we use a multi-resolution hierarchy, we can stop the search at a higher level, when the point spacing is small enough, i.e. about as large as the distance of neighboring rays.

This technique could potentially reduce the query costs. However, we cannot expect large savings: Pruning the depth of a search tree that has already a logarithmic depth will typically not have a strong effect on the running time. Hence, the level-of-detail control will not lead to performance gains comparable to those observed in forward mapping algorithms. Nevertheless, the multi-resolution approach can still improve the efficiency, but for more subtle reasons: A key

problem in raytracing is aliasing: Each ray that has been shot through a pixel only yields a single sample point of a ray surface interaction, prone to aliasing artifacts. Using a multi-resolution hierarchy allows us to use prefiltered sample points to reconstruct local surface properties, similar to mipmapping in texture mapping. This can potentially eliminated most aliasing artifacts.

The classic solution to the aliasing problem in raytracing is stochastic supersampling: Multiple rays are shot through each pixel and the results are averaged using a suitable filter kernel. The central limit theorem guarantees that this average will converge to the integral over the pixel (weighted by the filter kernel) stochastically. The problem of this approach is slow convergence: In the worst case, the convergence rate (standard deviation of the error) is $O(\sigma/\sqrt{n})$ for n sample rays and a standard deviation σ of the color estimator (see Section 1.3). As summarized in Section 1.3, the convergence speed can be improved using importance sampling and stratified sampling. However, in adverse cases, no improvement can be achieved. If the image signal is highly irregular, with large variance, neither stratified sampling patterns nor adaptive sampling strategies accelerate the convergence. Such adverse cases of highly irregular image signals can often be expected for images of highly detailed scenes showing complex geometry.

An alternative are methods that trace extended ray volumes such as cone tracing or beam tracing [Amanatides 84, Heckbert and Hanrahan 84, Kirk 87, Shinya et al. 87, Ghanzanfarpour and Hasenfratz 98]. These methods do not shoot infinitesimally small rays into the scene but larger cones with a cross-section corresponding to a pixel in the image. These techniques render anti-aliased images using only one ray per pixel. However, they suffer from a different kind of complexity problem: In a highly detailed scene, the cross-section of a ray cone may intersect with an arbitrarily large set of primitives. Thus, the intersection computations become prohibitively expensive. For this reason, methods following this paradigm are usually only applied to models of low complexity.

A possible solution is to use a hybrid approach [Amanatides 96, Genetti et al. 98]: Extended ray cones are used to detect boundaries in objects space. Then, super-sampling is used to integrate over the cross-section of the ray. These techniques allow a good control of the sampling density used for oversampling. Nevertheless, in regions of high variance, they suffer from the same convergence problems as the purely stochastic methods.

Using a point-based multi-resolution data structure, we can devise a raytracing technique with extended ray volumes that does not have the complexity problems of the classic cone/beam-tracing approach. Instead of intersecting the extended ray volume with potentially millions of geometric primitives we use only a few sample points with a spacing matching the ray footprint (i.e. cross-section of the ray cones). We use sample points with prefiltered attributes (precomputed average color attributes and differential properties such as an average normal and curvature information). Analogous to mipmapping in texture mapping, we can estimate the integral over the ray cross-section using a few of such prefiltered sample points in a footprint-assembly [Schilling et al. 96]. To determine the shape of secondary rays, a technique similar to ray differentials [Igehy 99, Schilling 2001] is used: The broadening or shrinking of the ray volumes is computed depending on the local surface curvature and the incoming ray directions.

This technique allows an approximate rendering of antialiased images. Additionally, we can also approximate effects such as soft shadows, depth-of-field, or blurry reflections by modifying the shape of the ray volumes. In either case, only one (primary) ray per pixel has to be used instead of several rays for a stochastic approximation.

```
Algorithm MRRaytracing(Image i, PointHierarchy H)
 For each Pixel p in i Do
   color(p) := recursiveTracing(primaryRay(p), root(H));
 End For
```

Algorithm 4: Multi-resolution point-sample raytracing (main procedure).

```
Algorithm recursiveTracing(RayVolume r, Node v)
   // empty list of intersections
   Intersections f = ∅;
   // compute intersections using multi-resolution hierarchy
   traceRay(primaryRay(p), root(H), f);
   // merge adjacent intersections to compute continuous surfaces
   computeSurfaceFragments(f);
   For each resulting surface fragment s in f Do
     color(s) = shading(s,r)
     If not maximum recursion depth reached Then
       // shoot secondary rays for resulting surface fragments
       If reflective(s) Then
         color(s) += recursiveTracing(reflectedRay(r,s), v);
       End If
       If transparent(s) Then
         color(s) += recursiveTracing(refractedRay(r,s), v);
       End If
     End If
   End For
   // compute resulting ray color from colored fragments
   Color c := compositing(f);
   Return c;
```

Algorithm 5: Recursive tracing of secondary rays: Intersections between ray and geometry are computed, the intersections are merged to surface fragments and secondary rays are shot for each intersected surface if necessary. In the end, all resulting color values are combined by a compositing algorithm.

6.2 The Raytracing Algorithm

In this subsection, we describe the multi-resolution raytracing strategy more in detail. We start with a conceptual overview of the new algorithm:

6.2.1 Overview

To perform point-based multi-resolution raytracing, we need four ingredients: First, we need a suitable point-based multi-resolution representation. The point clouds substituting more complex geometry must especially provide enough information to estimate the shape of secondary rays. Second, we need a ray model that describes ray volumes with few parameters. It should approxi-

```
Algorithm traceRay(RayVolume r, Node v, Intersections f)
 // check for intersection with current node
 If intersection(B(v), r) Then
   // first: check for intersections with "large" triangles
   For each Triangle t in v Do
     hit := intersection(t, r);
     If hit.intersection() Then f.addIntersection(hit);
   End For
   // second: check for appropriate point spacing
   If minRayDiameter(r, v) ≥ pointSpacing(v) Then
     // spacing ok -> compute point intersections
     For each PointSample p in v Do
       hit := intersection(p, r);
       If hit.intersection() Then f.addIntersection(hit);
     End For
   Else
     // otherwise -> recusive descent
     For each child c of v in front-to-back order Do
       // inspect node only if it is still visible along the ray
       If not occluded(r, c) Then
         traceRay(r, v, f);
       End If
     End For
 End If
```

Algorithm 6: Procedure for tracing a single ray and reporting all intersections. The algorithm descends into the hierarchy to find all intersections with point primitives of adequate sampling distance. If "large triangles" are found on the way down the hierarchy, they are also tested for intersection.

mate the volume filled by all rays passing through a pixel and the corresponding secondary ray volumes. We also need a technique to derive the parameters of secondary rays from surface properties and the parameters of an incoming ray. Third, we need intersection tests between the elements of our hierarchy (points, triangles, bounding volumes) and the extended rays. Lastly, we need a compositing algorithm that computes the integral color returned by a ray after several intersections with pieces of geometry. Note that a ray volume usually intersects multiple elements of the hierarchy (points, triangles) until it is fully occluded and the search can be terminated.

Using these ingredients, we can describe the high-level algorithm: It consists of three parts, analogous to a conventional raytracing algorithm. The first (trivial) part, the main procedure (Algorithm 4), iterates over all pixels of the image, creates a primary ray and calls the recursive raytracing procedure (Algorithm 5) to compute colors for each pixel. The recursive raytracing algorithm first calls the hierarchy traversal algorithm (Algorithm 6) to compute all intersections of the extended ray with the scene. The algorithm returns a list of all intersections with points of matching size and triangles for parts of the scene that are too large for an efficient point-based representation. Afterwards, the list of intersections is post-processed: Intersections with multiple points or triangles belonging to the same surface are merged into single surface fragments. A

suitable low-pass filter kernel is used to avoid aliasing in this resampling step. For each of the detected surface fragment, the material properties are evaluated. If necessary, secondary (extended) rays are computed and traced recursively. In the end, after the final colors of all surface fragments have been determined, a compositing algorithm is called that combines the color values into a single resulting color.

The multi-resolution ray query is performed within Algorithm 6, which is called for the intersection calculation. This algorithm performs a depth-first descent into the hierarchy, traversing the octree nodes in front-to-back order. It recursively checks each node for intersection. If no intersection is detected, the traversal is stopped. Otherwise, the triangles in the node are tested for intersection (again, during preprocessing, large triangles have been stored on the way down the hierarchy to avoid point sampling of low-resolution portions of the scene). Then, the algorithm estimates the minimum diameter of the extended ray volume. If this value is in the same range as the point spacing, the traversal is stopped and all point primitives of the node are tested for intersection. If the resolution of the current node is still too coarse, the hierarchy traversal is continued. Before any child node is entered, the algorithm checks whether the bounding box of the node is potentially visible. If the node is already occluded due to former intersections, the traversal is also stopped.

In order to describe the algorithm more in detail, we now consider the different building blocks one after another.

6.2.2 Data Structure

First of all, we need a multi-resolution point hierarchy. Theoretically, we could use all data structures proposed in Chapter 4. As our goal is efficient antialiasing, it makes sense to employ prefiltered sample points. Thus, we also have to use static sampling. It also makes sense to employ a stratified sampling pattern, such as neighborhood-based stratification or quantized grid stratification, as this causes the least overhead. It would also be possible to use non-prefiltered samples or even random dynamic samples. In this case we would have to perform supersampling in object space on the fly in order to fight noise artifacts. However, filtering during preprocessing simplifies the algorithm and leads to a higher raytracing performance.

Note, that the construction of prefiltered sample points according to our technique proposed in Section 4.2.4.2 also requires stochastic integration using several sample points. Nonetheless, this is much faster than stochastic sampling in the image plane. In the first case, only a set of sample point in object space have to be generated and averaged, which typically is a linear time problem. For sampling in the image plane, a ray query has to be answered for every sample point, which is much more expensive. In addition, the prefiltering is precomputed, which is advantageous if multiple frames of an animation with moving observer have to be rendered.

Special care must be taken considering the attributes stored with each sample point. Later, we will use the point representation to estimate the shape of secondary rays, which are focused or broadened by curved reflectors or refractors. Thus, we must be able to reconstruct both the surface orientation and curvature from the point-based representation. We account for these needs by computing *differential sample points* [Kalaiah and Varshney 2001] during preprocessing. This means, we store the normal and derivative information in two tangent directions for each sample point.

To compute prefiltered surface attributes, we use two point sample sets: As described in Section 4.2.4.2, a low density, stratified point set is stored in each octree node as "*representative*"

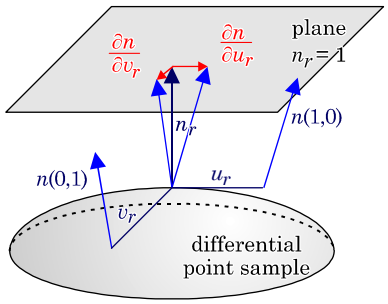


Figure 63: Differential sample points.

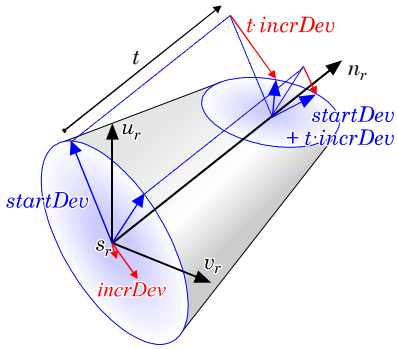


Figure 64: Ray coordinates.

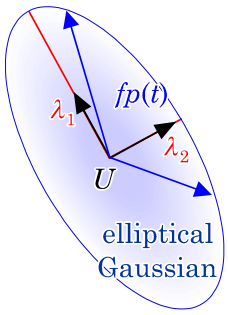


Figure 65: Principal component analysis.

points and an additional high density sample set is computed to perform prefiltering: A Gaussian radial basis function is placed around each representative sample point. All points of the high density set are determined and a weighted average of their color attributes and normal vectors is computed to obtain an average normal and color.

To obtain the derivatives of the normal, we fix an (arbitrary) tangential coordinate system (u_r, v_r) orthogonal to the average normal n_r of the representative sample point. Then we project all neighboring points into this coordinate system, yielding points (u_i,v_i) with normals $n(u_i,v_i)$. For each point, we calculate a two-dimensional normal deviation $\Delta n(u_i,v_i)$: The two components are the deviation to the original normal in u_r and v_r coordinates for a fixed third component $n_r = 1$ in normal direction (see Figure 63). This means, we measure the normal deviation as projection in a plane parallel to the tangential plane with distance one, as proposed by [Schilling 2001]. This reduces the amount of data to be stored for the derivatives (4 instead of 6 values).

To describe the surface curvature, we fit a bilinear function $n(u,v) = n(0,0) + u \cdot \partial n/\partial u + v \cdot \partial n/\partial v$ to the normal deviations $\Delta n(u_i,v_i)$ by solving a weighted least square problem, using the radial Gaussian filter function to determine the weights. The derivatives of this function $(\partial n/\partial u, \partial n/\partial v) = \nabla n$ are stored in the representative point along with the orientation of the tangential coordinate system (one vector is sufficient, the second can be computed via cross-product). Note that we fit a function to the normal vectors rather than fitting a height field to the positions of the sample points. This is necessary because we have to deal with normal interpolated triangles as input (the point hierarchy is build for a triangle model with per-vertex normals as input). Thus, the spatial deviation of the sample points might not match the specified normals. Subsequently, all point properties are quantized to small integer values (8 bit for color and material properties and position, 16 bit for differential properties) so that they can be stored compactly in the hierarchy.

6.2.3 Ray Representation

Next, we need a model that describes the ray footprint along a ray with few parameters and sufficient accuracy. Generally, ray footprints are obtained by approximating the propagation of the set of all rays through a single pixel through the scene. For primary rays, this is easy. Dealing with secondary rays is more involved: There are two different basic approaches: First, one can estimate the ray footprint by considering the differences between adjacent rays in image space [Genetti et al. 98]. This approach leads to problems at the boundary of objects, where special processing is necessary. The second option is to use differential information at the point of intersection, i.e. derivatives of the normal, to estimate the broadening or focusing effect of the surface on the in-

coming ray. We use the second strategy in a similar way as [Igehy 99]. This approach fits especially well in the context of a multi-resolution renderer where differential properties can be precomputed for different levels of resolution.

Our ray model is based on a linear approximation: Every ray r defines a local coordinate system, consisting of its origin s_r, its normalized direction n_r, and two tangential directions u_r, v_r (see Figure 64). The footprint is now described in the local u_r and v_r coordinates: Each ray stores two 2×2 matrices *startDev* and *incrDev*. The columns of *startDev* contain two vectors (in u_r,v_r coordinates) defining the footprint coordinate system at the ray parameter $t = 0$. The footprint at larger values of t is defined as

$$fp(t) = startDev + t \cdot incrDev. \tag{14}$$

The footprint is a matrix consisting of two column vectors describing a parallelogram corresponding to the extends of the ray volume at parameter t. The footprint can also be interpreted as a coordinate system. This coordinate system is used to associate a filter kernel with each ray parameter t. We use an elliptical Gaussian filter [Zwicker et al. 2001a] defined by:

$$weight(u,v) = e^{-\left(fp(t)^{-1}\begin{pmatrix}u\\v\end{pmatrix}\right)^2} = e^{-(u,v)fp(t)^{-2}\begin{pmatrix}u\\v\end{pmatrix}} \tag{15}$$

The columns of $fp(t)$ can be interpreted as a new coordinate system in which the coordinates from the (u_r, v_r) coordinate system have to be transformed before evaluating a unit Gaussian filter. Thus, the exponential decay of the filter weight is given by a quadric form of the coordinates of a point in local ray coordinates (u,v) using the matrix $fp(t)^{-2}$. This matrix is real and symmetric (because it has been squared before). Therefore, it can be decomposed into an eigensystem representation

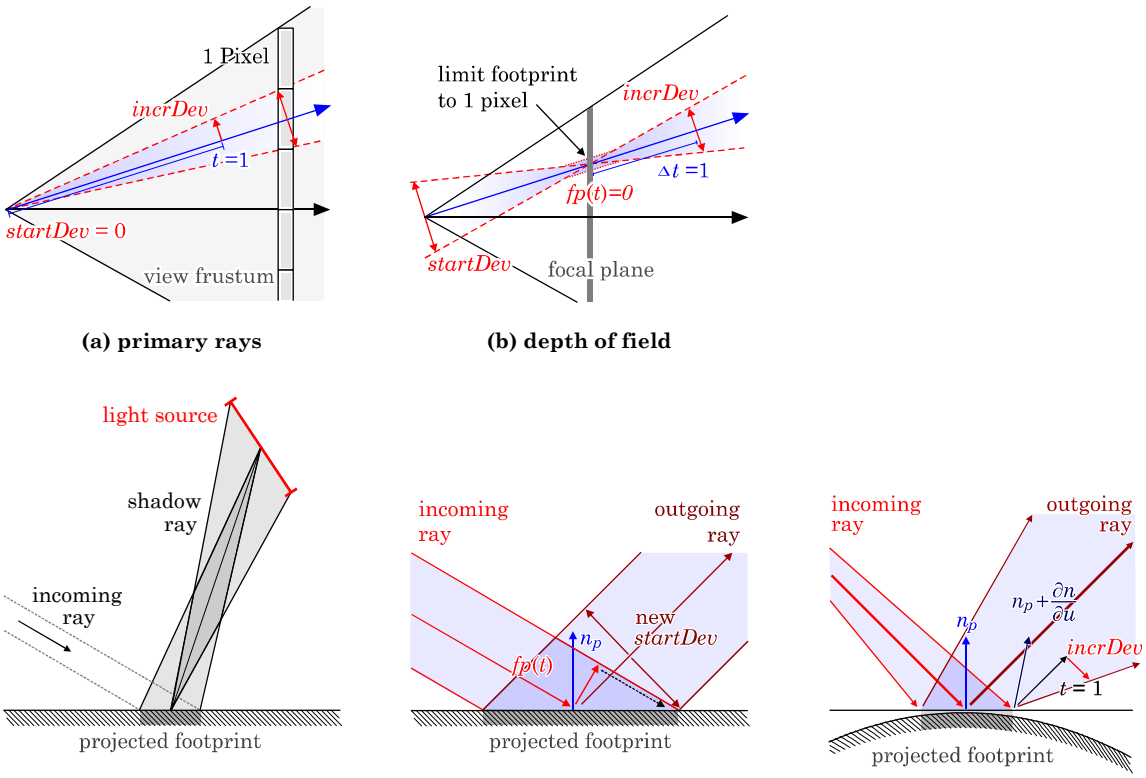
$$fp(t)^{-2} = U^T \begin{pmatrix} \lambda_1^{\,2} & 0 \\ 0 & \lambda_2^{\,2} \end{pmatrix} U \tag{16}$$

with an orthogonal Matrix U (see Figure 65). The eigenvectors point into the direction of the main axes of the Gaussian ellipse and the square roots (λ_1, λ_2) of the eigenvalues (λ_1^2, λ_2^2) are the length of the principal axes. We will use this decomposition for several purposes, e.g. to enhance the numeric stability of triangle intersection calculations or to estimate the minimum diameter of the ray footprint.

6.2.4 Ray-Surface Interaction

After defining the ray model, we discuss how we obtain ray parameters for primary and secondary rays as well as for shadow rays:

Primary Rays: Primary rays are constructed by specifying zero deviation at the ray origin and an increment matrix that broadens the ray so that it matches the extents of the pixels in the image plane (Figure 66a). This also allows depth of field effects: The lens model used by [Cook et al. 84a] leads to a diameter of the ray footprint that increases linearly to both sides of the focal plane (Figure 66b). This effect can be modeled by setting values for the deviation matrices such that the footprint coordinates are zero in the focal plane. To avoid aliasing in the focal plane, we must additionally compare the ray diameter (i.e. the eigenvalues of the $fp(t)$) with that of a conventional primary ray and take the maximum of both.



(a) primary rays **(b) depth of field**

(c) shadow rays **(d) footprint projection** **(e) reflected rays**

Figure 66: Ray parameter setup for primary and secondary rays

Shadow Rays: We allow arbitrary ellipses in three-space as light sources. The footprint of the shadow ray consists of *two parts*: First, an elliptical cone with the cross-section of the light source at the source and zero at the current surface intersection point, and second, an elliptical cone with the cross-section of the footprint of the intersection at the surface and diameter zero at the light source (Figure 66c). The footprint at the surface intersection point (needed for the first part) is calculated by projecting the incoming ray footprint onto the tangent plane of the surface intersection point.

Projecting a ray onto an intersection surface is illustrated in Figure 66d: We determine the ray parameter t for the intersection point, compute $fp(t)$, and transform $fp(t)$ into three-dimensional world coordinates by multiplying with the matrix of local ray coordinates. Then, we project the resulting two vectors onto the plane of the intersected surface fragment. In order to form a secondary shadow ray, we use the projected ray to compute *startDev* of the outgoing ray: The projected ray vectors are transformed into ray coordinates of the outgoing shadow ray by computing all scalar products of projected vectors with the local ray coordinates of the outgoing ray. *incrDev* is chosen to yield a zero footprint at the light source (i.e. $-startDev/\Delta t$).

The footprint at the light source (needed for the second part of the shadow ray) is obtained by transforming the two axes of the light source ellipse in ray coordinates. Then again, startDev is chosen to yield a zero footprint at the (other) end of the ray (i.e., again $incrDev = -startDev/\Delta t$). After forming two ray cones according to our linear model (Equation 14), the footprint at any ray parameter t is given by the convolution of the two footprint matrices of the two rays. The convolution can be computed by simply adding the squared footprint matrices [Zwicker et al. 2001a] be-

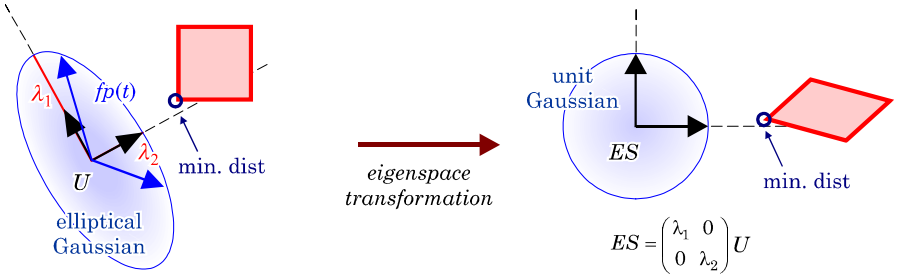


Figure 67: Intersection calculations with general primitives.

fore the evaluation of Equation 15. Another possibility is to use only one ray and interpolate between the start and the end coordinate system by matching (also by possibly mirroring) the footprint coordinate axes with similar direction. This is less exact because the linear interpolation cannot capture a potentially rotating, asymmetric footprint without distortion. However, the results in practice are satisfactory (this option is employed in the implementation).

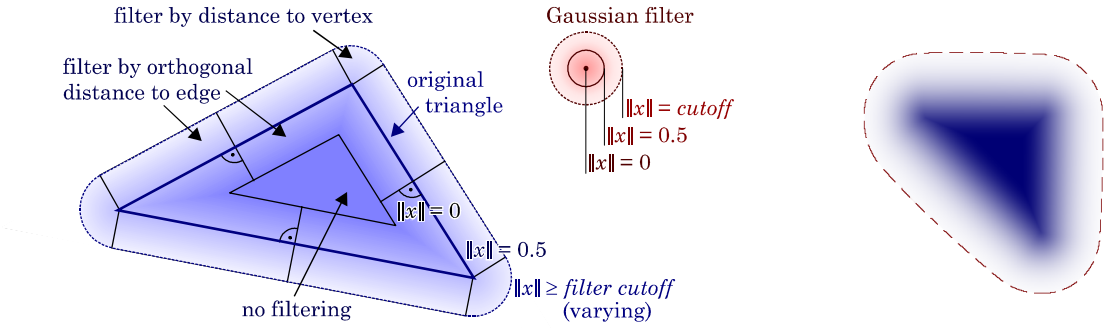
Reflected Rays: First, we reflect the center ray of the incoming ray cone at the normal of the intersection point. This yields a center ray for the reflected ray cone. It remains to calculate the ray footprint parameters *startDev* and *incrDev* of the reflected ray cone. To calculate *startDev*, we project the footprint of the incoming ray in ray direction onto the local tangent plane of the intersection point p, as described before (Figure 66d). This yields two vectors u_{fp}, v_{fp} that describe a footprint ellipse on the surface. Then we express the two vectors u_{fp}, v_{fp} in local ray coordinates u_r, v_r of the outgoing ray to obtain the columns of *startDev*.

To calculate *incrDev*, we estimate the normal directions at the points $p+u_{fp}$, $p+v_{fp}$ using the first order approximation matrix ∇n for the normals at the point of intersection. For each of the two points, we compute a direction of reflection, as shown in Figure 66e. Then the differences between the central direction of reflection and the directions at the two points are expressed in the local ray coordinates u_r, v_r to obtain the columns of *incrDev*.

Transmitted / Refracted Rays: Transmitted rays are handled the same way as reflected rays. The only difference is that the incoming vectors are refracted instead of reflected to calculate the outgoing directions. A small problem arises with total reflection: It is possible that a part of the rays in the footprint is reflected while others are refracted. In this case, we just decide for refraction or total reflection based on the direction of the center ray. However, this can lead to aliasing at the border of reflection and refraction. To remove this (subtle) source of aliasing, we could also send two ray cones and blend together the results of the two rays.

6.2.5 Intersection Calculations

We need three types of intersection calculations: Extended ray volumes must be tested for intersection with points, triangles and bounding volumes of the hierarchy. First, we need a criterion for an intersection with the ray volume. The ray volume is given by a varying elliptic Gaussian filter kernel around the central ray. Theoretically, this filter kernel has infinite support. However, the exponential function drops very quickly and we can safely truncate the filter for larger values. Usually, we set the filter value to zero for regions in which the filter value is below 2% of the maximum (i.e. $\|x\| > 2$ in e^{-x^2}, 2.8× the standard deviation), which is a good trade-off between efficiency and quality.



(a) Computing the distance to the closest point of the triangle. Two cases are distinguished: Points closest to an edge (also within the triangle) and points closest to a vertex. Afterwards a Gaussian filter is applied, shifted by $\Delta x = 0.5$.

(b) rendering result

Figure 68: Intersection and weight calculation for triangles

To determine intersections, we use two different strategies: Intersections with points can be handled directly by projecting the point into the footprint coordinate system and evaluating the filter function. Intersections with extended primitives (triangles, different types of bounding volumes) can be handled in a uniform way by projecting all vertices of the primitive into the (varying) footprint coordinate system and then computing the minimum distance to the object.

6.2.5.1 Ray-Point Intersection

To compute the intersection between a surface sample point p and an extended ray r, we first express the point in ray coordinates: Let d be the difference vector between the point p and the ray origin. The scalar product between d and the (orthogonal) ray coordinates n_r, u_r and v_r (Figure 64) expresses the point coordinates in ray coordinates. The n_r component is the ray parameter t at the point of the ray that is closest to the sample point. Thus, we can evaluate $fp(t)$ (Equation 14) and calculate a weight for the point p (Equation 15). If the weight is below the given threshold (usually 2% of the maximum), no intersection is reported.

6.2.5.2 Ray-Triangle Intersection

Our multi-resolution hierarchy uses the original triangles to represent the highest resolution of the model. Thus, we must also compute anti-aliased intersections of extended rays and triangles.

This is done in three steps: First, we transform the vertices of the triangle in ray coordinates. Then we evaluate the footprint matrices at the n_r-components. These matrices define a coordinate system in which the reconstruction filter is just a unit Gaussian. Therefore, in the second step, we transform the coordinates of the three vertices of the triangle in the footprint coordinate systems at the three points. We end up with a two-dimensional triangle and a unit Gaussian around the center of the coordinate system (Figure 67). In the last step, we calculate the distance d of the closest point of the triangle to the origin and set the weight of the intersection to $\exp(-(d-0.5)^2)$. This leads to triangles with a border blurred by the ray footprint (see Figure 68). The shift of 0.5 creates a shift of one in the arguments of the exponential function between the filter kernels of triangles adjacent to the same edge. This leads to a uniform weight sum (the Gaussian kernels approximately sum to one, [Zwicker et al. 2001a]). The interpolation of the footprint along the edges of the triangle is done implicitly as we use three different matrices for the transformation of the three vertices.

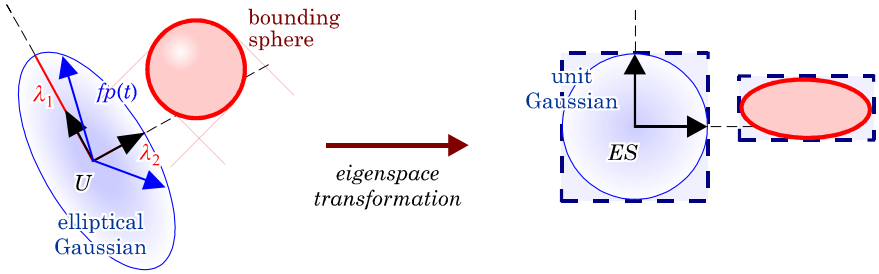


Figure 69: Simplified bounding sphere hierarchy intersection test. The center of the bounding sphere is transformed into the eigensystem of the ray (in which the filter kernel is a unit Gaussian). The a bounding box is formed by scaling the radius of the bounding sphere by the eigenvalues and a simple bounding box overlap test with a bounding square at the origin is used as conservative estimate for intersection.

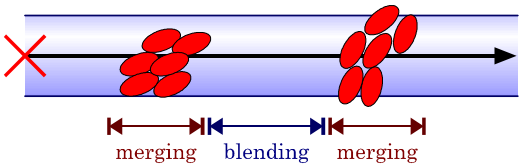


Figure 70: Identifying sample point intersections from the same surface by merging depth intervals: Overlapping depth intervals are merged, others are blended (alpha blending or subpixel mask compositing).

A further issue is the computation of the attributes of the "intersection point" that is stored in the list of intersections. In addition to the weight, we also have to interpolate the normals, colors and material attributes stored at the vertices. We use a linear interpolation based on barycentric coordinates, which are evaluated at the intersection point where (the center of) the ray hits the triangle. It is possible that this point is located outside the triangle, leading to negative interpolation values. We detect this special case and project the intersection point onto the closest edge of the triangle to determine the barycentric coordinates for the interpolation. The derivative of the normal is determined analytically: For each triangle, we precompute the normal derivative in the direction of one edge and orthogonal to this edge and project the result to a plane with distance 1 in normal direction. This coordinate system in conjunction with the two derivative vectors is reported in case of an intersection, being consistent with the information reported for sample points.

6.2.5.3 Ray-Bounding Volume Intersection

To perform the hierarchy traversal, we also need to compute intersections of extended rays with bounding volumes. A conservative intersection test can be performed using the same basic technique as for the intersection with triangles: The vertices of the bounding volume (e.g. axis aligned bounding boxes) are transformed into ray-footprint coordinates. We obtain a 2-dimensional polygon and we must determine whether it intersects a circle around the origin, representing the drop-off radius of the Gaussian filter.

However, this test is quite expensive (e.g. 8 transformations for bounding boxes). Thus, we use a cheaper test in the current implementation: Bounding spheres are used as bounding volumes in the hierarchy. The center of a bounding sphere is projected into the orthogonal coordinates of the filter (Equation 16: matrix U multiplied by the eigenvalues, see also Figure 69). The eigenvalues of the filter coordinates form an axis aligned rectangle around the center of the projected sphere. The support of the filter kernel is located within a circle around the origin with a diameter according to the cut-off radius. It is approximated by a bounding box, too. To test for intersection, the two bounding boxes are tested for overlap. For the performance of the algorithm,

it is important to do the intersection test with elliptical ray cones (no circular approximation). Otherwise, queries with highly anisotropic ray cones become expensive. The simple test accounts for this using the main axis transformation.

6.2.6 Compositing

After we have found all intersections of a ray cone, three tasks remain: First, we have to identify intersection events belonging to the same surface. Typically, multiple intersections are detected for one surface, which are caused by multiple sample points. Second, the different properties detected at the intersection events have to be combined, which is essentially a resampling step. Third, after recursively computing color and shading of the reconstructed surface fragments, we have to combine the colors of the individual fragments to obtain the resulting color of the ray.

We deal with all three topics using a per-ray A-buffer approach, similar to that proposed by [Zwicker et al. 2001a]. Each detected intersection is inserted into a single pixel a-buffer that stores surface fragments sorted by depth. To decide whether a new intersection belongs to a surface fragment that already has been found, we assign a depth interval to each sample point, proportional to the sample spacing in object space. If the depth interval does not overlap with the depth interval of a surface fragment in the buffer, a new surface fragment is created by just inserting the intersection into the buffer. We store the attributes at the intersection point together with the depth, the filter weight and the depth interval of the intersection. If the depth interval of an intersection overlaps with one or more depth intervals already present in the buffer, the fragments are merged [Zwicker 2002b] (Figure 70).

Merging the fragments corresponds to a reconstruction of the surface attributes at the point of intersection with the center of the ray volume. We do this by adding together the attributes of the points multiplied with their weight. After all fragments are processed, the attribute values are divided by the weight sum for normalization. This works well for surface attributes such as color, normal, transparency or shading parameters. However, special care must be taken with differential properties: The matrices with the two derivative vectors of the normal in u_p and v_p direction cannot be averaged directly because the reference directions (u_p, v_p) vary with each point. Therefore, a coordinate system transformation from the coordinate system of the point to a common coordinate system must be performed. A cheaper alternative is to neglect aliasing of the surface derivatives and to use nearest neighbor sampling for these properties, as aliasing artifacts in normal derivatives are often not visible (this is done in the current implementation).

After all intersections have been found and the corresponding surface fragments have been reconstructed, we compute colors using a local illumination model according to the reconstructed parameters. Additionally, secondary rays are computed and traced recursively, if necessary. Then a compositing step is performed to determine the final color of the ray. The simplest compositing technique is alpha-blending [Zwicker et al. 2001a]: The point hierarchy is constructed to guarantee that any fragment in the interior of any surface has at least a weight of one. Therefore, we can interpret weight sums of less than one as alpha values to blend edges. We track the alpha values (i.e. min(*weight*, 1)) already during the intersection calculation process together with the maximum depth value encountered. If the ray is fully opaque, we do not descend into nodes of the hierarchy that are farer away than the maximum depth value (Algorithm 6, condition "`if not occluded(r, c)`"). This leads to an early ray termination, avoiding inspecting occluded regions of the scene.

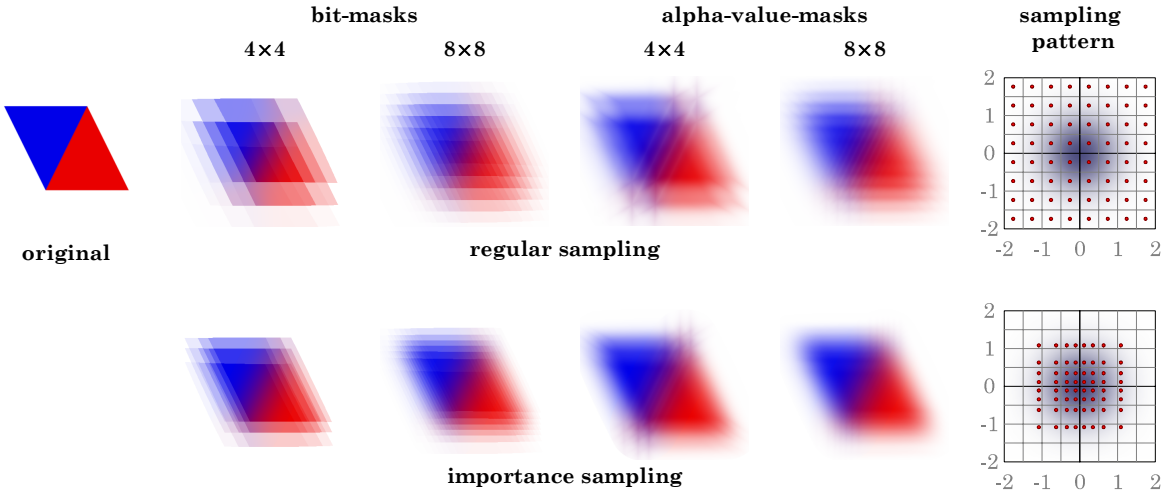


Figure 71: Comparison of different subpixel mask implementation and sampling densities (the test scene is the same as in Figure 72). Importance sampling with a subpixel alpha-blending heuristic strongly improves the rendering quality.

6.2.7 Subpixel-Masks

Compositing using alpha-values derived from weight sums is conceptually simple and computationally efficient. However, it has an important drawback: Typically, the extends of the objects are strongly overestimated at the borders of objects on the screen. As shown in the previous chapters, point-based representations come with unavoidable oversampling. Thus, the weight sum has to be much larger than one in most areas of the image in order to guarantee safe coverage anywhere within a closed surface. This leads to artifacts such as thickened borders or varying border opacity in regions of varying primitive density (for example in an area where two large triangles meet, see Figure 72). Those effects are especially unfavorable for rays with extends larger than a single pixel (e.g. for depth-of-field or shadow rays for extended light sources) where the false reconstructions cover a portion of the screen larger than a single pixel.

The main problem is that a single alpha value cannot express correctly the occlusion properties of a piece of geometry. It describes only the percentage of coverage, not the spatial location. Figure 73 illustrates the problem: It shows two fragments within a ray volume with 50% opacity. Depending on the overlap of the fragments, we can obtain between 50% and 100% overall opacity. In general, we can obtain any opacity ranging from the maximum single alpha value up to the sum of all alpha values (clamped to 100%). For a conservative estimate, we have to use the latter value, although, in the worst case, the composite opacity could be much smaller.

To solve the problem, we need information about the spatial location of the occluding surface fragments. A standard solution is the usage of subpixel-masks [Carpenter 84]: Subpixel-masks are bitmaps of k^2 bits in which set bits mark occlusions. We use a modified implementation for our purposes. First, we need a coordinate system to parameterize the bitmap. We use the (orthogonal) main axes of the current ray coordinate system multiplied by the main axis diameters (*ES* in Figure 67) as coordinates of the subpixel bitmaps: The center of the ray is located in the middle and the grid of k^2 bits covers the support of the unit Gaussian (which results after the coordinate transformation). It is possible that this coordinate system rotates around the center of the ray when the shape of the filter kernel varies. To our experience, this is no problem; the occlusion effects are still captured adequately.

(a) test scene: two triangles | (b) compositing using alpha blending | (b) compositing with subpixel masks

Figure 72: The compositing quality is strongly improved by employing subpixel masks to represent fragment occlusion. The test scene consists of two triangles that are rendered with rays with large ray footprint.

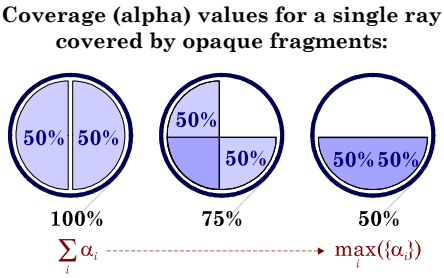


Figure 73: Compositing with alpha values is inherently ambiguous: the composite coverage can vary between the sum of all alpha values (conservative estimate) and the maximum single value ("worst case").

To compute the coverage of the subpixel masks, we perform a rasterization of the corresponding primitive: In case of triangles, we rasterize the triangles to the subpixel grid. For point primitives, we rasterize a tangential ellipse [Zwicker et al. 2001a] to enhance the resolution at borders (i.e. in areas where the ellipse becomes parallel to the view direction). As our data structure guarantees a d-sampling in *object space* (with d being the radius of the unprojected circle around the sample point in object space), we still ensure safe coverage of all surfaces.

To improve the approximation in terms of antialiasing, we use subpixel-masks in combination with alpha-blending: In contrast to the approach of [Carpenter 84], we store alpha values in *each* entry of the subpixel masks[29]. For each subpixel, we compute an alpha value according to our weight sum heuristic, employing a smaller filter kernel corresponding to the spacing of the subpixels (the unit Gaussian is scaled by the grid spacing). If we merge two fragments, we now add the corresponding alpha values of the subpixels, clamping the results to one (being a conservative estimate). This is also done to compute the composite occlusion of multiple fragments along a ray. To determine the visibility of a fragment, we subtract all alpha values of the (composite) occluding fragment from the occluded fragment (clamping to zero). Then the remaining subpixel masks is evaluated by multiplying all entries by the weight of the unit Gaussian filter in ray coordinates and dividing by the maximum possible sum for normalization. For compositing, the resulting value between zero and one is multiplied with the color of the fragment and the result is added to the color of the ray.

The efficiency of subpixel masks can be improved further by employing importance sampling: Instead of using a regular grid, we use a grid spacing adapted to the weights of the Gaussian filter. We use subpixel sample points with a spacing according to the integral of the filter function in between: Each interval between two adjacent sample points should have the same integral weight. For a one-dimensional filter e^{-x^2}, we compute the inverse of the distribution function $invDist(x)$ being the solution to

$$\frac{2}{\sqrt{\pi}} \int_0^y e^{-t^2} dt - x = 0$$

solved for $y \in [0, 1]$. Then, we take a one-dimensional regular grid of n values $x_i = (i+0.5)/n$, $i = -n \ldots n-1$ and place the sample points at $\text{sign}(x_i) \cdot invDist(|x_i|)$. For the two-dimensional sample

[29] Alpha-subpixel masks are too expensive for use with a complete framebuffer, as this is the case in [Carpenter 84]. However, for a single ray, the additional memory overhead is no problem.

pattern, the same transformation is applied independently for the x- and the y-coordinates. The resulting sampling pattern is shown in Figure 71 (lower right).

Importance sampling in combination with the weight-based alpha-blending heuristic noticeably improves the rendering quality: Figure 71 shows a comparison. Using both improvements, we obtain results with very low aliasing and a good approximation of occlusion for 8×8 subpixel masks. Even for 4×4 masks, the results are also already acceptable (some aliasing remains because of the non-optimal filtering of inner regions of triangles; this could probably be improved by an enhanced triangle filtering algorithm). Without the improvements, the aliasing problems are significantly worse. In practice, we usually employ 4×4 subpixel masks for performance reasons.

6.2.8 Adaptive Resolution

To control the resolution of the point cloud used for surface reconstruction, the algorithm traverses the hierarchy downwards until the spacing of the sample points is no larger than the minimum ray footprint. To decide whether the sample spacing matches the ray footprint within a bounding box of a hierarchy node, we have to determine the minimum eigenvalue of the ray footprint matrix at any parameter value inside the bounding box. The eigenvalues of the footprint matrix can be determined analytically by computing the roots of the characteristic polynomial of the matrix in Equation 14. Then, we could compute the derivative and solve numerically to determine the local minima. However, this procedure is involved and would considerably slow down the hierarchy traversal. This is not necessary; in practice it is sufficient to calculate the minimum eigenvalue just once (e.g. for the center of the bounding box) to control the resolution of the point cloud approximation: In practice, we can expect long thin rays with a footprint varying only slowly in respect to their current diameter. Only a few rays (e.g. reflected at a strongly curved object) might not fulfill this assumption. However, in this case, determining their color according to a footprint-sized point representation is already a rough approximation so that some additional inaccuracies in determining the ray footprint are of minor importance. Additionally, we will use octree nodes with only very few points per box for performance reasons (such as based on a 2^3 quantization grid, see Section 7.2.2.6). Hence, the size of the boxes is small in comparison to the ray diameter at their center.

Hierarchy traversal is controlled by the minimum eigenvalue of the footprint matrix in order to account for anisotropic ray intersections (such as a ray reflected from a cylinder). This approach can lead to performance problems. It might happen that the aspect ratio of a ray is strongly distorted so that a lot of points (with a spacing of the smaller main axis) have to be used to cover the ray footprint. Even if only a few such rays occur during rendering, they can consume a considerable amount of rendering time. Therefore, we have to limit the degree of anisotropy: Analogous to footprint assembly for texture mapping [Schilling et al. 96], we limit the number of sample points by limiting the anisotropy of the rays: After computing the eigenvalues, we check whether the minimum value is smaller than a constant fraction (typically 1/4) of the maximum. If this is the case, the minimum value is just set to that lower bound and the computation is continued using the modified filter kernel.

Another problem of the multi-resolution approach occurs at the transitions of different resolutions: Borders become visible between two adjacent resolution levels. As proposed by [Pfister et al. 2000] for the case of forward mapping, we use linear interpolation between two adjacent resolution levels to remove these artifacts. The interpolation is done by traversing the hierarchy one step deeper after the matching resolution is found and then linearly blending together the result.

The blending weights are given by the distance of the two point spacings of the two hierarchy levels to the required value. Blending is performed automatically by the fragment compositing step; it suffices to compute the corresponding weights, multiply them by the weight values of the filter kernel and insert the result into the ray a-buffer.

6.2.9 Implementation Notes

All intersection tests need an eigenvalue decomposition of the 2×2 ray footprint matrix (Equation 16), which is quite an expensive operation. The decomposition is needed to limit the spectral radii of the footprint and for the bounding sphere intersection tests as described above. Some steps of the algorithm like the triangle intersection test could theoretically be performed using the original footprint matrices. However, especially for the triangle test, this leads to wrong results in some cases. This is due to numerical stability issues: If the two footprint coordinates are nearly colinear, the projection into footprint coordinates and evaluation of the edge-ray distances is numerically unstable. The eigenspace transformation allows us to perform the test using orthogonal coordinates, avoiding the stability problems.

For this reason, we perform all intersection tests in orthogonal ray coordinates. In order to reduce the number of expensive eigenvalue decompositions, one can perform the decomposition only once per octree box and use the result for all intersection tests in the box (and to decide on further hierarchy traversal). For most test scenes, no visible difference was observed in comparison to the exact version of the algorithm that performs the eigenspace transformation for every vertex using the exact footprint matrix at that point. This optimization yields a speedup factor of 2-3.

Another issue is intersections of the ray with the originating surface: Unlike infinitesimally thin rays used in conventional raytracing, extended ray volumes cause more problems with intersection calculations: For secondary rays, it can easily happen that intersections with the surface the ray originates from are detected. This yields false occlusions and in the worst case terminates a ray before it enters the scene. Typically, the problem is worse for rays that are emitted at a small angle to the surface. In order to circumvent these problems, we need a criterion to detect such false intersections. We use a simple heuristic: Within a “no-intersection” zone around the point of intersection, intersections are ignored. The zone is a sphere with a radius proportional to the point spacing (minimum ray footprint) at the point of intersection. In addition, we also check the surface normals at intersection points. If the normals deviate by more than a constant angle from the normal at the intersection point, the intersection is accepted as we can assume that it detects a different surface. Another possible criterion to avoid self-intersections is to exclude all intersection points in a half-space defined by the tangent plane of the intersection point.

Chapter 7
Implementation and Results

In this chapter, we evaluate the proposed algorithms and data structures empirically. The chapter is divided into three parts: First, we describe the implementation our evaluation is based on. In the second section, the behavior of the algorithms in dependence on the different rendering parameters is examined. This includes parameters for the construction of the data structures (such as the number of sample points per octree node) as well as complexity parameters (such as the number of triangles in the scene). In the third part (Sections 7.3, 7.4, 7.5), we apply the proposed rendering algorithms and data structures to several benchmark scenes that could be typical applications for our techniques, comparing image quality and performance.

7.1 Implementation

We have implemented a prototype rendering system as test bed for our algorithms. The system has been implemented in C++ and tested on a conventional PC system. In this Section, we describe the most important aspects of the implementation. It is divided into three subsections: We start with a discussion of the underlying software architecture. Then we discuss some implementation decisions for algorithms and data structures that are specific to our implementation and have not yet been discussed in the general description. The last subsection discusses some technical aspects of the employed platform and consequences for our algorithms (such as issues with hardware accelerated rendering).

7.1.1 Software Architecture

The design of a suitable software architecture a crucial step for every non-trivial software project. In research projects, we have to deal with special constraints. Typically, we do not know the demands and characteristics of novel data structures and algorithms in advance. Thus, modularity and flexibility are especially important. The software framework must allow an easy integration of different approaches to a problem. An additional issue is evaluation and reproducibility: Unlike applications designed for end-users, a research application should allow the user to inspect low-level properties of the implementation at runtime and modify parameters and constants efficiently. The performed tests and measurements must be reproducible: We should be able to record all settings that have been made interactively and reload the same settings at any time. In

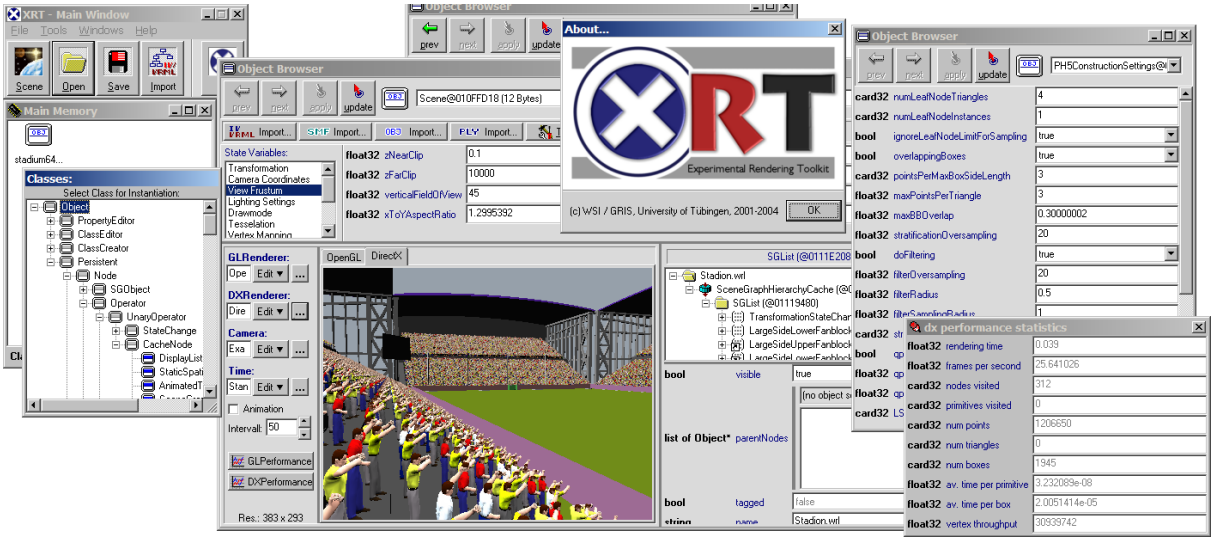


Figure 74: Screenshot of the XRT rendering system. The image shows automatically generated class selectors (left), automatically generated object editors (right), and a custom editor (scene view editor, middle), which itself consists of manual and automatically generated sub editors (e.g. the automatically generated hierarchy view, middle right).

order to fulfill these requirements, a layered system design has been devised that provides infrastructure at different levels of generality and abstraction. Our rendering system is coined XRT (eXperimental Rendering Toolkit, see Figure 74). It consists of three main layers. The first is the *foundation layer*, providing object handling, inspection and serialization. The second layer is the *scene graph library*, which has been designed to model and render highly complex scenes. Third, a *point-based rendering layer* is used that contains several libraries that implement several point-based rendering techniques. We will now discuss briefly the different layers of the system.

7.1.1.1 The Foundation Layer

An important design goal was to provide an interactive application that allows for interactive construction, rendering and evaluation. Thus, we need a basic infrastructure for inspecting data structures and triggering actions, i.e. applying algorithms. Traditionally, this is done by manually designing custom user interfaces for each task a user should be able to perform with the system. However, this causes a large implementation effort that is not practicable for a steadily changing research system. In order to simplify the construction of interactive applications, we have employed techniques from rapid application development [Borland 95, Sun 97]. The key idea is the use of structural reflection [Demers and Malenfant 95]: The system is implemented using object-oriented techniques, i.e. it is a collection of classes. All data structures are instances (objects) of these classes and the algorithms are methods of the classes. Structural reflection means to provide the structural information about the system design to the program at runtime: The program is able to query information about all classes, their inheritance relationship, the member variables, and to invoke methods dynamically. This technique is a standard technique that is provided by the run time system of many modern programming languages such as Small Talk or Java. In these languages, meta-classes are used to describe the structure of classes. A meta-class is itself an instance of a class that has been designed for describing classes (this technique of reflecting structural elements of a program is referred to as *reification* [Demers and Malenfant 95]). The meta-class provides lists of class properties and methods, which are objects that describe the corresponding class components and allow access to these components.

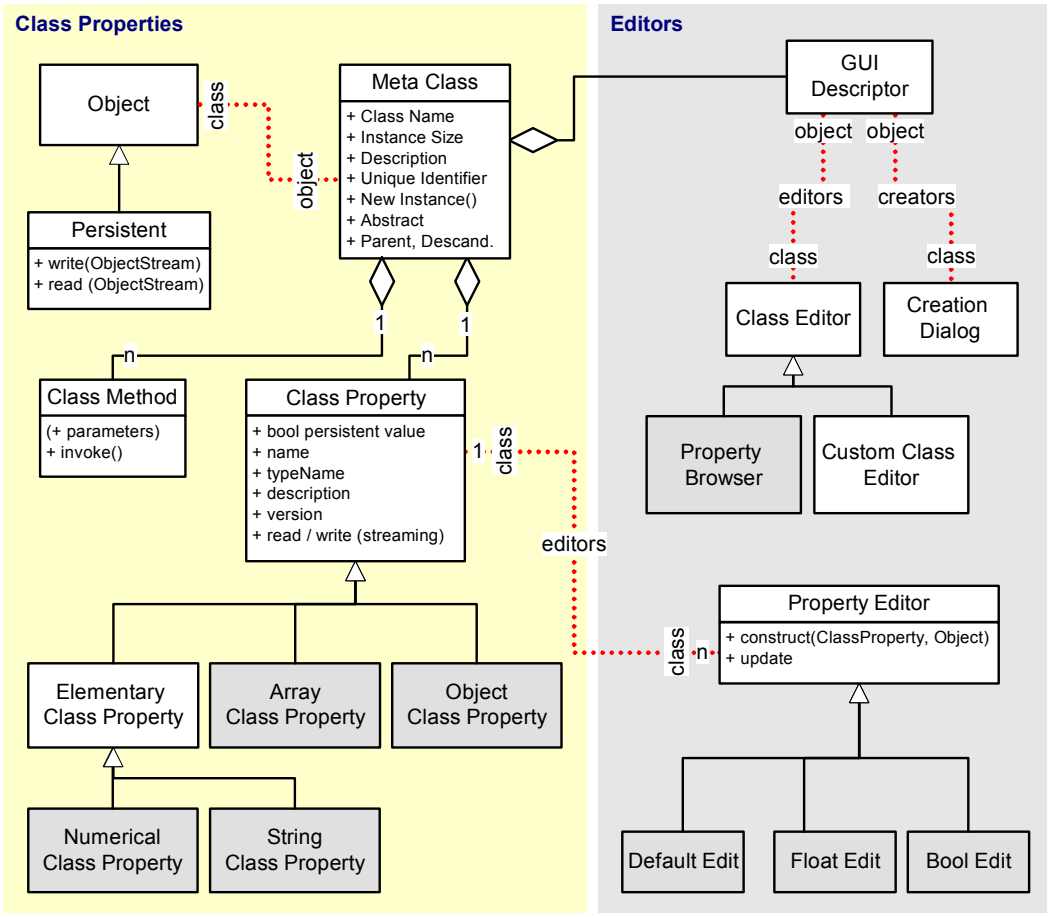


Figure 75: Schematic class diagram for the foundation layer. All classes are derived from "Object". Instances of "Meta Class" (or descendants) are used to describe the objects by specifying methods and properties. For each piece of meta-information (class, property), an editor class is provided that allows interactive editing. The base editor ("Property Browser") is constructed automatically from the meta-information.

Unfortunately, C++ does not provide a full structural reflection mechanism. Thus, the first task was to design a library for handling such meta-information (Figure 75): All classes in the system (except simple value types such as vectors) are derived from a common base class "`Object`". This class is associated with an instance of "`MetaClass`" or a descendant of this class. The meta-class describes the structure of the class type such as properties (member variables) and methods. This description is extendible. It is for example possible to define member variables directly or via access methods (`getX()` / `setX()`, similar to JavaBeans [Sun 97]).

Each structural element is associated with a set of editors (one being the default) that allow interactive editing. A standard editor is provided for each case: The standard class instance editor creates itself by examining the properties of the class of an instance and creating editors of all properties [Borland 95, Sun 97]. In addition to this standard class editor, other generic editors are provided such as a hierarchy view (see Figure 74) that displays hierarchies of objects of certain base classes in a tree view widget. For class properties, the standard property editor displays a text box with a string describing the value of the property. The standard editors do not work in all cases. For example, an automatic editor is not useful for editing images or sound files. In these cases, a custom editor has to be implemented. However, in many cases, this is not necessary (at least for a prototype application) and thus saves a lot of work. In addition to class editors and

property editors, we can also declare a construction dialog that assists in the interactive construction of new class instances. This is useful for classes that cannot be initialized with meaningful default properties.

The structural reflection mechanism also allows for automatic serialization of objects, similar to the mechanism provided by the Java runtime library [Arnold and Gosling 96] or the Delphi interface designer framework [Borland 95]: Serializable objects are derived from the base class "`Persistent`", which is a direct descendant of "`Object`". This class contains predefined `read()` and `write()` methods that create a serialized byte stream that can be used for storage on hard disc or network transmission. The predefined methods inspect the properties of the class and store them in linear order automatically. A version number is used to allow for incremental versioning: All properties are written into the stream. If a stream of older version is to read again, all properties that have not been defined at the time of writing are replaced by default values. This allows for adding properties incrementally, but not for deleting properties. In this case, as well as in cases were automatic serialization is not efficient (e.g. if compressed image storage is demanded), a manual implementation of `read()` and/or `write()` is needed. Again, this is only needed for a small fraction of the classes, thus saving much implementation effort.

In addition to the reflection system, the foundation layer also contains other basic infrastructure libraries such as vector algebra and geometric primitives.

7.1.1.2 The Scene Graph Library

The second layer on top of the foundation layer is a library for handling large scenes. As discussed in the introduction, we use a scene graph based modeling approach [Rohlf and Helman 94, Wernecke 94, Foley et al. 96]. Two main goals have been motivating for the design of this part of the system: First, we should be able to model highly complex scenes using hierarchical instantiation. The system should provide sufficient flexibility and extensibility for integration a variety of different modeling techniques. Second, we must be able to integrate data structures necessary for output-sensitive rendering seamlessly into the system.

A scene object consists of two components (Figure 76): a directed acyclic graph of scene graph nodes and a scene graph state. The scene graph state is a general collection of state variables. Some of them (local attributes) can be changed by nodes of the scene graph during traversal (such as the current transformation). Others are global parameters, such as the camera settings. The scene graph state can easily be extended by declaring additional types of state variables. Similarly, new types of objects (describing geometry) and operators (grouping, instantiating and modifying objects or state variables) can be added by implementing additional subclasses.

The scene graph is purely descriptive, it does not provide rendering techniques by itself. This means that we do not attach "`renderMe()`" methods to each node, as often done in other scene graph libraries. Instead, any scene graph node reports a description of the subgraph it represents. This is done using a two step approach: First, each node provides a method to describe its subgraph by reporting a set of sub scene graph nodes, each associated with a current state. In addition, every (reported) node that describes a concrete object is also able to provide a list of geometric primitives (given the current scene graph state). This scheme is very general: A list operator for example describes itself by just delegating the description query to all nodes contained in the list. A replication object, that creates e.g. multiple instances along a path, calls the report method of the replicated object multiple times but with different scene graph state for each instance. A very general modeling operator, such as a Boolean operation, could determine a triangle mesh of the subgraph and create a completely new (virtual) scene graph node that is presented if a description of its subgraph is demanded.

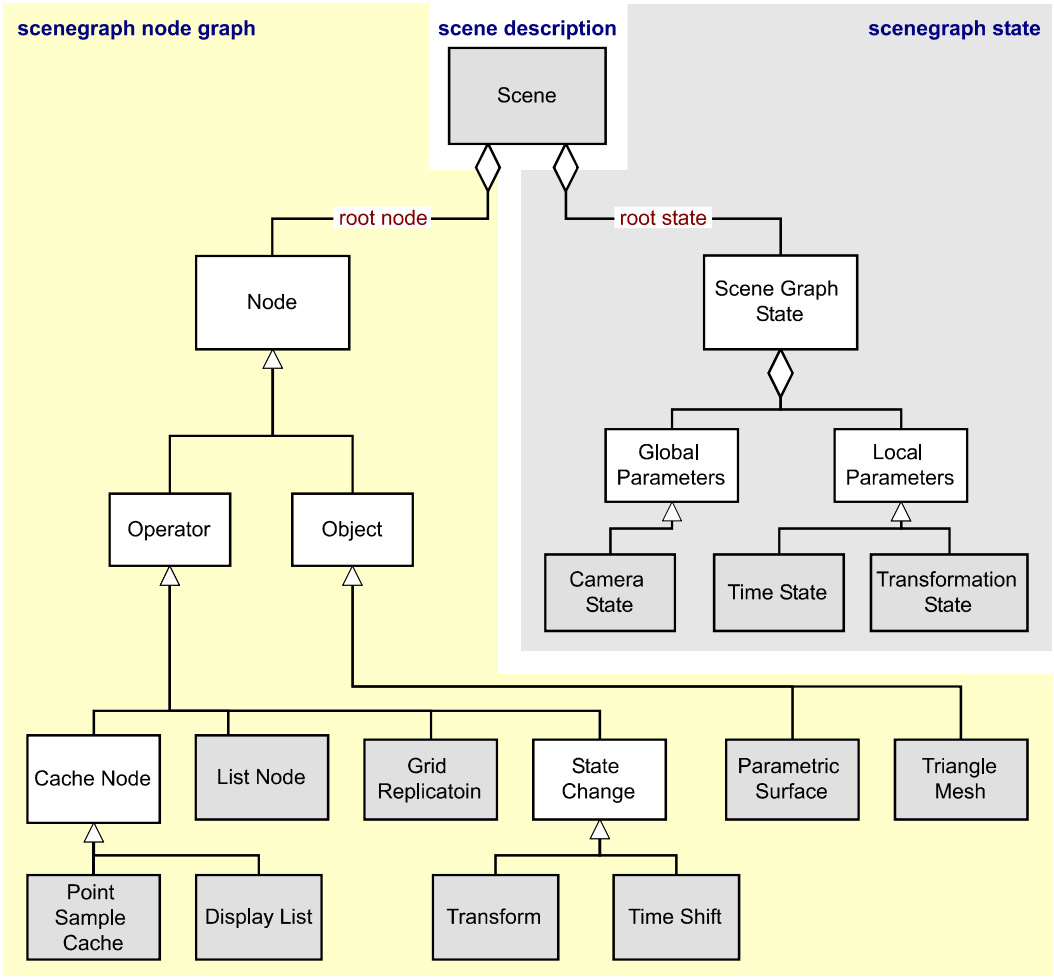


Figure 76: Architecture of the scene graph library in a rough sketch. A scene consists of a state and an acyclic directed graph of nodes. Nodes can specify geometry, change the current state or group and instantiate other nodes. Precomputed data structures for accelerated rendering are stored in "Cache Nodes". Each cache node represents the subgraph of all nodes below, excluding those nodes stored in further cache nodes of the same type.

To incorporate data structures for point sampling and other accelerated rendering techniques, we define a cache node: This scene graph node comprises precomputed information. A corresponding rendering technique can detect the cache node during hierarchy traversal and stop the traversal at that node to employ the precomputed information rather than the original geometry.

To allow for instantiation of cache nodes, we place multiple cache nodes into the hierarchy. Each cache node performs a scene graph traversal of its subgraph (i.e. reporting all nodes below it). If cache nodes of the same types are encountered during this traversal, it is not continued below those nodes. Instead, the changes to the scene graph state are recorded (the state object comprises a local state difference stack that records all state changes since the beginning of the traversal). The state change information together with a pointer to the instantiated cache are recorded and stored as instance object in the according data structure to be precomputed. Please note that this mechanism requires a descriptive rather than a self-rendering ("`renderMe()`") scene graph design.

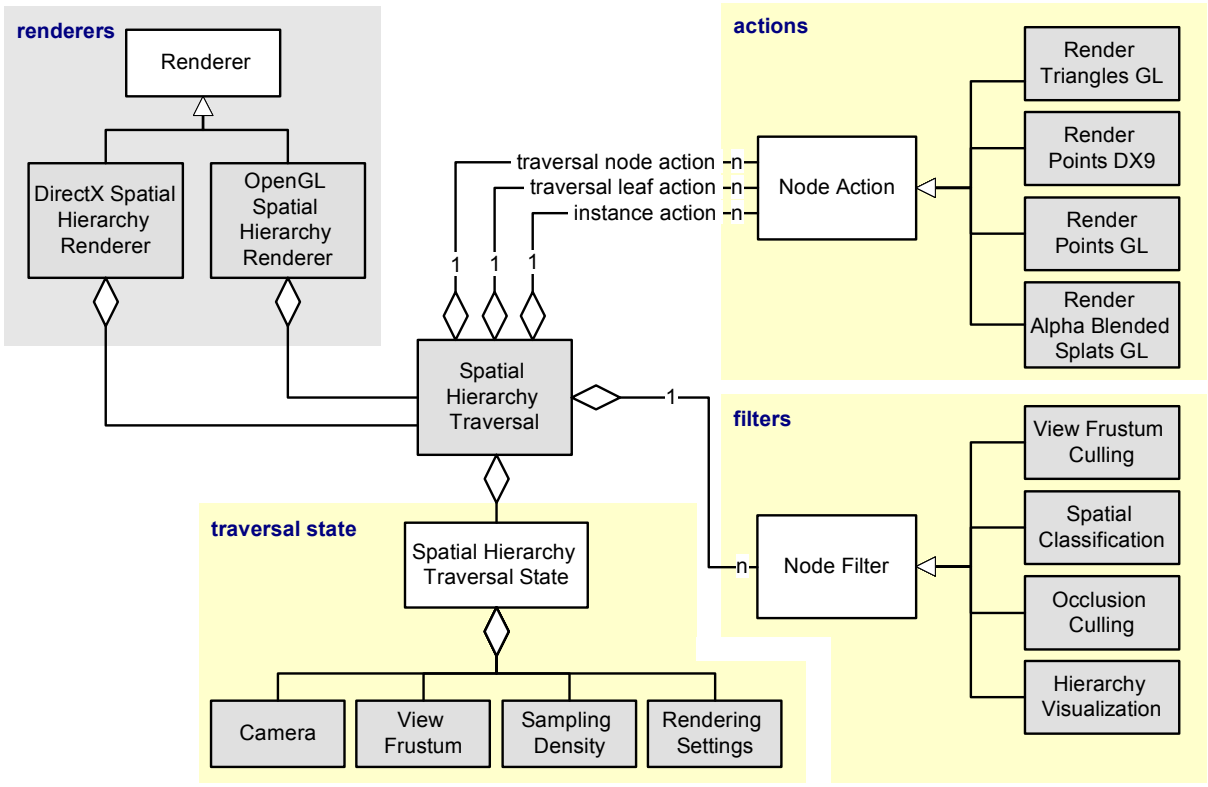


Figure 77: General rendering architecture for displaying spatial hierarchies. A collection of filter and display agents controls the hierarchy traversal, based on a current traversal state. The hierarchy nodes are accessed via a general, abstract interface. A concrete rendering strategy is specified by plugging in a collection of filters/actions. Different rendering strategies can be implemented while reusing most of the code.

7.1.1.3 Point-Based Rendering

Using the scene graph library, we can easily integrate point-based rendering by providing corresponding cache and rendering classes[30]. The cache node collects geometry and instances as described before. The rendering object traverses the scene graph, looking for point-sample caches. If such a cache is found, the traversal of the subgraph is canceled and the data structure is evaluated to render the point-based representation. Additionally, we have also implemented a meta-renderer that searches in depth-first manner for cache nodes and automatically selects a rendering strategy associated with each cache. This allows us to combine multiple rendering techniques in one scene graph, such as e.g. display lists for coarse geometry and point-based caches for highly detailed objects. This is important in practice where we usually need to combine multiple rendering techniques for different parts of the scene in order to obtain a good performance.

Different point-based rendering algorithms (nested sampling, animated sampling, hierarchies with prefiltering for raytracing) have been implemented as different cache objects. Despite the separate implementation, we were able to reuse much of the underlying code. An interesting example is the rendering module for forward mapping, which is used with minimal change for all data structures. The design of the rendering module has been inspired by the design of the Jupiter scene graph library [Bartz et al. 2001]. In this library, a set of “agents” is used that control the

[30] Only static sampling has been implemented within the XRT system. The dynamic sampling technique, being the oldest variant, has been implemented in a different program, but using the same compiler and mostly the same base libraries (for math, graphics). Thus, the resulting performance measurements are (roughly) comparable.

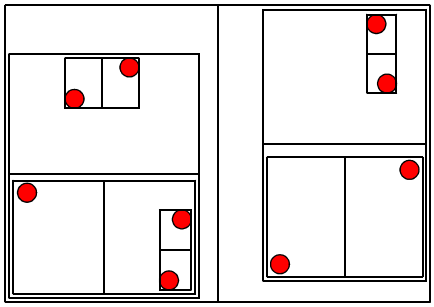


Figure 78: Example of a fair-split tree.

traversal of the scene graph. We use a similar paradigm for traversing the spatial hierarchy: A general traversal algorithm processes a spatial hierarchy, accessing the nodes of the hierarchy via a general interface. The general hierarchy node interface allows navigation in the hierarchy and access to point and triangle primitives as well as access to instances. (This interface class has to be reimplemented for each separate data structure.) To control the traversal, two different kinds of "agents" are called: Filters prune the hierarchy based on information about the current node (bounding box etc.) and the current traversal state (camera settings etc.). View frustum culling and point sampling density adaptation is performed using filter agents. Action nodes trigger render actions such as drawing the current nodes triangles or splatting point primitives.

The main advantage of this architecture is that most of the code can be reused for different tasks and rendering problems. In order to support both OpenGL- and DirectX-based rendering, we only need to rewrite the render action agents. To perform a simple hierarchy visualization, we just replace the triangle and point rendering nodes with bounding box rendering nodes. Occlusion culling can be added by providing an additional filter node. The drawback of the general rendering approach is its moderate performance. Due to the many virtual function calls, the general implementation is not as efficient as a highly optimized implementation.

7.1.2 Implementation of Algorithms and Data Structures

After describing the software architecture in a rough sketch, we now discuss some subtleties of the implementation of the proposed algorithms that have not been treated in detail in the previous chapters:

The implementation of the dynamic sampling data structure differs slightly from the scheme proposed in Section 4.1: Instead of an octree, we use a fair-split tree [Callahan 95] as base data structure. A fair-split tree is a binary tree. It is constructed by starting with a bounding box of the scene. Then, subsequently, the box is split into two equally sized halfs in the dimension of the largest box side length by an axis aligned cutting plane[31]. After each split, the two new bounding boxes of the two resulting primitive set are computed ("shrink-to-fit"-step). The scheme is carried on recursively until a minimum number of triangles are contained in a node, which then becomes a leaf node. Figure 78 shows an example. For the modified data structure, we still apply the same rules for dealing with triangles exceeding the bounding box of a node. We just substitute the maximum side length for the (symmetric) octree box side length.

[31] [Callahan 95] gives a slightly more general definition that is needed for an efficient parallel construction of the data structure.

A fair-split tree leads to a similar hierarchy as an octree if the volume of the root bounding box is fully covered with geometry. The only difference in this case is that the 1:8 split of an octree is replaced by three subsequent binary splits in the three spatial dimensions. Consequently, the theoretical upper bounds for the running time derived in Chapter 5 still hold for the modified data structure as it behaves like an octree in the worst case. If the scene is not volume-filling, a fair-split tree yields potentially a better performance, as it is able to adapt to the distribution of the primitives more tightly. In practice, we need about half the number of tree nodes to approximate the depth-factor up to the same accuracy for a typical input scene than with a conventional octree. For static sampling, we have used conventional octrees as we need symmetric cubes as nodal boxes in order to define the sample spacing. Similarly, the variant of the dynamic sampling data structure that allows for dynamic updates of its content (`insert()`, `delete()`) also uses an octree because updates of an octree are much easier to implement than updates of a fair split tree [Callahan 95].

Note that a fair-split tree does not need shortcuts to achieve guaranteed linear memory consumption as it never performs useless splits. It can also be computed in $O(n \log n)$ time, using a similar procedure as described in 4.1.3.2 for octrees (see [Callahan 95] for details). However, our implementation uses the simple, worst case quadratic construction procedure in both cases (octrees and fair-split trees). For the octrees, the implementation of short-cuts also has been omitted. These two improvements are only necessary from a theoretical point of view. As discussed in Section 4.1.3.2, they are usually not important for computer graphics applications. Problems could only occur if scenes of extremely varying scale were processed, which are rarely encountered in practice.

7.1.3 Technical Aspects

In this section, we discuss some aspects of our implementation related to technical aspects of the employed hardware. This comprises a description of the reference platform used for all benchmarks and in how far platform specific optimization techniques have been employed.

7.1.3.1 Software and Hardware Platform

The XRT system (as well as the older dynamic sampling system) has been implemented in C++.

Fragment processing speed [10^6 pixels/sec]	OpenGL
visible triangles	640 MPix/s
occluded triangles	2,000 MPix/s

Table 3: Measured fragment processing performance of our reference platform (nVidia GeForce FX Go5650)

Vertex processing speed [10^6 vertices/sec]	OpenGL (intermediate mode)	OpenGL (display Lists)	OpenGL (std. vertex arrays)	DirectX (managed vertex buffers)
triangle vertices / points	7.2 MVert/s	8.6 MVert/s	21.5 MVert/s	28.5 MVert/s
animated tri.-vert. / points	4.5 Mvert/s	7.1 MVert/s	15.6 MVert/s	32.7 MVert/s

Table 4: Vertex processing performance of our reference platform (nVidia GeForce FX Go5650, driver version 6.14.10.4581) for different APIs. In the "animated" case, a vertex program has been used that interpolates between two vertices (position and color) linearly. The OpenGL implementation for rendering animated geometry uses the NV_VERTEX_PROGRAM extension [SGI 2004].

The source code has been compiled using Borland C++ Builder 6.0. All benchmarks have been performed on a Dell Inspiron 8600 Notebook PC running Windows XP Professional. The system had been equipped with a 1.5 GHz Pentium-M processor, 1 GB of RAM and an nVidia GeForce FX 5650 graphics processor. The overall performance of this PC system corresponds to a mid-range system at the time of writing (both in terms of processor and graphics performance).

As already expressed during the exposition of the software architecture, the main goal of our implementation was flexibility and modularity, not maximum performance. The code has not been optimized strictly. Arithmetic for geometric computations is mostly done using simple standard techniques; no assembly code, cache optimizations or low-level hardware extensions were exploited. Therefore, our measurements can only give a rough overview of the possible performance and allow a relative comparison of different techniques. In terms of absolute values, a hand tuned implementation would probably yield some performance improvements in many cases. Nevertheless, some details of the hardware characteristics should be considered for a meaningful interpretation of the results. This is especially important for an implementation of forward mapping techniques using graphics hardware:

7.1.3.2 Hardware Accelerated Forward Mapping

Rendering using forward mapping consists of two main steps in terms of running time: Hierarchy traversal and rendering of the collected primitives. The first step is always performed in software. For static sampling data structures (and caching in dynamic sampling), the second step can be implemented using programmable graphics hardware, which is usually more efficient than even an optimized software implementation. The traversal step typically has to deal with only a small number of data structure nodes (usually some thousand in contrast to many million primitives per frame). Thus, the costs for hierarchy traversal are usually dominated by the point projection costs. Unfortunately, modern graphics hardware imposes some subtle constraints on the primitive processing process for achieving a high performance. Thus, we first have to take a brief look at the architecture of current PC-based rendering hardware.

Current programmable graphics hardware (as contained in our reference PC platform) provides a coprocessor (GPU, graphics processing unit) to accelerate typical rendering tasks[32]. In most cases, the graphics coprocessor resides on a separate graphics board. It contains separate local video memory that is connected to the GPU with a high bandwidth interface. The graphics subsystem communicates with the CPU via the accelerated graphics port (AGP). The bandwidth of this interconnect is usually an order of magnitude smaller than the bandwidth of the local graphics memory.

Current graphics processors are programmable according to a programming model specifically targeted at forward mapping rendering purposes. Rendering is organized as a pipeline with three separate stages: The first stage is a programmable vertex processor. It can perform arithmetic calculations on each vertex of a primitive (point, triangle). A vertex consists of a set of attributes that are transformed by a vertex program. A vertex program is a batch script of a few arithmetic and vector instructions. At this stage, the perspective transformation is done and additional tasks such as blending between different vertex sets of a keyframe animation can be performed. The second step of the pipeline is the rasterizer, which generates all pixels within a primitive (triangle, line, point splat). This step is not programmable but it is only possible to choose from a fixed set of primitives. The third step is a pixel/fragment program. Again, a small

[32] We provide only a rough description to highlight the main implementation concerns. A more detailed description of current programmable graphics hardware can be found for example on the web pages of the respective hardware vendors [ATI 2004, nVidia 2004a].

batch script can be uploaded to the graphics hardware that performs arithmetic calculations for each pixel that is written into the framebuffer. The script can use interpolated attributes, global constants, and access texture maps to compute the color of the pixels.

The throughput of the different stages varies. We can identify roughly two different values: the throughput of the vertex processing stage and the throughput of the fragment processing stage (rasterization and pixel shading). The throughput depends of course strongly on the employed vertex and pixel programs. The vertex processing rate is additionally dependent on the number of attributes that are input for every vertex. For all current architectures, the processing speed of the pixel pipeline segment is larger than the vertex processing speed by at least an order of magnitude (in terms of instructions per second). For our reference platform we have measured up to 30 million vertices per second (transformation only, no lighting, position and color for each vertex) and 640 million fragments (2,000 million in the case of occlusion[33]) per second. See Table 3 / Table 4 for details. For point-based rendering, we have to process rapidly large sets of vertices. Thus, a good utilization of the vertex processing capabilities is crucial for a high rendering performance.

An important problem of both vertex and fragment processing is bandwidth: If the graphics board processes 30 million vertices per second (3×32Bit float values for the position, 4 bytes color = 16 Bytes), this already corresponds to a bandwidth of 480 megabytes per second. Thus, an optimal performance can only be achieved if the vertices are stored in local video memory of the graphics board or at least in "AGP-memory", i.e. in a memory region that can be accessed by the GPU via fast DMA (direct memory access) operations. Thus, the location and transfer mode for geometry has a strong effect on the obtained rendering performance. We have tested four different alternatives: The OpenGL programming interface offers a variety of transfer modes. We have tested the intermediate mode (one function call for every vertex attribute), display lists (precompiled batches of intermediate mode calls) and standard vertex arrays (non AGP-memory blocks that are copied to the graphics board for every rendering call. In addition, we have also used managed DirectX vertex buffers that automatically determine the most efficient memory location. In either case, we have measured the vertex throughput by sending unconnected triangles or point primitives down the graphics pipeline (which both lead to similar values). For optimized triangle mesh rendering, topological optimizations concerning the rendering order (triangle strips, indexed primitives, optimized index order [Hoppe 99]) have to be considered for optimal results. As this is of minor importance for point-based rendering, we have not examined this topic further. If our implementation has to render triangles, it always just uses lists of single triangles without optimized topological order.

The results of a synthetic benchmark are summarized in Table 3. Usually, the DirectX implementation yields the best performance while the OpenGL implementation is slower. This is due to problems with the standard interface. For example, the vertices stored in standard vertex arrays have to be copied to the graphics memory during each rendering path. This is necessary to provide the correct rendering semantic as defined by the interface. Intermediate mode rendering is always quite slow by design and display lists are not always fully optimized in current drivers. The performance deficits of OpenGL can be compensated by employing vendor extensions for a more efficient memory management (such as NV_VERTEX_RANGE, EXT_VERTEX_ARRAY_OBJECT [SGI 2004]). These optimizations are often platform-dependent so that we prefer to use DirectX for optimized rendering. Nevertheless, it is possible to obtain similar rendering speeds with both interfaces if all options for optimization are employed.

---

[33] Due to z-compression and early hierarchical rejection techniques, the processing rate is different for visible and occluded fragments. See [nVidia 2004a] for details.

A further problem is due to latencies: Every call to the graphics API for specifying a new set of primitives to be processed causes a fixed amount of costs. Using DirectX vertex buffers, it is not possible to send more than about 100-200 thousand rendering commands ("batches") to the graphics processor per second on a typical PC platform, even if they contain only a small number of primitives [Wloka and Huddy 2003]. OpenGL is usually faster than DirectX for small batch sizes (about 2 times, depending on the vertex transfer mode) but still shows the similar problems [Wloka and Huddy 2003]. As a consequence, we have to design the rendering system for processing large batches. For the multi-resolution algorithms this means that we will prefer using rather large numbers of sample points in each hierarchy node as every node corresponds to one (or two, if triangles are present) batches.

7.1.3.3 Raytracing

The point-based multi-resolution raytracing technique has been implemented in software only. We have not employed special low-level optimizations but only straightforward C++ code. Therefore, the performance is not comparable with highly optimized raytracing packages such as the technique proposed by [Wald et al. 2001], which uses involved cache, coherency, and SIMD optimizations for a significant performance advantage.

To allow at least a rough performance characterization, we have implemented a conventional triangle-based raytracer using the same code basis (for linear algebra operations, intersection calculations, hierarchy construction etc.). To evaluate the performance, we compare our implementation with this reference implementation. This does not allow a precise prediction of the performance of a highly optimized implementation but can at least provide some qualitative hints.

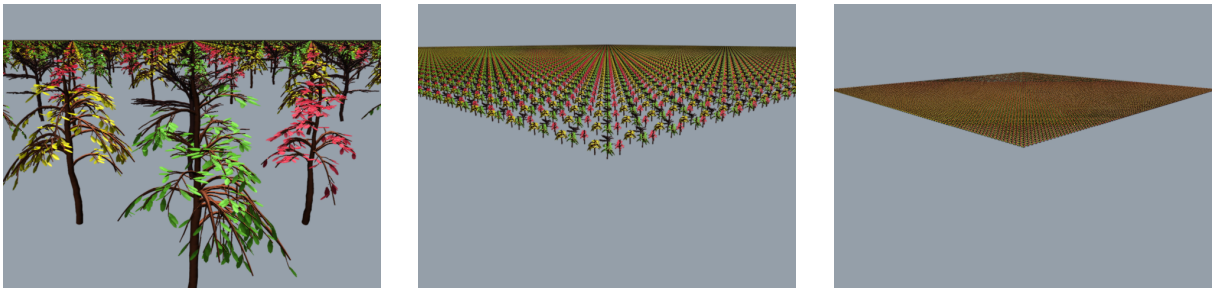
7.2 Preprocessing and Rendering Parameters

In this section, we examine the influence of different parameters on rendering time, memory demands and image quality of our rendering techniques. Our goal is to identify the complexity characteristics and to find suitable trade-offs for practical applications. The section is divided into two parts, dynamic and static sampling, because these two approaches show a different behavior concerning algorithmic parameters. In addition to the diagrams shown in this section, all measured running times and parameters are also given numerically in Appendix A, Table 12 - Table 27.

7.2.1 Dynamic Sampling

The dynamic sampling data structure provides several parameters that can be tuned for optimal performance: The most important is the accuracy of the spatial adaptivity ε. A good choice of this parameter is crucial for optimal performance. In addition, we can also decide to employ orientation classes to improve the sampling adaptivity. In this case, we are looking for the number of classes that delivers the best performance. In addition, we can also vary the settings for area classes that are used to identify large triangles that are not included in sampling. Beside these algorithmic parameters, we are also interested in the dependence of the rendering complexity on the scene complexity. We will examine this behavior in this section, too.

In our measurements, we use an example scene consisting of a forest of trees. The forest consists of instances of four prototype trees that are replicated 10,000 times (using two layers of



(a) near view **(b) medium distance** **(c) far view**

Figure 79: Test scene for examining the influence of the accuracy of the spatial adaptivity on the rendering time

10×10 instances for hierarchical instantiation). The four trees consist of 93,180 triangles, leading to an overall scene complexity of 931,800,000 triangles. We measure the rendering time for a camera path with strongly varying distance to the scene (see Figure 79). This leads to a varying number of hierarchy nodes to be traversed at the same approximation accuracy. This permits examining the influence of this parameter on the choice of the approximation accuracy. The forest scene has been chosen because it is a typical input to our rendering algorithms. Additionally, it provides a distribution of geometry and orientations that is typical for a large class of scenes such as landscapes or extended urban scenery. We will examine other scenes in Section 7.3.

7.2.1.1 Preprocessing

The dynamic sampling data structure requires only few parameters to control the preprocessing process itself (such as number of triangles per leaf node, maximum node overlap). For realistic values, these parameters have only little effect on the running time. Thus, we do not examine the influence of these parameters in detail but just use standard values (≤ 64 triangles and ≤ 1 instance per leaf node, max. overlap factor 2) for the forthcoming experiments. The resulting preprocessing time is short (in comparison to static sampling, as described later on): Building the distribution tree for our benchmark scene (Figure 79) took 8 seconds and required 22.8 MB of main memory. Please note, that the implementation has been optimized for neither efficient preprocessing nor memory demands.

7.2.1.2 Spatial Adaptivity

The most important rendering parameter is the accuracy of the spatial adaptivity: Our rendering algorithms use a spatial hierarchy to extract point sets with a sampling density according to a given importance function. In the case of forward mapping using dynamic sampling, the importance function is proportional to the depth factor $1/z^2$ of the corresponding surface fragments. The approximation accuracy is explicitly given as a factor $(1 + \varepsilon)$. The rendering algorithm collects boxes from the spatial hierarchy in which the depth factor of the given projection does not vary by more than $(1 + \varepsilon)$. Triangles that are too large to be classified correctly (i.e. exceeding the tolerance zones of the corresponding hierarchy nodes) are reported as is and rendered using triangle rasterization. The same is also done for triangles that are so large that they would obtain several sample points if rendered with point-based rendering.

Choosing an optimal value of ε is a trade-off: A good approximation leads to increased traversal costs while a bad approximation leads to more sample points (and thus also to higher rendering costs, too, due to the conservative nature of the approximation approach). Usually, we

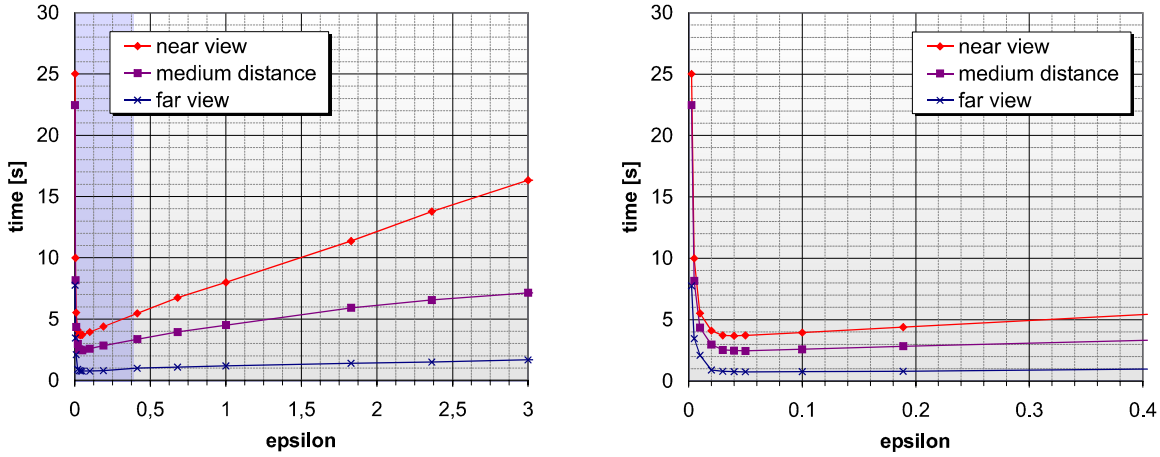


Figure 80: Rendering time for the scene depicted in Figure 79 in dependence of the approximation accuracy ε. The right diagram shows a close-up of the optimal region (marked in blue in the left diagram).

expect an optimal running time for medium values. The location of the optimum depends on the rendering technique: The larger the processing costs for a sample point, the more we expect to profit from a good approximation accuracy.

We have rendered our test scene for three different viewing positions, as depicted in Figure 79. The resulting timings are shown in Figure 80 as a plot of the running time (for the three viewing points) in dependence of the approximation parameter ε. As expected, we obtain one distinct optimum. For the example scene, the optimum value is found in the range of $\varepsilon \approx 0.03 \ldots 0.04$. This value is the same for all three viewing settings. This observation can be explained by a simple model: If we assign costs for sampling (proportional to the oversampling factor) and for box processing (triangle rendering, hierarchy traversal) to each box, we would expect the best rendering times when the two costs are balanced. The viewpoint independence is only given for one scene and only as long as these average costs do not change. If the either the average sampling costs or the per-node processing costs change, the point for the best trade-off shifts according to this costs. To show this effect, we have repeated the measurements for different sampling costs per box (Figure 83): In the experiment, we have varied the sampling density for one and the same scene. The result is a strongly varying value ε at which the minimum of the rendering time is reached. However, we still obtain a distinctive optimum. The same effect can be expected if the average surface average area of the geometry in the hierarchy nodes changes. This value depends on the scene. Additionally, a similar effect can be expected if the viewer moves within one and the same scene and turns his field of view towards a part of the scene with increased projected area.

In order to examine the trade-off between node and sample processing costs more in detail, we have also measured the number of objects (hierarchy nodes, points, triangles) that are processed by the algorithm (Figure 81). It turns out that three effects are responsible for the overall rendering time: First, the number of hierarchy nodes that have to be processed increases with decreasing ε. Conversely, the number of sample points increases with growing ε. Third, the number of triangles also increases with decreasing ε. The last effect is caused by the strategy for handling of triangles that exceed the bounding box of the spatial hierarchy: The smaller the selected boxes are, the more triangles do not fit into the boxes and are reported “as is”. Therefore, the number of triangles increases strongly for very small values of ε. The latter effect is only observed for closer views, in which large triangles are seen in the projected image. For the far view of the same scene (Figure 82), no sample points are replaced by triangles for the measured values of ε

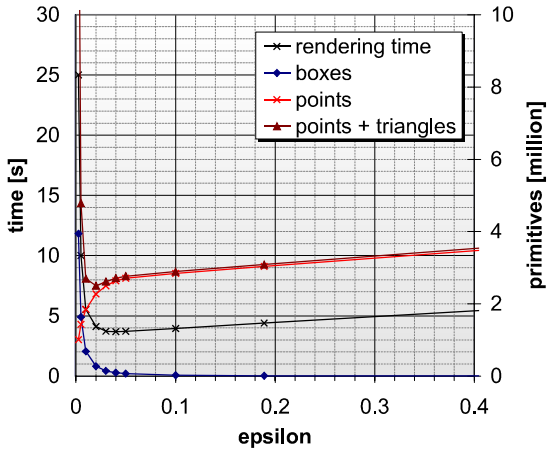


Figure 81: The number of processed primitives for varying approximation accuracies ε (near view). For small ε, processing the boxes and box-exceeding triangles dominates the rendering costs, for large ε, the costs are dominated by processing sample points.

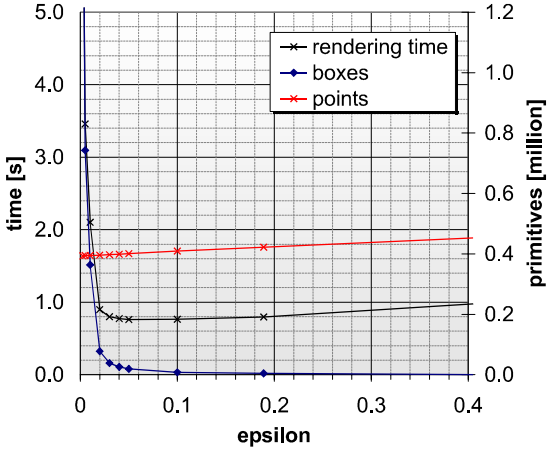


Figure 82: Number of processed primitives for varying approximation accuracies ε (far view). In the far view, no triangles are used (in contrast to the near view). Even for small values of ε (i.e. deep subdivision) no triangles are introduced.

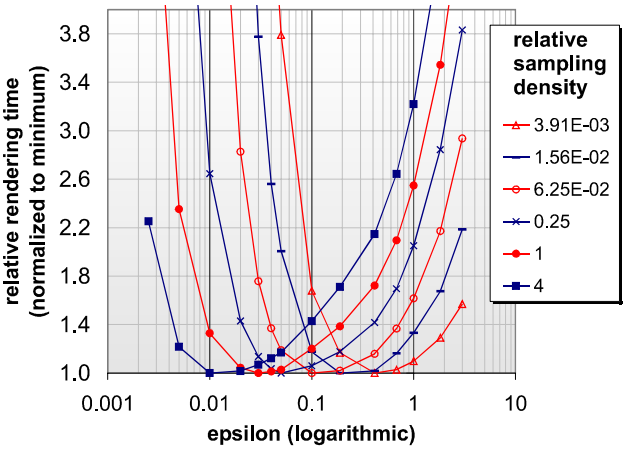


Figure 83: Trade-off between the rendering cost components for varying ε and different sampling densities. The larger the number of sample points per box, the smaller is the optimal approximation accuracy ε.

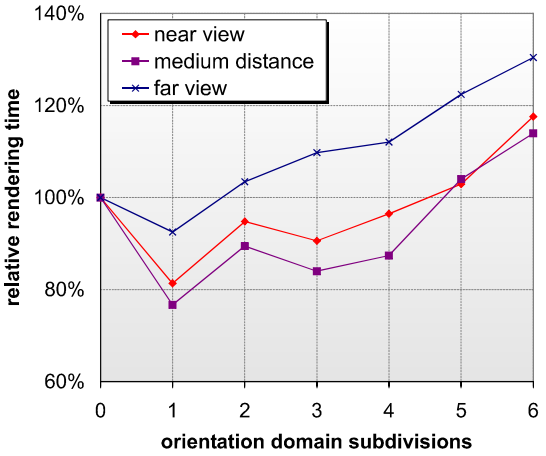


Figure 84: Effect of orientation classes on the rendering performance. The x-axis shows the number of binary subdivisions of the domain of normal directions in polar coordinates. The y-axis shows the relative rendering time.

because hierarchy nodes are much larger (in absolute units) in this case. Nevertheless, a similar increase of the running time is observed due to the increased number of hierarchy nodes to be processed.

The increased number of processed bounding boxes (and in certain cases additionally the increased number of processed triangles) leads to a strong increase of the rendering time for very small values of ε. The approximation inaccuracies lead to an increase of the rendering time for larger ε; thus, we obtain one minimum for the sum of the two monotonic cost functions. Overall, it is not possible to determine one absolute optimum valid for all scenes and viewing conditions. Actually, we have to determine the optimal value dynamically, according to the current sampling and node processing costs. For our experiments, we have determined ε manually. However, we could imagine a fully automatic scheme that performs a numerical optimization algorithm on

measured rendering times in order to adapt the parameter dynamically during the rendering of an animation. As we have always observed a distinct optimum, a stable automatic adaptation scheme could probably be implemented easily.

7.2.1.3 Orientational Adaptivity for Dynamic Sampling

In addition to the depth, we can also consider the relative orientation in respect to the current viewpoint in order to determine the sampling density. For random sampling, we should scale the probability of receiving sample points proportionally to the cosine of the angle between view vector and triangle normal[34]. To determine the angle, we have the option to build classes of triangles with similar orientation, as detailed in Section 4.1.4.

We have measured the running time for our example scene with different numbers of orientation classes using the dynamic (random) sampling technique. To create the orientation classes, we have extended the construction of the spatial hierarchy (the fair-split tree). Instead of starting with spatial subdivision, we first perform d_n splits in the orientational domain: We compute the normal directions of the triangles and express them in polar coordinates. The resulting two-dimensional domain is then subsequently split in its largest extend, similar to the spatial subdivision and the resulting sets of triangles are assigned accordingly to the child nodes. After the maximum orientational subdivision depth is reached, the subdivision is continued in the spatial domain, as usual.

Figure 84 shows the results for our test scenes. The x-axis shows the number of binary subdivisions and the y-axis shows the resulting relative performance for the three views of our test scene. The performance has been determined by rendering using the same set of different ε-values as in the previous examples and picking the minimum of the resulting rendering times. Then, the optimal value is divided by the value for using no orientation classes[35].

The resulting curves show a minimum rendering time for a split depth of 1, i.e. for two orientation classes. For a larger number of orientation classes, the benefits of reducing the sample size are (over-)compensated by the requirement of much more spatial boxes to be processed. Even in the optimal case ($d_n = 1$), we only obtain moderate savings (up to about 20%). The measurements have been performed without using area classes to identify large triangles. In combination with area classes, we expect less benefit from orientational classification. As the identification of large triangles is more important and the potential savings of orientation classes are small, we will not use orientation classes in remaining rendering examples.

7.2.1.4 Large Triangles

For the identification of large triangles, we use classes of triangles with similar area and build a spatial data structure for each class. Two parameters control the efficiency of this scheme: the area class spacing and the point of transition between sampling and triangle rasterization. The area class spacing is the factor by which two triangles should differ to be placed into different area classes. The choice of this factor has only a minor influence on the rendering time (see Figure 85). Within a range of sensible values of 1.5-64 we obtain similar rendering times (again, we have optimized the ε parameter separately for each area class spacing). The observed small

---

[34] For stratified sampling, we would need view angle dependent stratification patterns. Thus, we do not consider it here. We have only implemented and tested orientation classes for dynamic sampling, as we could expect the largest benefits in this case (in which sampling is most expensive).

[35] This experiment requires the measurement of a large number of rendering times for different parameters. In order to reduce the overall measurement time, we have used only half the resolution (0.25× the sampling density) as in the previous examples.

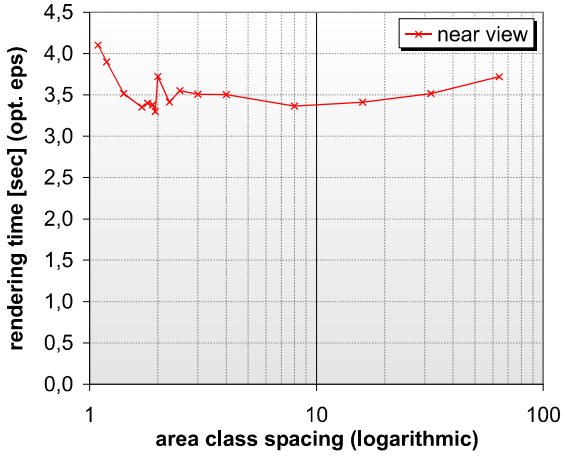


Figure 85: Choice of the area class spacing factor. The dependence of the running time on this parameter is rather weak within a wide range of values.

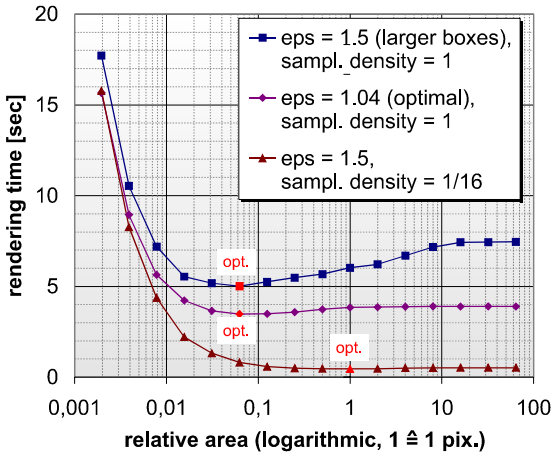


Figure 86: Transition between point and triangle rendering. The optimum depends on the on-screen sampling density. We obtain an optimum at about 1/16 of the on-screen density.

deviations are rather unpredictable. This is probably due to the discrete nature of the classification and subdivision process. For very small factors, the number of boxes to be extracted from the multiple spatial hierarchies grows and the running time increases, too. For all subsequent measurements, we have used an area class spacing factor of 8, which leads to satisfactory results.

A more critical choice is the transition value at which we switch between triangle rendering and point-based rendering: We have rendered the near view of our test scene (Figure 79) using different transition values. First, we have employed the optimal ε-value of 0.04, yielding the purple curve in Figure 86. The x-axis shows the area value of transition: A value of 1 corresponds to the unprojected size of a single pixel (i.e. the area of a pixel reprojected into object space). For small values, i.e. more triangle-based rendering, the rendering time grows strongly due to the high scene complexity. For large values, the rendering time converges towards the time needed for rendering with sampling only, which is a bit longer. For an optimum ε value, the second effect cannot be seen clearly because at this ε value a lot of geometry is already rendered by triangle rasterization due to the deep subdivision of the spatial hierarchy. The effect becomes more obvious for larger values of ε: The blue curve shows the same experiment for $\varepsilon = 1.5$, at which all geometry can be potentially rendered using sample points. For our implementation, the optimum transition point is at about 1/16 of an unprojected pixel (for a sample spacing of 1 pixel). It is reasonable to conjecture that this value is not an absolute value but rather a fixed fraction of the on-screen sampling density. To verify this, we have repeated the measurements using 1/16 of the previous sampling density, which indeed led to a transition point of about 1.

7.2.1.5 Scene Complexity

The running time of the dynamic sampling data structure depends weakly on the scene complexity[36]. Theoretically, we expect a logarithmic dependence on the number of triangles in the scene. To verify this experimentally, we have made benchmarks using a synthetic test scene: The scene consists of a flat chessboard of $k \times k$ squares, which are subdivided recursively in order to increase the scene complexity without affecting other parameters. Hierarchical instantiation with an in-

[36] In contrast to static sampling: Static sampling uses precomputed sample sets so that the underlying number of triangles does not matter.

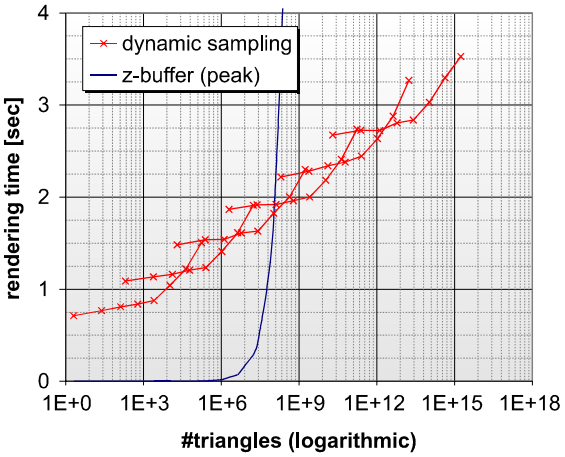


Figure 87: Runtime complexity of the dynamic sampling algorithm. The running time is logarithmic while conventional rasterization is linear. Please note the logarithmic scale of the x-axis (#triangles).

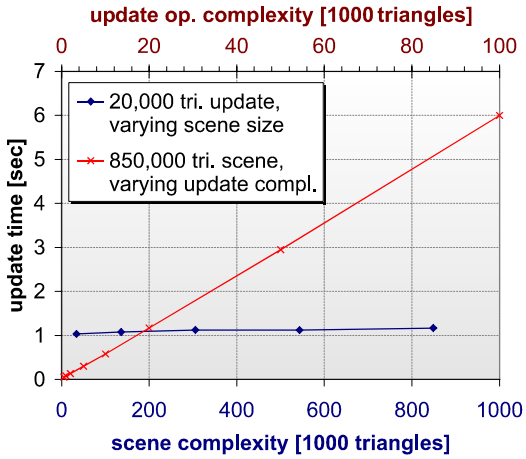


Figure 88: Dynamic updates for varying scene and update complexities. For typical update tasks (change some 1000 triangles in a 10-100 thousand triangle scene) we obtain roughly constant update time of about 50-60 µsec/triangle.

creasing number of instantiation levels is employed to overcome the limitations of main memory. Figure 87 shows the timing results for rendering the test scene consisting of $n = 2k^2$ triangles. The diagram shows a linear increase of the running time of the dynamic sampling algorithm for an exponentially increasing scene complexity, confirming a logarithmic time complexity. For comparison, we have also plotted the running time of a conventional z-buffer renderer without a multi-resolution approach. The blue curve shows the estimated running time assuming a peak triangle throughput of 60 million triangles per second (corresponding to 30 million vertices per second with optimal vertex caching). For small scene complexities, the simple rasterization approach is much faster than sampling. Please note that we have deactivated the rasterization of "large triangles" as the goal of this experiment was to measure the sampling performance. For larger numbers of triangles the running time increases linearly and soon exceeds the range of interactive rendering. For the most complex benchmark scene ($1.7 \cdot 10^{15}$ triangles), we would expect a (best case) rasterization time of about one year while the sampling based rendering processes the scene in 3.5 seconds.

The details of the rendering time are also interesting: The red curve describing the rendering time of the sampling renderer consist of several segments, corresponding to 0-5 levels of instantiation. Each additional instantiation level adds an additional (roughly constant) overhead to the processing costs of each sample point (400ms overall per instantiation layer). This corresponds to sampling the higher level distribution tree and processing the instantiation transformation. Within one instance, the curve consists of two roughly linear parts with different slope (in logarithmic scale). We assume that the first curve segment corresponds to triangle sets that fit into the second level cache, while the second curve segment with larger slope corresponds to main memory access. For the random sampling algorithm, the latency times of accessing a random memory address are an important issue determining the performance so that we obtain a speedup for data fitting into the cache. The data structure size (of the triangles within the leaf instance) at the end of the first part (subdivision level 5) is 480KB and 1850KB at the start of the second segment (subdivision level 6). This corresponds well to the second level cache size of 1MB provided by the Pentium-M processor of our test platform.

7.2.1.6 Dynamic Updates

We have also implemented the dynamic update algorithms that allow insertion and removal of objects into/from the hierarchy without rebuilding it from scratch. In order to quantify the performance, we have measured the time to insert and remove objects of different complexity into/from scenes of different complexity (see Figure 88). The test scenes were built without using instantiation. The dynamic updates always operate within one spatial hierarchy. Inserting or deleting a triangle requires the same operations (and thus similar costs) as changing an instance entry (which has not been implemented but the results would be similar). The test scene consisted of replications of a variant of the city model depicted in Figure 107. We did not use the tree test scene as the city model consists of fewer triangles, enabling a more fine granular control of the scene complexity. For the dynamic updates, we have inserted different variants of the Stanford Buddha model (Figure 102b, [Stanford 2004]) that have been simplified using QSlim [Garland and Heckbert 97] to obtain models of varying complexity. Figure 88 shows the resulting update times: The update time is mostly independent of the scene complexity and roughly linear in the number of changed triangles. We obtain average update costs of 50-60 µsec per triangle. The theoretical analysis predicts worst case costs of $O(n \log h)$ with n being the number of updated triangles and h being the height of the tree. In our test cases, we have examined scenes with 30-850 thousand triangles. Thus, the height of the octree (usually $h \in O(\log n)$) does not change strongly so that we obtain roughly constant costs per triangle. The performance is sufficient for typical interactive local modifications of the scene. Please note that the code has not been optimized for performance in any way. Thus, it is probably possible to improve the constants significantly if an application requires more efficient dynamic updates.

Providing dynamic update capabilities also affects the rendering time, as we need to implement a dynamic data structure for the distribution lists. We have implemented the variant of the dynamic data structure that uses the spatial tree itself as dynamic distribution lists (see Section 4.1.7). For our implementation, the time for choosing a sample point was increased by about 50% for the dynamic data structure (factor 1.41 for the city scene and 1.48 for the tree scene used in the other benchmarks). In addition, the version of the dynamic sampling data structure supporting dynamic update operations uses an octree rather than a fair split tree because dynamic update operations for octrees are much simpler to implement than for fair split trees (see [Callahan 95]). Using an octree instead of the fair-split tree led to an increase of the number of bounding boxes by a factor of about 2.2 (averaged for a camera path). Overall, the performance of the dynamic data structure is at least roughly comparable to the static version.

7.2.2 Static Sampling

In this section, we evaluate the effect of different preprocessing parameters on rendering and preprocessing costs of the static sampling technique. We examine the influence of the depth approximation accuracy ε, the prefiltering costs, and the transition to large triangles. Additionally, we will also study the dependency on preprocessing parameters of raytracing and animated point sampling. An additional, important parameter is the employed stratification technique. We will examine the different options more closely in Section 7.3.1.2. Up to then, we will always use greedy neighborhood-based point removal, which appears to be the best choice according to our simplified analytical model (see Section 4.2.3). The empirical results will later confirm this property.

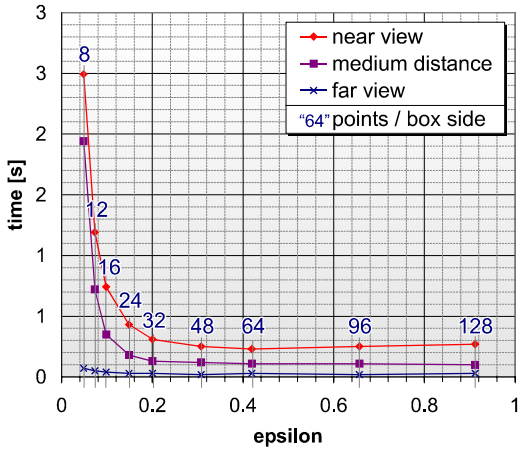


Figure 89: Rendering time (static sampling) for the scene depicted in Figure 79 in dependence of the approximation accuracy ε. For static sampling, much larger values of ε are favorable due to smaller point processing costs.

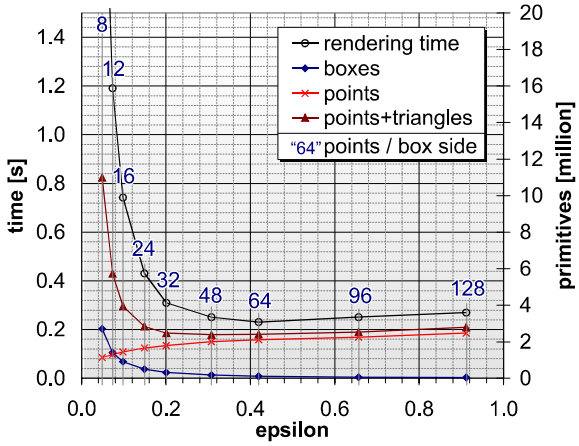


Figure 90: Number of processed primitives for varying approximation accuracies ε (scene: Figure 79, near view). Again, the number of sample points grows for larger ε while the number of hierarchy nodes and triangles grows for small ε.

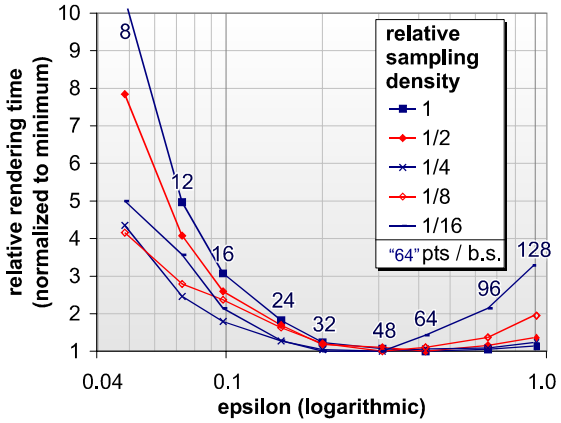


Figure 91: Trade-off between the rendering cost components for varying ε and different sampling densities. For static sampling, the optimal approximation accuracy is (roughly) the same for the different rendering resolutions (in contrast to dynamic sampling, see Figure 83)

7.2.2.1 Spatial Adaptivity

In this first subsection, we examine the influence of the spatial approximation accuracy on the running time for static sampling (cf. Section 4.2). Using static data structures, the approximation accuracy is fixed during preprocessing by specifying a relative sampling density for the hierarchy nodes (given as sample points per side length of an octree box). As it is not possible to change this preprocessing parameter during rendering, the determination of a suitable approximation accuracy is especially important.

Structurally, we expect roughly similar results for static sampling as for the dynamic sampling data structure. However, point and hierarchy node processing costs differ strongly from the dynamic case. Thus, we will expect a different ε-values for an optimal trade-off. We repeat the same measurements as for dynamic sampling, starting with a comparison of the rendering time

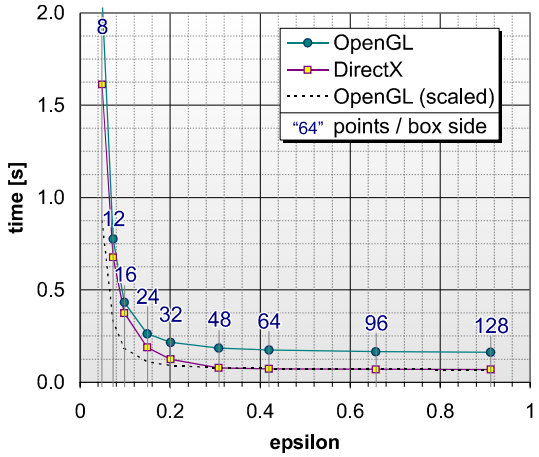


Figure 92: Average rendering time (camera path near to far) for different rendering APIs. The DirectX-based implementation achieves a higher throughput of primitives so that box processing costs become more dominant.

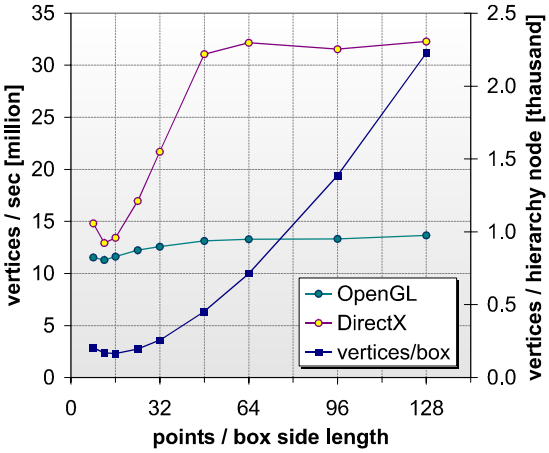


Figure 93: Comparison of the vertex throughputs of the two rendering implementations. The DirectX version is much more sensitive to the box size because of a higher overall throughput and larger per-batch time penalties.

for different sampling accuracies and different views of our test scene (Figure 79a-c). Figure 89 shows the results: For a small number of sampling points per hierarchy box, the rendering costs are quite high. For larger box sizes, the rendering time drops rapidly. An optimal rendering time is achieved for $k \geq 48..64$ points / box side length ($\varepsilon \geq 0.2...0.3$). For moderate box sizes of up to 128 points per box side length, the rendering time remains roughly constant.

The behavior is caused by the same mechanisms as in the dynamic case but shifted due to the reduced primitive processing costs: Figure 90 shows a plot of the different processing cost components (points, triangles, hierarchy nodes). For small box sizes, both the number of processed triangles and boxes increase strongly, leading to a sub-optimal rendering time. For larger boxes, the loss of approximation accuracy leads to an increased number of sample points to be processed. This effect is not as critical as in the dynamic case because of two facts: First, we have used a smaller maximum value of ε in our tests. This is a limitation of the implementation: It uses a simple three-dimensional array for stratification and filtering of sample points that does not allow processing very large numbers of sample points per box side length due to the memory demands[37]. Second, the processing costs for sample points are much lower due to the precomputed, stratified sample sets and hardware accelerated rendering. Thus, an increase of sample points is less critical.

Again, we should examine if the trade-off is stable for different rendering conditions (different sampling densities, different scenes). Figure 91 shows the relative rendering time for different sampling densities (i.e. different splat sizes at a fixed resolution of 640×480). The rendering time has been normalized by dividing by the minimal rendering time. The x-axis showing the ε-values has been scaled logarithmically to display the result more clearly. For static sampling, we always obtain an optimal rendering performance for $k = 48$ points per box size, independent of the sampling density. This has been expected as we have used a fixed data structure: In contrast to dynamic sampling, the number of points in each hierarchy node remains constant, only the traversal depth changes for varying sampling resolutions. This means that the ratio of box proc-

[37] The implementation could be improved by substituting hash tables for arrays (see Section 4.2.3). However, as the measured running times show, this is not necessary because it would not reduce the rendering time (at least for our test scenes and image resolutions).

essing costs and primitive processing costs remains fixed, keeping an optimal trade-off. Nevertheless, we still expect a shift of the optimal trade-off for different geometry. The more area we have in each octree box on the average, the smaller we can choose the number of points per box side length to balance box and primitive processing costs.

We can also observe that the increase of rendering time due to inaccurate approximation of the depth factor is worse for larger on-screen sample spacing. This is probably caused by the fact that fewer triangles but more sample points are used at coarser resolutions so that oversampling becomes more visible. Please note also that all curves in Figure 91 have been normalized to the minimum rendering time. Thus, the absolute rendering time is quite small for large sample spacings. Thus, although the relative increase is significant, the absolute increase is not as critical as suggested by the figure.

Trading-off box processing and point processing costs also depends on the employed rendering API: We have tested two variants: rendering with standard OpenGL 1.1 vertex arrays[38] (stored in client memory) and DirectX 9.0 managed vertex buffers (transferred automatically to local video memory). The second rendering technique achieves a higher primitive throughput (Table 4). Therefore, the box processing costs are more critical. In addition, the DirectX interface shows larger costs for submitting a single "batch" of primitives (points, triangles) to the graphics hardware than the OpenGL variant [Wloka and Huddy 2003]. Figure 92 shows a comparison of the rendering times for different approximation accuracies: Although the number of primitives (points and triangles) is the same for both implementations, the two plots are not proportional to each other (see dotted line). For the rendering technique with a higher primitive throughput, the hierarchy processing costs are more dominant. This shows that the increase of the rendering time is not only caused by the increase of triangles stored in inner nodes. This observation becomes more obvious in Figure 93: In this diagram, we have computed the vertex throughput ((*points* + 3×*triangles*) / *time*) for the different rendering settings. For the renderer that uses standard OpenGL 1.1 client side vertex arrays, the throughput remains roughly constant. The overhead due to box processing has only a minor effect. For the more efficient DirectX (managed vertex buffers) renderer, the choice of the box size is much more critical: It achieves an optimal throughput only for a box size of $k \geq 48$, corresponding to about 1000 vertices per box. This number is probably even larger for more powerful graphics hardware than our test system [Wloka and Huddy 2003].

Overall, we see that static sampling yields the best rendering performance for larger values of ε than dynamic sampling. This is mostly a result of faster point processing performance. For values of $k \geq 48$ points per box side length, we have observed satisfactory rendering times. In general, it is less critical to use hierarchy boxes with a bit too many points than using boxes that are too small as this affects the performance more severely. Especially for high-end graphics hardware, large boxes are mandatory in order to achieve an optimal utilization of the graphics hardware.

7.2.2.2 Preprocessing Costs for Different Approximation Accuracies, Handling Large Triangles

After examining the relevance of the depth approximation accuracy for the rendering time, we also have to consider its effect on the preprocessing costs. At first sight, this parameter does not seem to affect the preprocessing costs. However, as Figure 94 shows, this is not the case. Indeed, the preprocessing costs (time and memory demands) grow with increasing parameter k (number

[38] If not noted otherwise, all results in this section have been measured using the OpenGL renderer.

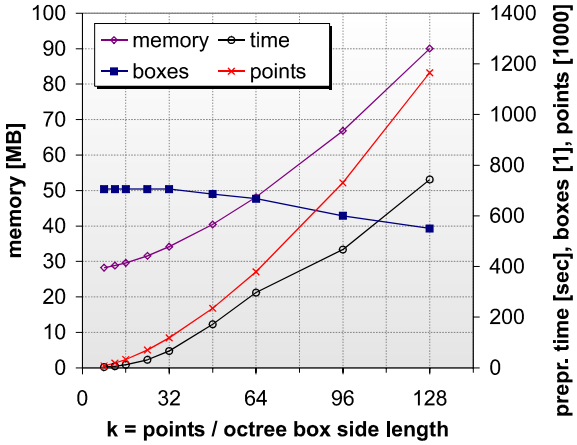


Figure 94: Preprocessing costs for varying approximation accuracies (static sampling), including the numbers of primitives the hierarchies consist of.

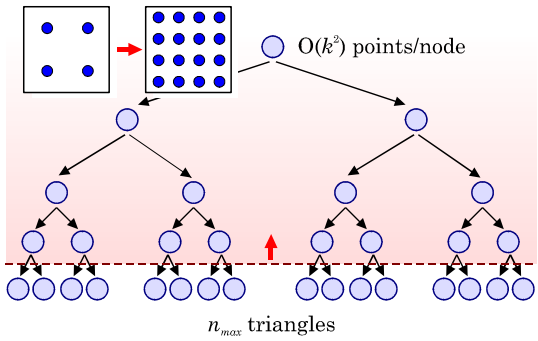


Figure 95: Increasing the maximum number of triangles in leaf nodes proportional to k^2 in order to keep the number of point in the hierarchy constant.

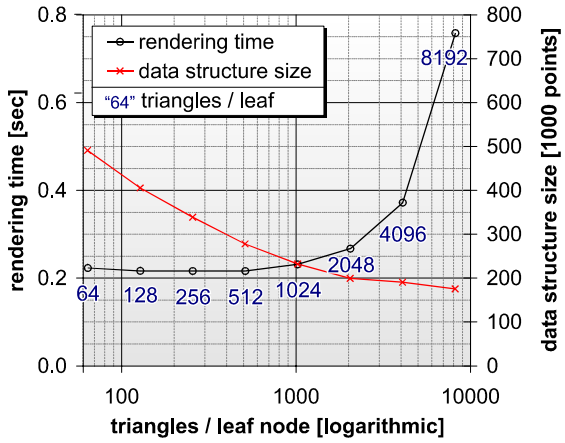


Figure 96: Rendering time and data structure size for a varying number of triangles per leaf node (for k = 48 pts / box side length). A good trade-off is to use as many triangle vertices per leaf node as points in the inner nodes on the average (here: about 1000).

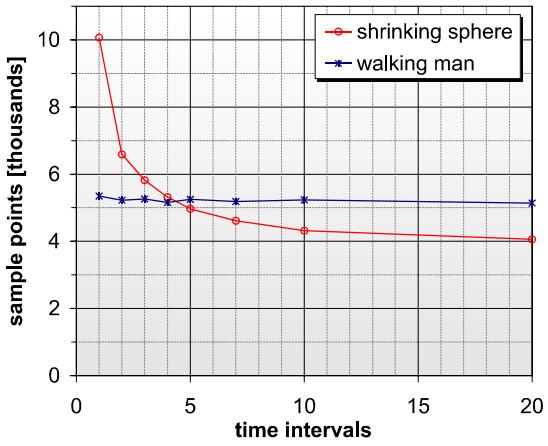


Figure 97: The influence of time discretization for preprocessing animated geometry. The diagram shows the average number of sample points for a complete time interval. The first model is a sphere, shrinking to a 1/10 of its original size; the second model is a model of a walking human.

of sample points per box side length). The reason for this behavior is easy to explain. Figure 94 shows that the number of points in the hierarchy also increases with increasing k but the number of hierarchy nodes (boxes) remains roughly constant. More specifically, the number of sample points in the hierarchy grows approximately quadratically with k. If we fix the hierarchy and only increase k, we expect that the number of sample points grow quadratically with k as we sample the same surfaces with more sample points per unit area.

The only effect that works against this is the termination criterion for the hierarchy: We stop sampling triangles in child nodes that have already received more than p_{max} (in our test case p_{max} = 1) points. In addition, we also terminate the recursive subdivision if the number of triangles in a node falls below n_{max} triangles (we have used constantly n_{max} = 1024 for the measurements). In practice, the second criterion (batching triangles in the leaf nodes) dominates as termination criterion. This means that the number of hierarchy boxes remains roughly constant

(as confirmed by Figure 94), leading to a quadratic growth of the number of points (and thus also of preprocessing costs in terms of time and memory) with the parameter k.

We have two options to fight this behavior: First, we can omit the parameter n_{max} so that a higher sampling density always leads to a hierarchy of reduced depth. However, this would lead to leaf nodes with few triangles, which is not optimal for efficient rendering using graphics hardware. Second, we can increase n_{max} proportionally to k^2. In our experiments, this technique kept roughly constant memory and preprocessing time demands and ensures to process geometry in large batches: As a rule of thumb, it makes sense to keep the average number of triangle vertices per leaf node in the same range as the number of points in inner nodes to balance the processing costs. Figure 96 gives empirical evidence on this: We have varied n_{max} for the preprocessed static sampling data structure of our forest benchmark scene and compared the rendering times (near view, Figure 79a). For large values of n_{max}, the rendering time increases because the adaptivity of the multi-resolution approach is reduced. For values of about 250-500 triangles (i.e. 750-1500 vertices) and below, we obtain the best rendering times. This corresponds to about 1000 sample points per inner node. For our test scene, a value of about $n_{max} = 500$ triangles per leaf node provides probably a good trade-off between preprocessing costs and rendering time.

The second parameter for controlling the handling of large triangles is the maximum number of sample points a single triangle may receive until it is stored "as is" in an inner node. As most triangles are excluded from sampling by the n_{max} leaf node criterion, this parameter has only a minor effect on the running time. For artificially small values of n_{max} ($n_{max} = 2$), we have determined the best rendering times for $p_{max} \geq 3$ points per triangle.

7.2.2.3 Intermediate Sampling Levels

A further option during the construction of static sampling hierarchies is the usage of intermediate sampling levels. Instead of binding the resolution steps to the octree levels, which causes sampling spacings to vary in powers of two, we can store multiple point sets with varying spacings in each node. This reduces the oversampling due to the discrete steps in the sampling density of the spatial hierarchy.

For a test scene consisting of replicated trees, we have obtained a reduction of the number of points for a rendered image by 24% for one additional point cloud in each node, corresponding to a resolution (sample spacing) stepping of $\sqrt{2}$. For two additional point sets in each node (resolution stepping $\sqrt[3]{2}$), we obtain a reduction of 30% relative to a simple octree. Using 5 point clouds (4+1, stepping $\sqrt[5]{2}$), we can reduce the number of rendered points by 46%. The rendering time decreases by the same factor. The memory requirements are increased by a factor of 1.3 (one additional point set), 1.6 (two additional point sets), and 2.1 (four additional sets), respectively. In practice, at least one layer of points might be a good choice, delivering some improvement in rendering efficiency at moderate storage costs.

7.2.2.4 Prefiltering

Another preprocessing parameter of static sampling is the oversampling for creating prefiltered sample points. This is a trade-off between noise artifacts and precomputation time. In order to find satisfactory settings, we have rendered a worst case test scene with different oversampling and compared the image quality. Our worst case scene is a large chess board consisting of black and white squares. We have positioned the camera so that multiple squares are represented by a single sample point. The resulting point cloud has then been rendered using the raytracing technique described in Chapter 6, applying effectively a Gaussian reconstruction filter (with renormalization by the sample weights) to the sample points. The results are shown in Figure 98. As

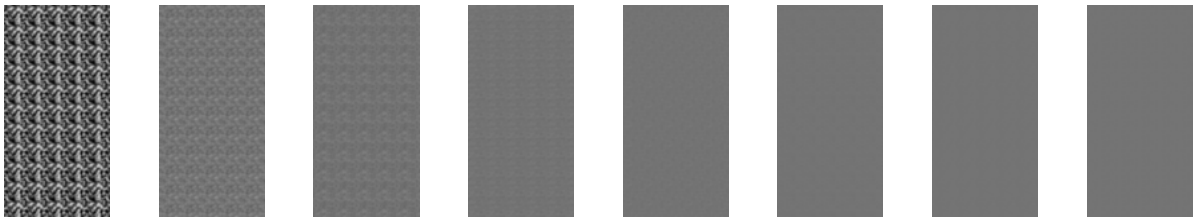


Figure 98: Convergence of the prefiltering algorithm for an increasing oversampling factor. The image shows a portion of a large chess board consisting of a grid of black and white squares with a side length below the pixel level. The images have been rendered by reconstruction from an approximating point cloud using a Gaussian reconstruction kernel (using the raytracing technique described in Chapter 6). The first row shows the results (please note that the regular structure of the noise artifacts reflects the instantiation grid). The second row contains the oversampling factors; the third row gives the required preprocessing time.

expected for a Monte Carlo integration technique, the image quality converges quickly for small sample sizes but more slowly for larger sample sizes, due to the $O(n^{-1/2})$ error behavior (see Section 1.3.2.1).

We obtain an acceptable image quality for 100× - 200× oversampling, but even at the highest level (1000×) we still obtain deviations by ±1 bit (for 8 bit color values). This is a general problem. Even enhanced numerical integration techniques such as Gaussian quadrature do not yield a better convergence rate for high variance, quasi random functions such as our test scene. However, they could speed up convergence in less adverse, smoother cases.

7.2.2.5 Animation

We have implemented a variant of the static sampling data structure for handling keyframe animations, as described in Section 4.3.3. This data structure uses the same construction parameters as the static version. Thus, the results from the preceding section apply also to the animated variant. In addition, the animated data structure provides an additional parameter to control time quantization, i.e. to use multiple sample sets within the time interval between two keyframes in order to adapt more tightly to the required sampling density at a certain time.

In order to examine the effect of this parameter, we have examined two test scenes: The first is a sphere shrinking linearly to a tenth of its original size. The second scene shows a walking human character (exported from the Poser animation package [Curious Labs 2001], see Figure 109). For the shrinking sphere, the average number of sample points can be reduced by a factor of 2.48, employing 20 time subintervals within the two keyframes (the theoretical maximum for shrinking to zero size is 3, cf. Section 4.3.3.3). A good approximation is already obtained using 3-5 time intervals, yielding already a reduction factor of 1.8-2. For the animated human, we do not achieve substantial savings using time discretization because the surface area does not change significantly; the maximum deviation of the number of sample points was below 4%. This means that we will not need time discretization for typical applications such as rendering animated humans, animals or machinery as their surface area usually does not change significantly over time. We have also measured the preprocessing time. As sample point generation is the most expensive preprocessing step in static sampling, the preprocessing time grows approximately linearly with the number of time intervals employed. The storage overhead is fairly small (a few percent only). In our implementation, it is dominated by storage costs for point attributes, hierarchy nodes and the original scene graph.

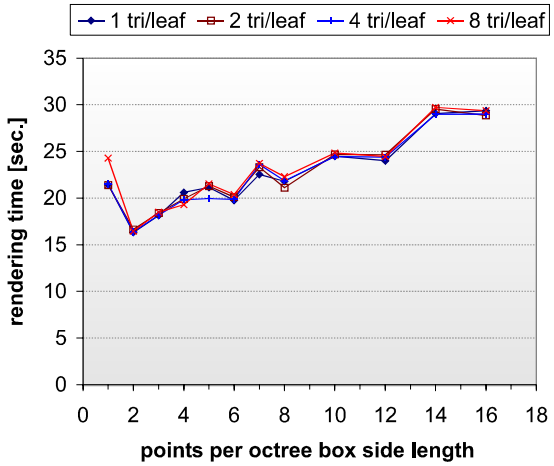


Figure 99: Rendering time for point-based multi-resolution raytracing (scene depicted in Figure 79a, near view) for a varying number of points per octree node and different numbers of triangles in leaf nodes. An optimal rendering time is achieved for $k=2$ points per box side length. The number of leaf node triangles has only a weak effect on the rendering time.

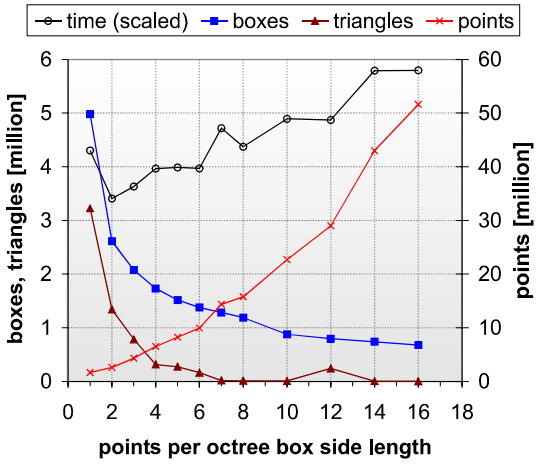


Figure 100: The number of points, triangles and boxes that have been tested for intersection by the point-based multi-resolution raytracing algorithm, in dependence of the number of points per octree box side length. Please note that the number of points is scaled differently (right caption) from the number of boxes and triangles (left caption).

7.2.2.6 Raytracing

We have implemented a point-based multi-resolution raytracer based on the static sampling data structure with prefiltered sample points. The most important preprocessing parameter is again the spatial approximation accuracy. In the case of general raytracing, we cannot specify an approximation accuracy ε as this depends on the ray density and its variation due to light propagation. Therefore, we plot the (following) measured running times as depending on the number k of points per box side length.

For point-based raytracing, two effects affect the trade-off for the rendering time: First, we obtain a better adaptation to the ray density if we use smaller point clouds in each node. Second, the hierarchy is also used as acceleration data structure for the ray queries. This means that we also have to test more points probably not intersecting a ray if we increase the number of points in each octree box. We expect that the second effect affects the rendering time severely, shifting the optimal rendering time closer to a very small number of sampling points per octree box.

Figure 99 shows the measured rendering time for the near view of our test scene (Figure 79a). As expected, the optimal rendering time is achieved for a small number k of points per box side length, in our case for $k = 2$. We have repeated the measurements with different maximum numbers of triangles in the leaf nodes of the octree (this is the termination criterion for the recursive construction). For a larger number of triangles, the performance drops slightly, but, overall, the parameter has only a limited effect on the running time. Figure 100 shows the composition of the rendering costs, i.e. the number of points, triangles and octree nodes that have been tested for intersection. Again, the number of triangles and boxes increases for smaller k while the number of tested points increases with a larger k. As expected, the increase is much stronger than in the forward mapping case because we have to test all points in a box for intersection if it is intersected.

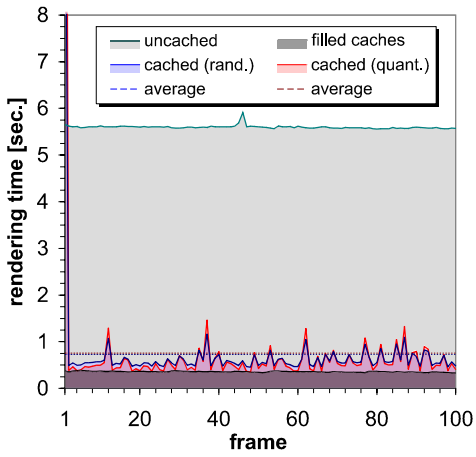
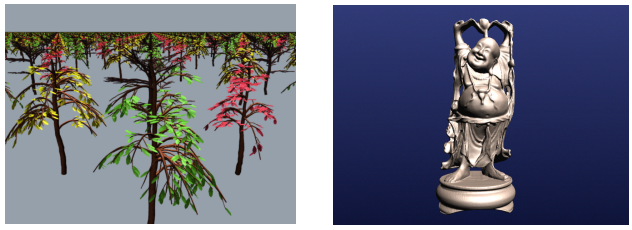


Figure 101: Speeding up dynamic sampling for walkthrough animations using sample caching. For moderate walking speed, the rendering time comes close to static sampling.



(a) replicated trees **(b) Happy Buddha**

Figure 102: Test scenes for measuring stratification efficiency.

7.3 Comparing Forward Mapping Techniques

After determining suitable parameter sets for using our algorithms and data structures, we compare the performance and image quality of our different rendering approaches. In this section, we compare different forward-mapping rendering techniques.

7.3.1 Performance

We can choose between several variants of sampling data structures: Dynamic sampling and static sampling. Dynamic sampling can be speed up using sample caching, static sampling can be implemented with hierarchically nested sample sets or full resampling at each level. In this section, we examine the consequences of different strategies concerning the rendering performance. We use the different data structures to render our benchmark scene (Figure 79) and compare the resulting performance.

7.3.1.1 Comparing Dynamic and Static Sampling

First, we compare the performance of dynamic and static sampling for our test scene (Figure 79, near view). Table 5 - Table 7 summarize the results. Employing optimized rendering parameters, dynamic sampling of a near view of our example scene takes 5.7 seconds. Static sampling (k = 48 points/box side length, neighborhood-based stratification) displays the same image within 279/88ms (OpenGL/DirectX renderer). In order to enhance the performance of dynamic sampling, we can employ sample caching. Instead of performing a full resampling for each image, sample sets are cached (being stored in an OpenGL display list) and reused for subsequent frames. This approach is able to provide a considerable speedup: Once the sample sets are cached, rendering can be performed in 517ms (using rendering parameters optimized for rendering form caches). Performing grid stratification by quantization on-the-fly, we can reduce the rendering time further to 390ms. Neglecting the implementation specifics of using different rendering APIs, this is roughly comparable to static rendering. This result has been expected, as cached dynamic sam-

pling uses a similar data structure for rendering as static sampling. The numbers of sample points and triangles used for stratified rendering are comparable in both cases.

However, rendering from precomputed caches is not realistic for typical applications. Usually, the caches are created dynamically on demand during a walkthrough of the scene. In such a situation, the rendering performance can vary between full resampling and rendering from cache. To examine the performance for a typical case, we have recorded a camera path with typical walking speed and recorded the resulting rendering speeds. Figure 101 shows the results: In contrast to precomputed caching, dynamic caching leads to a varying frame rate. The average rendering time is increased to 0.731 seconds for random sample caches and 0.759 seconds for quantized caching. This is 1.5-2 times slower than precomputed caches but still 8 times faster than a full resampling at every frame. Interestingly, our implementation of quantized caching is slightly slower than random sample sets on the average due to higher processing costs (the sample cache uses quantization by successive bucket-sorting in all three-dimensions, which is general but fairly expensive, see Section 4.2.3.3). Of course, in general, the outcome depends on the walking speed, scene characteristics and on implementation details of the quantization algorithm.

The static sampling algorithm renders the camera path with 261ms (OpenGL) and 90ms (DirectX), respectively, with near-constant framerates. Therefore, in conclusion, the static sampling strategy is superior concerning rendering performance. Using a comparable rendering API, it is typically about 2 times faster. The only performance advantage of dynamic sampling is the small precomputation time (8 seconds in comparison to several minutes, see Section 7.2.2.2).

7.3.1.2 Stratification

Using static sampling, we have the choice of different stratification techniques that also affect performance. In order to examine the influence, we apply the four main techniques (purely random sampling, grid stratification, quantized grid stratification, neighborhood-based point removal) to two example scenes (Figure 102) and compare the resulting number of primitives. The first example scene is the replicated-trees scene from the previous sections. The second is a smooth mesh, the well known Stanford “Happy Buddha” mesh ([Stanford 2004], data taken from [Garland 2003]). The resulting preprocessing and rendering costs are summarized in Table 8 and Table 9.

Random sampling: According to Section 5.3.1.1, the expected value for covering all pixels

	full resampling	caching	quantized caching	precomp. quant. cache
near view	5.725 sec	0.517 sec	0.390 sec	–
fly over (average)	5.598 sec	0.731 sec	0.759 sec	0.357 sec

Table 5: Performance of dynamic sampling for our test scene (Figure 79).

	OpenGL (std. vertex arrays)	DirectX (managed vert. buffers)
near view	0.279 sec	0.088 sec
fly over (average)	0.261 sec	0.090 sec

Table 6: Performance of static sampling.

	dynamic random	cached random	cached quantized	static sampling
rendered points	4,170,910	4,670,691	1,979,219	2,019,438
rendered triangles	367,116	367,116	367,116	374,732

Table 7: Rendering primitives used for display.

"Replicated Trees"	random-12	random-8	grid-20	grid-8	quant. grid	neigh. rem.
oversampling	12	8	20	8	20	20
prepr. size [MB]	110	75	79	49	41	21
prepr. time [sec]	20	16	150	140	36	125
rendering points	13,365,786	8,910,580	9,675,980	5,875,439	3,923,541	1,713,152
reduction [%]	100,0%	66.7%	72.4%	44.0%	29.4%	12.8%
rendering triangles	188,811	188,811	188,811	188,811	188,811	188,811
rendering time [sec]	2.480	1.675	1.813	1.142	0.770	0.425

Table 8: Efficiency of different stratification techniques (scene: replicated trees, 8×8, 2 instantiation layers, near view). The rendering times have been determined using unoptimized OpenGL immediate mode rendering (cf. Table 4).

"Happy Buddha"	random-12	random-8	grid-20	grid-8	quant. grid	neigh. rem.
oversampling	12	8	20	8	20	20
prepr. size [MB]	226	187	190	171	144	129
prepr. time [sec]	39	35	79	76	40	143
rendering points	630,106	419,468	443,763	334,965	181,806	96,549
reduction [%]	100.0%	66.6%	70.4%	53.2%	28.9%	15.3%
rendering triangles	0	0	0	0	0	0
rendering time [sec]	0.117	0,076	0.080	0,061	0.034	0.022

Table 9: Efficiency of different stratification techniques (scene: "Happy-Buddha" mesh). The rendering times have been determined using unoptimized OpenGL immediate mode rendering (cf. Table 4).

of the image by random sampling is $\ln a$. For our benchmarks, this means that we need an oversampling factor of about $\ln(640 \times 480) \approx 12$ (due to the logarithmic behavior, the exact value of a is of minor importance). Oversampling means that we first divide the surface area within an octree box by the size of an unprojected pixel and count the number of resulting "pixels". Then, we multiply this value by the oversampling factor to determine the necessary number of random points. Using this oversampling factor, the example scenes need 13M (trees) / 630K (Buddha) points for display. Alternatively, we have also tried different oversampling factors and examined the resulting images manually. Especially for the smooth mesh model, holes and missing pixels are easily visible. Using this approach, we have determined oversampling factor of 8 leading to an acceptable image quality (for the mesh model). At this sampling density, only a few single pixel-sized holes become visible for certain viewpoints. This reduces the size of the sample set by one third[39]. For the forthcoming examples using post-processing of a random sample set for further stratification, we have used an oversampling factor of 20, providing sufficiently save coverage at moderate costs.

Grid stratification: Applying simple grid stratification (i.e. deleting all points except one in a regular grid) to a high density random sample set (oversampling factor 20), we obtain sample sets of similar size than reduced random sampling (with oversampling factor 8). However, this technique provides safe coverage with much higher probability. (If we apply grid stratification to the smaller random sample set, we still obtain a reduction of the sample size to about 70% of the original size). Quantized grid stratification is more efficient. It allows a reduction of the sample size to less than 30% of the random sample set. Additionally, the quantized grid stratification technique is the most efficient technique. It needs only slightly more computation time than the random sampling itself.

[39] We have used a similarly reduced sampling density for the dynamic (random) sampling examples, too.

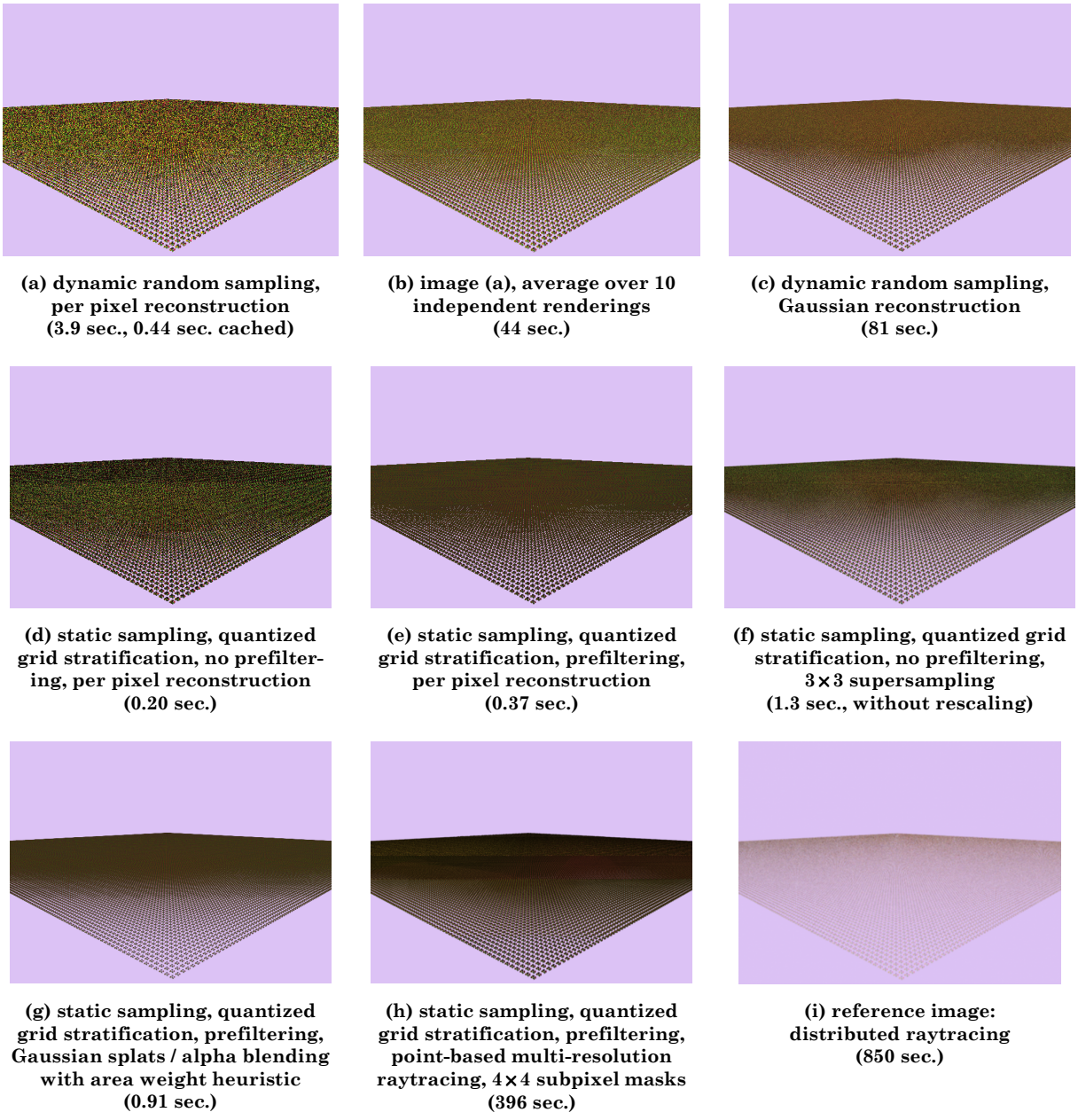
Neighborhood-based point removal: The neighborhood-based point removal strategy is more expensive in terms of precomputation time (about 4 times slower than quantized grid stratification, both including the initial random sampling). However, it yields the best results: The resulting sample sets contain only 13-15% of the number of sample points of the random sampling technique, still guaranteeing the same sample spacing.

7.3.1.3 Nested Sampling and Full Sampling

The static sampling data structure can be implemented in two variants: Full sampling and nested sampling. The first variant (used so far) stores a new sample set at each hierarchy level. The second propagates subsets of higher density sample sets to parent nodes in order to create lower density sample sets. Comparing the number of sample points within the hierarchy (employing grid stratification), the full sampling data structure uses 1.6 (the 4 trees from the replicated trees scene) to 2.1 (Buddha) times more sample points than nested sampling. Hence, nested sampling can save some memory if prefiltering is not needed (which does not work for nested sampling). However, in comparison with current point cloud compression techniques [Botsch et al. 2002], the savings in practice are moderate. Thus, the technique is more of theoretical interest: it allows the usage of a deterministic rendering algorithm with a precomputed data structure using optimal (i.e. linear) asymptotic memory demands.

7.3.2 Image Reconstruction

After examining the performance implications of the different rendering techniques, we now take a look at the image quality obtained using different rendering techniques. We have combined different sampling and reconstruction techniques described previously. The resulting set of rendering techniques has been applied to a set of benchmark scenes. Please note that the implementation used in this section has not been optimized specifically for rendering performance (rendering has been mostly done in OpenGL immediate mode) as this is not the topic of this experiment. Though, timings are given for the example scenes to give an impression of the relative performance. Examples with maximum rendering performance are discussed in Section 7.3.3. The following rendering techniques have been used in the forthcoming comparison: dynamic random sampling is combined with per-pixel reconstruction, averaging, and Gaussian reconstruction. Static sampling is used with and without prefiltering, both with per-pixel reconstruction. Additionally, screen space oversampling and alpha blending are considered to improve the image quality. For a comparison, we have also included images rendered with point-based multi-resolution raytracing and a reference solution rendered with adaptive distribution raytracing. We have applied the techniques to 4 example scenes: Figure 103 shows the rendering results for a forest scene consisting of 65536 trees (6.1 billion triangles), seen from far apart. This scene demonstrates noise and aliasing problems as well as overestimation of subpixel occlusion. A closer view of the same scene is shown in Figure 104. Figure 105 shows a landscape scene consisting of a fractal heightfield and several instances of tree models, similar to the previous scene (400 million triangles). This scene demonstrates the consequences for a more realistic scene (especially concerning overestimation of occlusion and prefiltering artifacts). Figure 106 shows the results for the (classic) chess board benchmark. This scene reveals aliasing artifacts more clearly than the previous models. Lastly, Figure 107 shows a city scene consisting of a large collection of simple building, car and street models (leading to a scene complexity of 1.9 billion triangles). The scene reveals aliasing problems as well as the behavior for complex occlusion effects.



(a) dynamic random sampling, per pixel reconstruction (3.9 sec., 0.44 sec. cached)

(b) image (a), average over 10 independent renderings (44 sec.)

(c) dynamic random sampling, Gaussian reconstruction (81 sec.)

(d) static sampling, quantized grid stratification, no prefiltering, per pixel reconstruction (0.20 sec.)

(e) static sampling, quantized grid stratification, prefiltering, per pixel reconstruction (0.37 sec.)

(f) static sampling, quantized grid stratification, no prefiltering, 3×3 supersampling (1.3 sec., without rescaling)

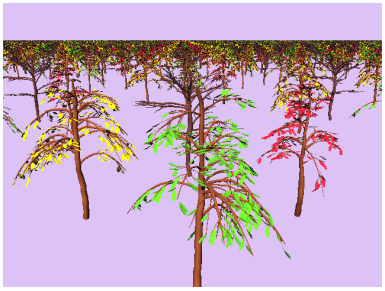
(g) static sampling, quantized grid stratification, prefiltering, Gaussian splats / alpha blending with area weight heuristic (0.91 sec.)

(h) static sampling, quantized grid stratification, prefiltering, point-based multi-resolution raytracing, 4×4 subpixel masks (396 sec.)

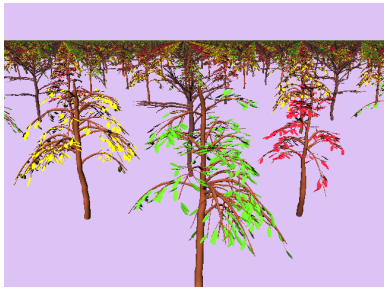
(i) reference image: distributed raytracing (850 sec.)

Figure 103: Image quality / antialiasing comparison for different image reconstruction techniques (640×480 pixels). Scene: replicated trees, far view (16×16, 2 hierarchy layers, cf. Figure 79)

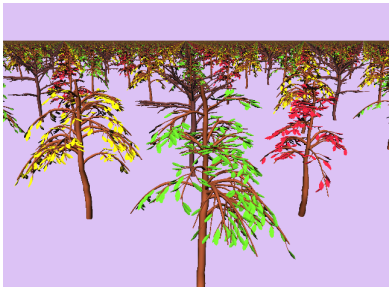
The rendering techniques use different shading models, leading to some variation in the output: The forward mapping algorithms use precomputed diffuse lighting. Due to a slightly different implementation, the scenes computed with dynamic sampling appear a bit brighter. To compensate for this effect, we have increased the contrast in the static sampling images in Figure 105 and Figure 107. The raytracing algorithms use a Phong lighting model, again showing some color differences. Besides this implementation specific issues, some general effects can be observed. In the following, we will discuss the observations for the different techniques:



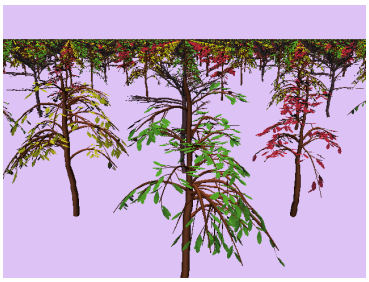
(a) dynamic random sampling, per pixel reconstruction (8.3 sec., 1.0 sec. cached)



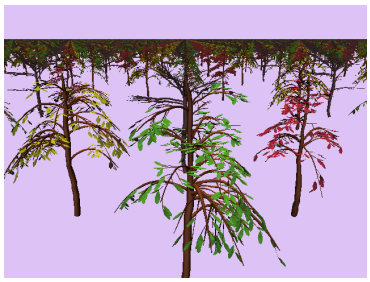
(b) image (a), average over 10 independent renderings (44 sec.)



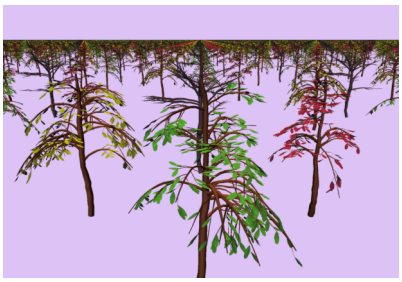
(c) dynamic random sampling, Gaussian reconstruction (81 sec.)



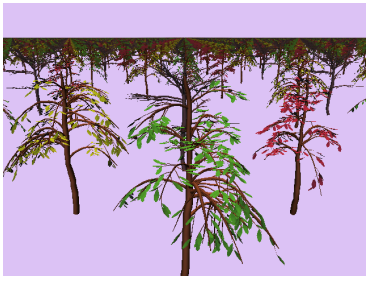
(d) static sampling, quantized grid stratification, no prefiltering, per pixel reconstruction (0.20 sec.)



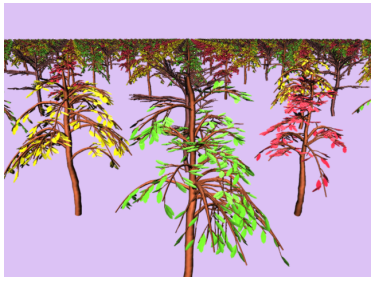
(e) static sampling, quantized grid stratification, prefiltering, per pixel reconstruction (0.37 sec.)



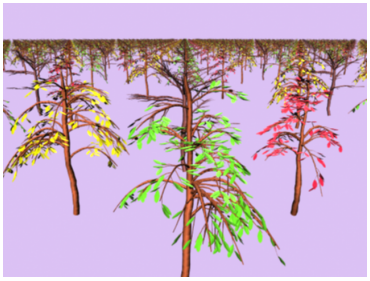
(f) static sampling, quantized grid stratification, no prefiltering, 3×3 supersampling (1.3 sec., without rescaling)



(g) static sampling, quantized grid stratification, prefiltering, Gaussian splats / alpha blending with area weight heuristic (0.91 sec.)



(h) static sampling, quantized grid stratification, prefiltering, point-based multi-resolution raytracing, 4×4 subpixel masks (396 sec.)

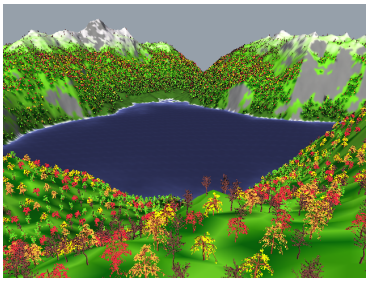


(i) reference image: distributed raytracing (850 sec.)

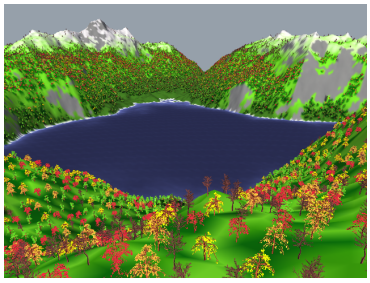
Figure 104: Image quality / antialiasing comparison for different image reconstruction techniques (640×480 pixels). Scene: replicated trees, near view (16×16, 2 hierarchy layers, cf. Figure 79)

Per-pixel reconstruction, no prefiltering: We have used this reconstruction technique with dynamic (images with letter (a)) and static (images (d)) sampling. Dynamic sampling uses random sample sets; the static technique uses a quantized version of a random candidate set. No prefiltering is performed.

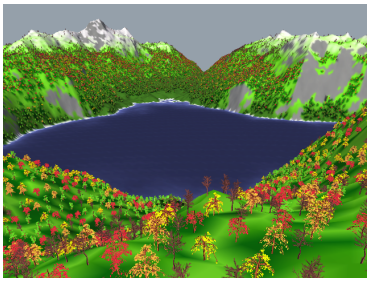
The rendering technique leads to noise and aliasing artifacts for two reasons: First, the random selection of sample points creates noise. Second, the display on screen creates aliasing. Two effects are relevant: The oversampling factor leads to a depth dominance effect, i.e. only the



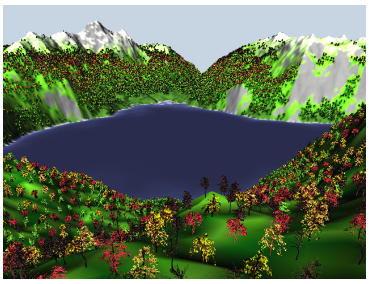
(a) dynamic random sampling, per pixel reconstruction (3,9 sec., 0.8 sec. cached)



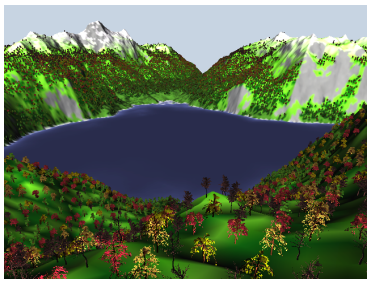
(b) image (a), average over 10 independent renderings (28 sec.)



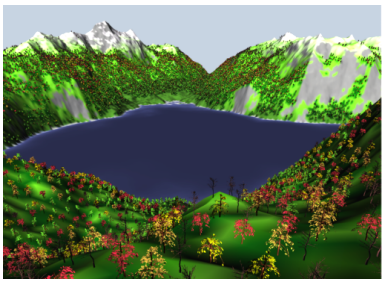
(c) image (a), average over 100 independent renderings (340 sec.)



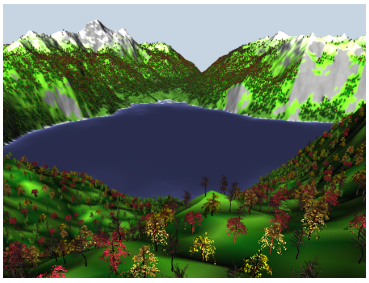
(d) static sampling, quantized grid stratification, no prefiltering, per pixel reconstruction (1.5 sec.)



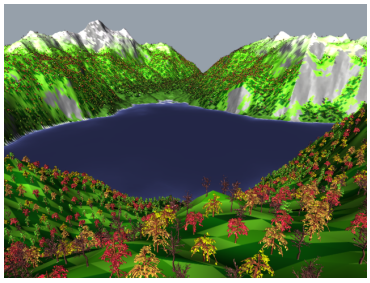
(e) static sampling, quantized grid stratification, prefiltering, per pixel reconstruction (0.71 sec.)



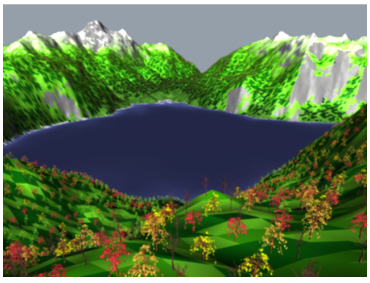
(f) static sampling, quantized grid stratification, no prefiltering, 3×3 supersampling (6.8 sec., without rescaling)



(g) static sampling, quantized grid stratification, prefiltering, Gaussian splats / alpha blending with area weight heuristic (1.7 sec.)



(h) static sampling, quantized grid stratification, prefiltering, point-based multi-resolution raytracing, 4×4 subpixel masks (missing color interpolation for the terrain, 215 sec.)

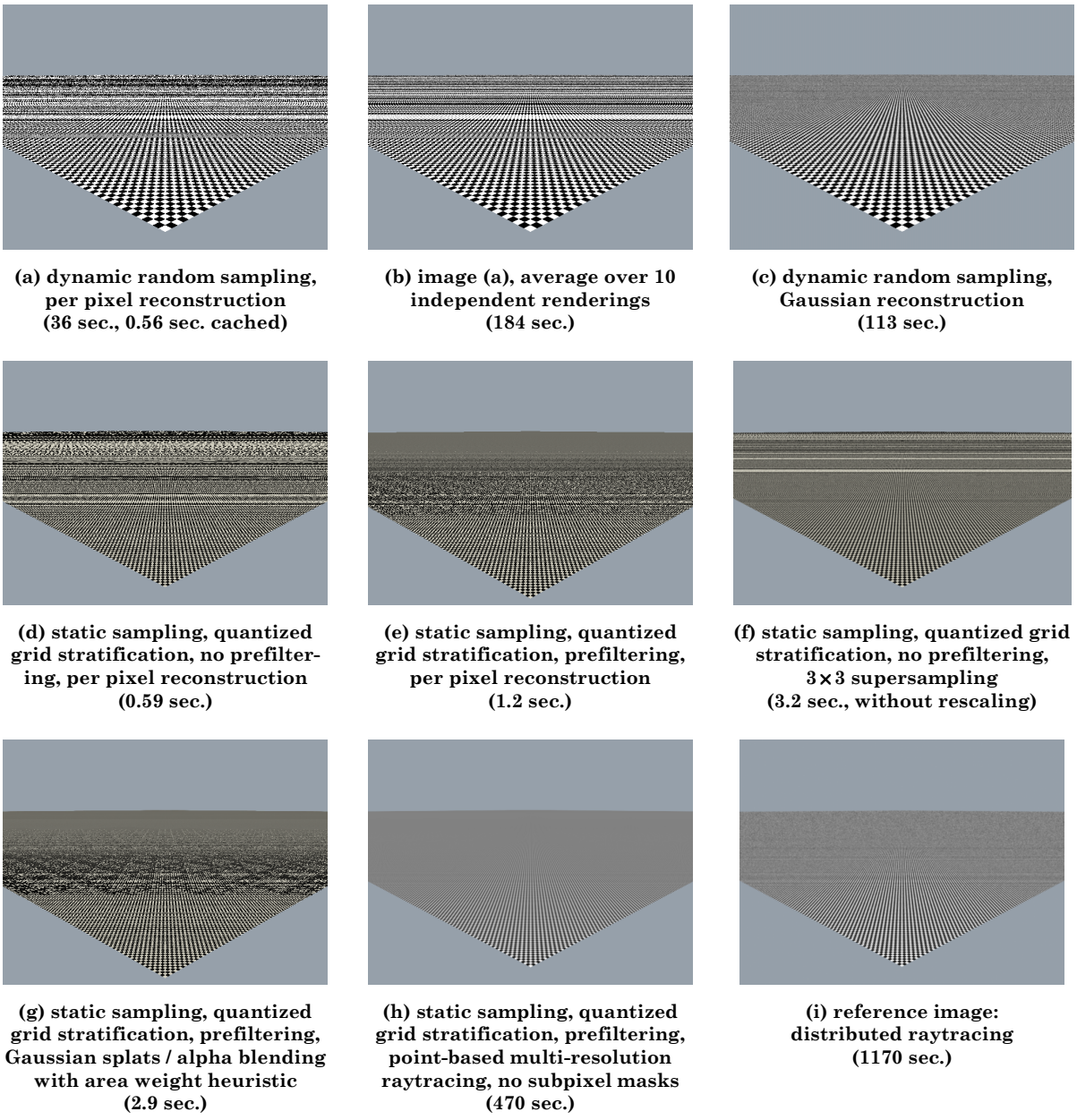


(i) reference image: distributed raytracing (missing color interpolation for the terrain model, 760 sec.)

Figure 105: Image quality / antialiasing comparison for different image reconstruction techniques (640×480 pixels). Scene: replicated trees, near view (16×16, 2 hierarchy layers, cf. Figure 79)

foremost pixel is selected by the underlying z-buffer process, creating structured aliasing pattern. Second, the quantization to a pixel grid itself also creates aliasing, as no prefiltering is performed.

The effects can be seen nicely in the chess board example scene (Figure 106). The noise artifacts are visible in the other scenes, too (whereas structured aliasing is not as noticeable due to the irregular scene structure). A closer look at Figure 103d also reveals an additional property of static sampling: The noise patterns are repeated in different regions of the scene, according to the instantiation pattern. This does not occur for dynamic sampling. Additionally, the transitions



(a) dynamic random sampling, per pixel reconstruction (36 sec., 0.56 sec. cached)

(b) image (a), average over 10 independent renderings (184 sec.)

(c) dynamic random sampling, Gaussian reconstruction (113 sec.)

(d) static sampling, quantized grid stratification, no prefiltering, per pixel reconstruction (0.59 sec.)

(e) static sampling, quantized grid stratification, prefiltering, per pixel reconstruction (1.2 sec.)

(f) static sampling, quantized grid stratification, no prefiltering, 3×3 supersampling (3.2 sec., without rescaling)

(g) static sampling, quantized grid stratification, prefiltering, Gaussian splats / alpha blending with area weight heuristic (2.9 sec.)

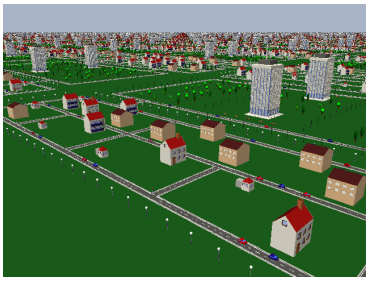
(h) static sampling, quantized grid stratification, prefiltering, point-based multi-resolution raytracing, no subpixel masks (470 sec.)

(i) reference image: distributed raytracing (1170 sec.)

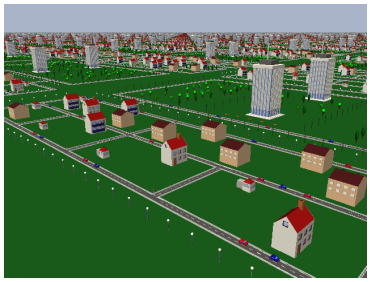
Figure 106: Image quality / antialiasing comparison for different image reconstruction techniques(640×480 pixels). Scene: chess board with 8×8 black and white quads, replicated on two instantiation layers of 16×16 instances each.

between different hierarchy levels are clearly visible, as we do not perform an interpolation between different hierarchy levels for this rendering technique.

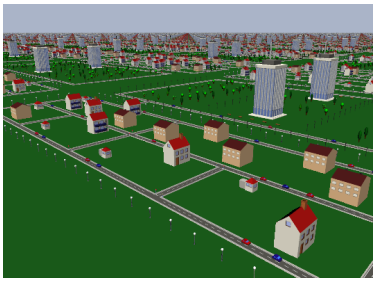
Averaging: To overcome the problem of noise artifacts, we have the option to increase the sampling density and compute an average color for every pixel. The simplest implementation is averaging with dynamic sampling: For the rendering examples (images (b)), we have computed 10 images independently and computed the average image using the OpenGL accumulation buffer. This already leads to a considerable noise reduction, but still some noise artifacts are still visible. This is especially a problem for animations. For a higher rendering quality, more sample images



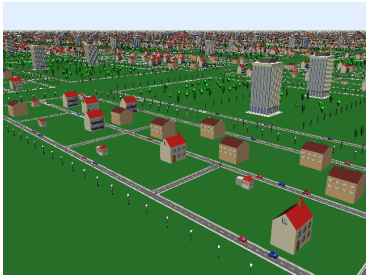
(a) dynamic random sampling, per pixel reconstruction (4.3 sec., 0,20 sec. cached)



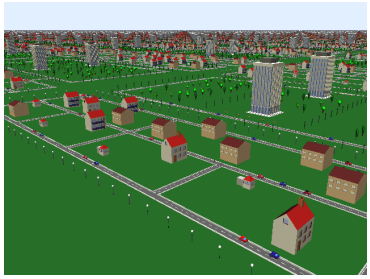
(b) image (a), average over 10 independent renderings (49 sec.)



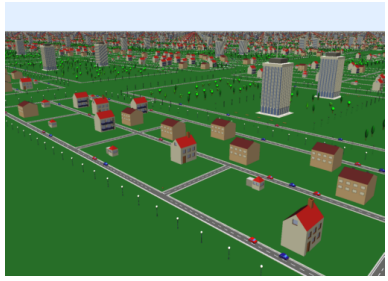
(c) dynamic random sampling, Gaussian reconstruction (85 sec.)



(d) static sampling, quantized grid stratification, no prefiltering, per pixel reconstruction (1.1 sec.)

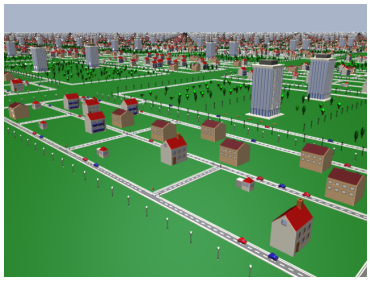


(e) static sampling, quantized grid stratification, prefiltering, per pixel reconstruction (0.9 sec.)

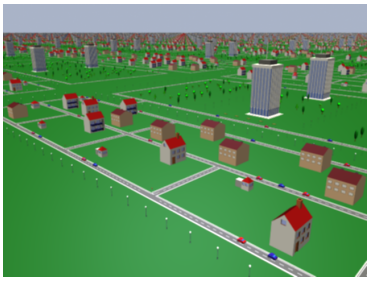


(f) static sampling, quantized grid stratification, no prefiltering, 3×3 supersampling (3.0 sec., without rescaling)

(g) static sampling, quantized grid stratification, prefiltering, Gaussian splats / alpha blending with area weight heuristic (2.5 sec.)



(h) static sampling, quantized grid stratification, prefiltering, point-based multi-resolution raytracing, 4×4 subpixel masks, no mipmapping (887 sec.)



(i) reference image: distributed raytracing (996 sec.)

Figure 107: Image quality / antialiasing comparison for different image reconstruction techniques (640×480 pixels). Scene: a city scene consisting of 192,431 triangles, replicated on two instantiation layers of 10×10 instances each.

are needed: Figure 105c shows an image created by averaging 100 images, leading to an improved quality in comparison with Figure 105b. Even in conjunction with random sampling, averaging does not avoid aliasing artifacts. The phenomenon of depth dominance in conjunction with the simple box filter (which averaging corresponds to) can still create considerable structured aliasing artifacts.

This is clearly visible in the chess board example (Figure 106). For scenes structured less regularly, structured aliasing artifacts pose only a minor problem but they are still visible. This

can be seen for example in the far view of the tree scene (Figure 103), especially in comparison with better reconstruction techniques as used in image (c).

The idea of averaging can also be used with static sampling. However, here it is not possible to create a temporal sequence of independent images. Instead, we have to create a larger number of sample points by increasing the resolution, i.e. performing a deeper traversal of the point hierarchy. The simplest implementation is supersampling: We compute an image at a higher resolution and perform downsampling using a suitable lowpass filter later on. The images with letter (f) show the results for 3×3 oversampling. In this case, downsampling was done using image processing software and is not accounted for in the reported rendering time: Again, noise and aliasing artifacts are reduced. For unstructured scenes, less aliasing artifacts are observed than for simple averaging because the image processing software uses a better downsampling filter than a simple box filter. Additionally, occlusion is resolved at a higher resolution due to the supersampling approach. Nonetheless, critical scenes such as depicted in Figure 106 still cause aliasing artifacts due to depth dominance and the simple oversampling grid (see also Figure 107f).

Gaussian reconstruction: In order to avoid the remaining aliasing problems, we need an image reconstruction technique that does not rely on quantization grids. The images with letter (c) show results of the Gaussian reconstruction technique described in Section 5.3.2.5. For each pixel, neighboring pixels are deleted if they are farer away (after subtracting a constant depth tolerance value in projective z-coordinates). Additionally, backface culling is used to avoid artifacts at silhouettes. A weighted average using a Gaussian filter is then computed for the visible pixels. The resulting images show very little aliasing artifacts. Some noise artifacts are still present as we have used only a moderately sized sample size (corresponding to approx. 5 times oversampling). The implementation is fairly slow (it has been implemented using OpenGL framebuffer readbacks on a high resolution version of the rendering). Therefore, the resulting rendering times give only a rough indicate of the necessary effort. The simple depth tolerance heuristic for merging adjacent points is not always applicable: For the landscape scene in Figure 105, no single tolerance interval could be determined that resolves local occlusion correctly. Thus, no image has been included. In contrast, the improved depth interval heuristic (Section 6.2.6) employed in the backward mapping renderer works well in all example scenes (see below).

Prefiltering: In order to avoid aliasing artifacts in static sampling, we have the option to perform prefiltering: Point attributes are computed by suitably averaging over the local neighborhood. We have implemented this technique for color attributes: Diffuse lighting is precomputed and the resulting color is used to build the hierarchy, prefiltered using a Gaussian filter kernel. The results (in conjunction with simple per-pixel reconstruction) are shown in the images (e): Prefiltering effectively removes noise and aliasing artifacts for strongly minified regions. However, for regions where the frequency of the image details is in the range of the pixel frequency, we still obtain aliasing due to the simple per-pixel reconstruction. Prefiltering also has some drawbacks: If complex geometry is substituted with a constant parameter set, inevitable errors are introduced. The usage of a fixed prefiltering algorithm (especially the usage of a simple averaging scheme, as done here) can easily lead to a systematic bias. The problem can be seen for example by comparing images (d) and (e) (static sampling with and without prefiltering) in Figure 104 and Figure 105: The leafs the small branches of the trees have varying color, especially due to the darker back sides. Thus, the average is too dark. The problem becomes especially visible in comparison with the foremost trees in the images, which have been (automatically) rendered using z-buffer rendering. For pure point samples (no prefiltering), this problem is not as critical. We still cannot resolve subpixel occlusions with guaranteed precision. However, it is easier to reconstruct believable images. Leakage of back sides can for example be avoided by applying backface culling based on the normals of the sample points (this has been done in images (a)-(c)).

Splatting with alpha blending: An alternative to reconstruction by weighted averaging of visible sample points is the alpha-blending technique proposed by different authors [Rusinkiewicz and Levoy 2000, Coconu et al. 2002]. Splats with a unit Gaussian in the alpha channel are drawn in back to front order. Additionally, we multiply the alpha values (during prefiltering) with estimated opacity values derived from the area-based weight heuristic described in Section 4.2.4.4: The transparency is estimated by the ratio of the local area around a sample point and the minimum area needed for completely covering a pixel. This approach is combined with prefiltering of color attributes. Additionally, attributes and opacities are interpolated between two adjacent hierarchy levels. The results are shown in images (g). For distinguishable, complex occlusion effects (Figure 103, forest far view) the reconstruction quality of alpha blending is better than simple splatting and even slightly better than Gaussian reconstruction, which overestimates the opacity more strongly. This is due to the area-based opacity heuristic. For the reconstruction of regularly structured continuous surfaces, alpha blending does not provide significant advantages. As expected, alpha blending fails to reconstruct the local surface attributes faithfully for the chess board scene Figure 106g. The foremost sample points on each pixel are dominant, creating again aliasing artifacts. The artifacts are comparable to those obtained by simple splatting.

Point-based multi-resolution raytracing: Better antialiasing properties can be achieved using surface splatting [Zwicker et al. 2001a]. We use a modified variant of surface splatting for the point-based multi-resolution raytracer described in Chapter 6. A more detailed evaluation will be given in the next subsection. However, we include some examples here for a comparison (images (h)). The example renderings use primary rays only. Thus, the same images could also have been created using a forward mapping algorithm based on an a-buffer implementation [Zwicker et al. 2001a], which would be substantially faster. In general, the compositing strategy of the multi-resolution raytracing approach yields the best image quality in our comparison. The technique does not show strong prefiltering artifacts for our test scenes because it uses average surface properties (normal, curvature, material properties) rather than average precomputed colors. For strongly varying properties (e.g. a set of leafs of a tree collapsed into a single sample point), this is also only a crude approximation but the artifacts do not become similarly apparent: We obtain a somehow random shading instead of darkened colors which represents the features in the test scenes more faithfully. Structured aliasing artifacts are avoided, even in problematic scenes (see Figure 106h). Lastly, it should be noted that two different reconstruction techniques have been applied: For the forest scene (Figure 103), the depth tolerance region has been set to zero, effectively performing compositing by subpixel masks only (no surface merging). The other scenes have been rendered using merging of adjacent point and triangle intersection events. Non-merging compositing yields better results for complex, irregularly structured occlusion. However, the current implementation does not yet allow interpolation between adjacent hierarchy levels for non-merging compositing, leading to visible borders in Figure 103h. Using linear interpolation, such artifacts are avoided (Figure 106h). More details on point-based multi-resolution raytracing will be discussed in the next Section (7.5).

Reference Images: The last set of images (letter (i)) show reference solutions (rendered using the same shading parameters as images (h)). They have been created using distributed raytracing with high oversampling. Except for noise artifacts, this technique creates images that could be considered “correct” solutions to the rendering problem. The most apparent difference to all point-based multi-resolution rendering results is the reproduction of subpixel occlusion: Especially for the landscape scenes, the branches and leafs are much more transparent in the reference solution than in the multi-resolution renderings. Alpha blending reconstruction with area-based opacity estimates creates the least opacity overestimation of all examined techniques. Though, it still creates a strong overestimation in comparison to the reference images. Apart from

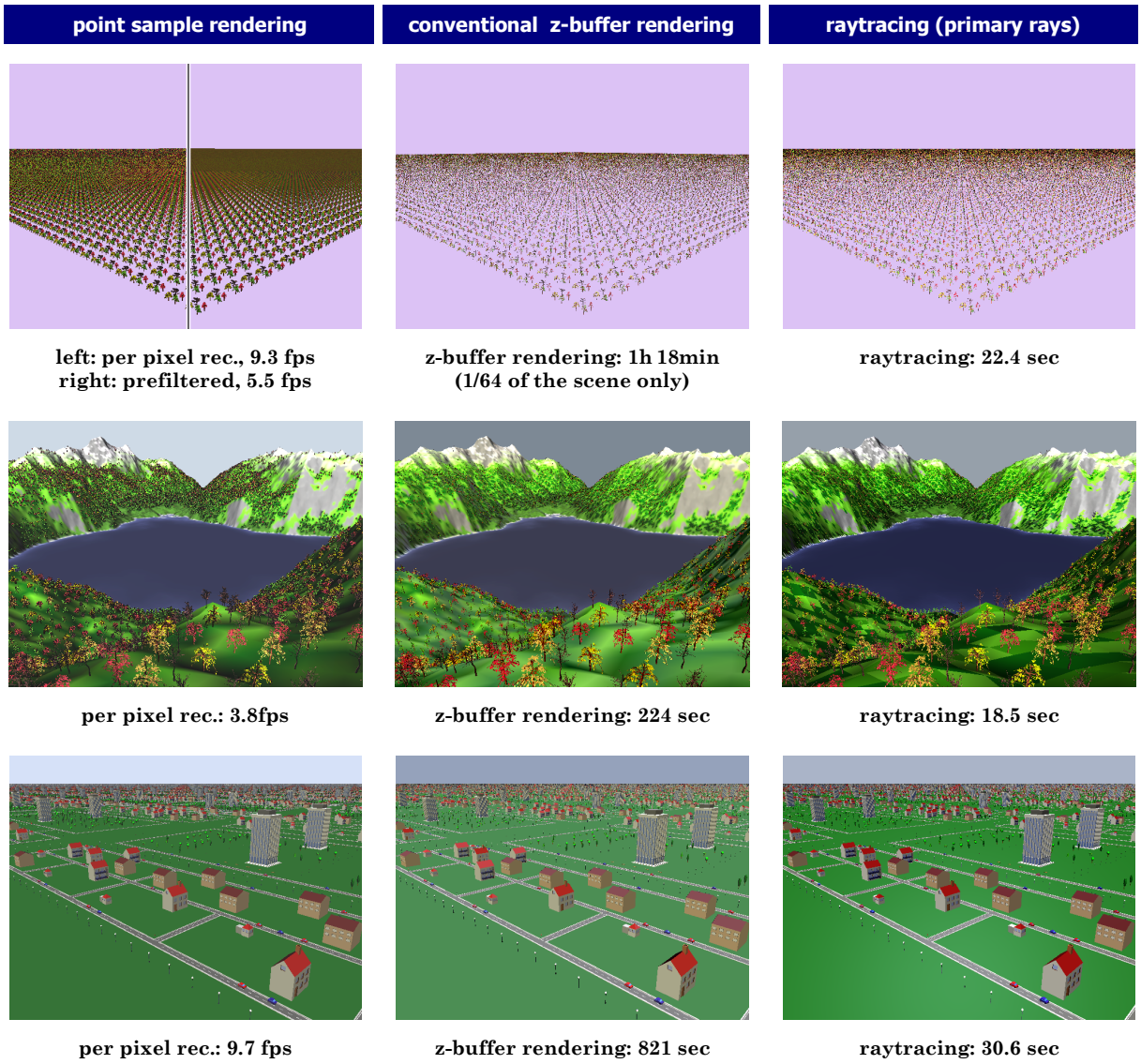


Figure 108: Visualization of complex scenes – comparing image quality and rendering performance to conventional z-buffering and raytracing (640×480 pixels).

this issue, the different point-based multi-resolution renderings (especially using Gaussian reconstruction or raytracing) already come close to the reference solution.

7.3.3 Comparison with Conventional Rendering Techniques

In this subsection, we compare rendering time and quality of point-based forward mapping with conventional z-buffering (no multi-resolution) and raytracing techniques. In practice, simple z-buffer rendering is still the predominant technique in interactive computer graphics. Raytracing (using spatial hierarchies) is an output-sensitive technique, thus being often used to handle complex scenes in offline applications.

As examples, we have picked three scenes from the image quality test section (7.3.2): the replicated trees scene, the mountain sea scene, and the city scene (Figure 108). Again, contrast

and brightness have been adjusted to compensate for the effect of slightly different shading algorithms. The replicated trees scene has been extended, now containing 4.2 million instances. This yields 390 billion triangles. Using the DirectX splatting renderer, drawing one pixel sized opaque splats, we can render the scene at 5-9 frames per second. A z-buffer rasterization of the foremost 1/64 of the scene already needs more than an hour (using OpenGL display lists). For the full scene, we would expect more than 11 hours of rendering time, even if the full throughput of 30 million vertices per second could be reached (the display lists renderer is slower). Raytracing is much more efficient. Using our simple implementation (see Section 7.5), an image can be computed in 22 seconds. Of course, our raytracing implementation has not been optimized and is far from reaching the performance of implementations such as [Wald et al. 2001]. Nevertheless, there are more than two orders of magnitude between raytracing and point-based rendering; and point-based rendering also leaves still some room for improvement[40].

For the other two example scenes we obtain similar results: Point-based rendering runs at real-time framerates while z-buffer rendering is very expensive. Raytracing achieves reasonable run times, but still being roughly two orders of magnitude slower than point-based multi-resolution rendering. Concerning image quality, we observe again an overestimation of silhouette opacity of the point-based technique. Raytracing and z-buffer rendering show strong aliasing artifacts, demanding for additional supersampling to remove these artifacts. By using prefiltered point attributes, these problems are reduced for the point-based approach. For fully antialiased rendering, screen-space filtering would have to be used. Currently, no sufficiently efficient implementation is available within our system (see last section). However, this is no general problem; the corresponding techniques have already been discussed in literature [Ren et al. 2002, Botsch et al. 2002].

7.4 Animated Scenes

The point-based multi-resolution data structure proposed in this thesis allows efficient rendering of complex keyframe animations. In order to employ the technique, a set of keyframe transitions has to be specified and corresponding point hierarchies are precomputed. This is possible for both animated triangle meshes and for instances of such models already providing a local point hierarchy. Within a precomputed hierarchy, motion has to be fixed in advance, only a set of keyframe transitions is possible. This includes precomputed hierarchies of instances. However, each top level instance (not being included in another hierarchy), can be instantiated freely, e.g. using geometric transformations or varying local time. Thus, we have to deal with two aspects in applications of the technique: First, we have to identify object groups and keyframes for which a multi-resolution hierarchy is precomputed. It is possible that these groups are also composed of instances of lower level hierarchies, especially if the model is fairly complex. Second, we have to design an algorithm for global control, i.e. for the dynamic instantiation of root level instances. The rendering of each point hierarchy itself is output-sensitive; the rendering time depends (mostly) on the projected area, not on the geometric complexity. If we place multiple instances in a scene, e.g. for a dynamic crowd simulation, the rendering time will be always at least linear in the number of root instances used. This still allows for a considerable scene complexity, but not for the virtually unlimited complexity that is possible within the precomputed instances.

---

[40] Probably not as much as the raytracing renderer. However, our implementation achieves only a throughput of 31 million points per second for this scene. The latest graphics hardware announced promises more than 300 million vertices per second [nVidia 2004b].

(a) far view (2.6M points, 0.30M triangles, 6500 nodes, 8.5/7.7fps w./w.o. supersampling)

(b) near view (2.6M points, 0.35Mtriangles, 6753 8.0/7.6fps w.o./w. supersampling)

Figure 109: A crowd of one million walking humans modeled using hierarchical instantiation. Rendering using per-pixel reconstruction and prefiltered point attributes (DirectX renderer), 640×480 pixel, 4× hardware oversampling. The framerates are averages over one animation period.

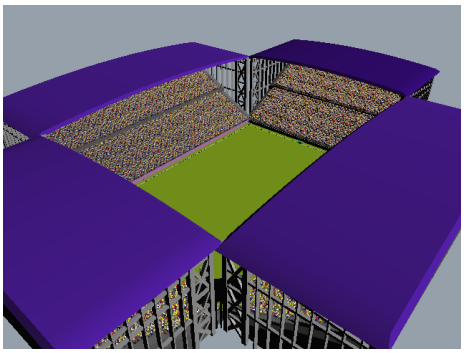
In order to demonstrate the different aspects and to show the achievable performance, we will consider two different types of application examples: The first uses hierarchical instantiation only. The second uses dynamic placement of instances in order to create a dynamic crowd simulation. The preprocessing costs for all example scenes are summarized in Table 10.

7.4.1.1 Static Hierarchical Instantiation

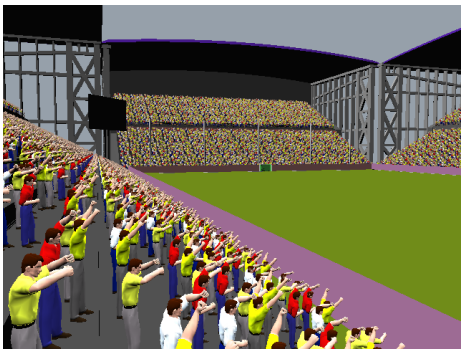
The first example shows a crowd of walking humans, modeled using the Poser animation package [Curious Labs 2001]. Each of the four models consists of 6400 triangles and 9 keyframes. The scene consists of 3 layers of 10×10 instances, leading to 1 million instances. Overall, 6.4 billion triangles are encoded. Precomputation took 628 seconds, creating a precomputed data structure of 153 MB. The rendering results are shown in Figure 109: Using the DirectX implementation (using $k = 64$ points per octree box side), we can render an overview of the scene at about 8 frames per second, corresponding to a vertex throughput of about 24 million vertices per second. This is about 70% of the measured maximum performance of the graphics hardware (see Table 4). Rendering is done using 1-pixel sized opaque splats and prefiltered color attributes. As a result, aliasing and noise artifacts are reduced but not avoided completely. The transition between different resolution levels is also visible, as no interpolation between adjacent hierarchy levels is performed (see e.g. the darker area near the horizon in Figure 109a). As the implementation is mostly geometry limited, not fill-rate limited, we can activate hardware supersampling at small costs (10% less performance). The images in Figure 109 have been rendered using 4× supersampling, removing staircasing artifacts from the triangles rendered in the near field.

	walking crowd	football stadium	landscape with horses
scene complexity	6.4 billion triangles	105 million triangles	42 million triangles
base models	6400 triangles	6400 triangles	18K / 96K triangles
instances	1000×1000 (3 layers)	16416 (2 layers)	1292 / 194 (dynamic control)
keyframes	9	9	15
preproc. time [sec]	628 sec.	1631 sec.	405 sec.
memory [MB]	153 MB	273 MB	166 MB

Table 10: Complexity and preprocessing costs for the benchmark scenes

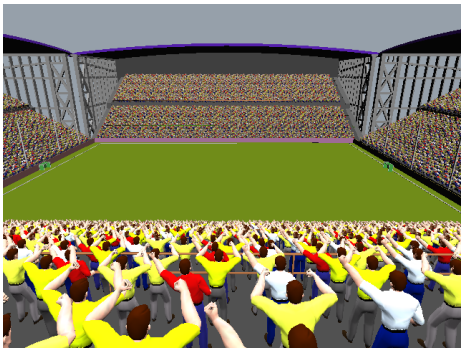


(a) overview (881K points, 223K triangles, 1224 nodes, 20.5fps)



(b) near view (1.7M points, 585K triangles, 3641 nodes, 9.3fps)

(c) close-up (2.6M points, 884K triangles, 5510 nodes, 6.0fps)



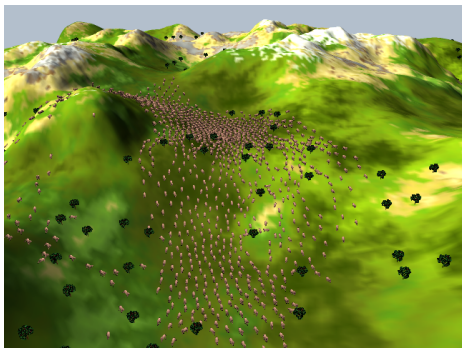
(d) view with large relative depth range (2.1M points, 681K triangles, 4272 nodes, 8.5fps)

Figure 110: Football stadium with 16,416 fans. Rendering using per-pixel reconstruction and prefiltered point attributes (DirectX renderer), 640×480 pixels. The framerates are averages over one animation period.

A large crowd of uniformly walking human models is a rather artificial example. As more realistic application example, we have modeled a football stadium with animated football fans. Such a scene could e.g. serve as background in a computer sports game. The scene consists of 16,416 football fans (again modeled using Poser) with 6,400 triangles each. Neglecting the few triangles of the stadium model, we obtain 105 million encoded triangles. Again, we render the scene using sample points with prefiltered color attributes and per-pixel reconstruction. The resulting animation can be rendered at frame rates of about 6-20 frames per second, depending on the viewpoint. The framerates are near to constant over the complete time interval of the animation.

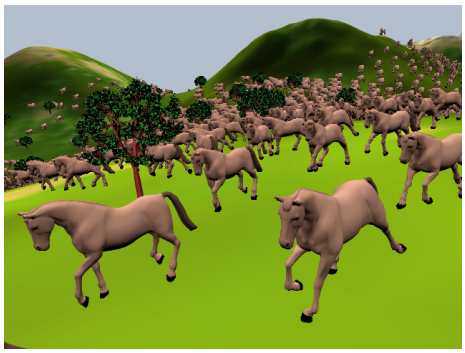
7.4.1.2 Dynamic Instantiation

A more complex crowd animation can be modeled using dynamic instantiation. Typical applications are herds of animals or large crowds of people with a dynamic behavior. Renderings of algorithmically simulated crowd dynamics have recently appeared in many movie productions. One of the first computer generated large scale crowd animation has appeared in the carton movie “The Lion king” [Disney 94]: It features a sequence in which a large herd of buffalos runs through a narrow valley, which would have been to expensive to animate using traditional hand draw cell animation. Later, even feature films such as “The Mummy Returns” [Universal 2001] or the recent “Lord of the Rings” trilogy [New Line 2003] have used computer generated crowd animations to depict the behavior of large crowds (in these examples to depict large scale battle scenes).



(a) overview (prefiltered, per-pixel reconstruction, 485K points, 101K triangles, 4.5K nodes, 8.8fps)

(b) near view (prefiltered, 2×2 splatting, 488K points, 175K triangles, 5.1K nodes, 8.2fps)



(c) close-up (prefiltered, 2×2 splatting, 425K points, 172K triangles, 6.4K nodes, 7.5fps)

Figure 111: Dynamic simulation of a herd of 1292 horses. DirectX renderer, 640×480 pixels. The framerates are averages over 200 frames, including simulation and rendering.

These animations have been computed offline, using sophisticated simulation and rendering techniques, aiming at high-quality results. However, similar techniques could also be of interest for interactive applications, such as computer games. The point-based multi-resolution rendering technique proposed in this thesis could be a first step to enable the usage of such techniques in real-time applications. In order to examine the potential results and performance, we have implemented a simple behavioral simulation for a herd of horses.

The foundation of most contemporary crowd simulations is the “boids” approach proposed by [Reynolds 87]: The behavior of the crowd is computed by assigning local rules to each of its members. The entities in the crowd, which are called “boids”, derived from “bird objects”, have three aims: First, they try to follow the other boids in their neighborhood. Second, they try to avoid collisions with other boids and obstacles. Third, they have a global aim, such as “follow the leader” or “reach that mountain”. These three goals define local forces at each object, leading to a system of ordinary differential equations that has to be solved numerically. The example animation depicted in Figure 111 has been created using a simple version of such a “boids” simulator. It assigns a small local force field to each boid and each obstacle. Additionally, the gradient of a user defined terrain model is considered to penalize climbing of steep hills. To avoid a quadratic runtime complexity for computing the forces, we use a two-dimensional regular grid with lists of objects as search data structure to retrieve the local objects quickly. Numerical integration is performed using a simple Euler rule [Press et al. 95], which is sufficient for this purpose.

The example scene consists of 1292 horses and 194 tree models serving as obstacles, located on top of a fractal landscape. Each horse consists of 18,000 triangles and 15 keyframes, each tree adds 96,196 triangles and the landscape mesh consists of 131.072 triangles, leading to 42 million triangles overall. All elements are rendered using point-based multi-resolution rendering.

Figure 111 shows some example screenshots from the animation. The scene can be rendered at average framerates of about 8 frames per second, depending on the viewpoint. Simulation and scene graph traversal take about half of the overall computation time (46%, measured for the overview viewpoint). The simulation could probably be optimized, allowing for more instances to be controlled in real-time. Rendering performance is currently limited by two issues: First, the hierarchy starts at a resolution of k points per box side length (in our case $k = 16$ for the horses and 48 for the other objects). For instances seen under strong minification, this can lead to oversampling even for the root node. The problem can be solved by adding some additional point clouds with reduced sampling density to the root node. However, this has not yet been implemented for animated rendering. The second problem is the per-batch rendering overhead (this second issue currently even prevents to take advantage of the proposed optimization if it would have been implemented). The submission of a set of point or triangle primitives with the same transformation causes a considerable base overhead per batch. This overhead is a significant limitation of the number of dynamically placed instances in a complex scene. Note that this is an interface problem of current generation graphics hardware [Wloka and Huddy 2003]. At the time of writing, the next generation of graphics hardware (DirectX 9 shader model 3.0) is being announced, which should offer hardware support for instantiation operations: It allows vertex shader registers to be updated at varying frequencies [nVidia 2004b] so that one data stream with instantiation information and one vertex stream with geometry can be used. This enhancement will probably solve our performance problems, as soon as the corresponding hardware becomes available.

7.5 Evaluation of Backward-Mapping Techniques

In this Section, we discuss empirical results of the point-based multi-resolution raytracing technique described in Chapter 6. We try to characterize performance and image quality of the new proposal. We will do so by comparison with the current standard technique, distributed raytracing. We have implemented a distributed raytracing algorithm [Cook et al. 84a] using the same code basis. It uses the same octree (the point-based variant is a subclass) for retrieving triangles and the same basic math routines. Thus, the results are roughly comparable: We cannot definitively judge on the performance relation of highly optimized versions of the algorithms, but at least we obtain an indication for the relative performance.

Some details of the distributed raytracing technique are important for the evaluation: The algorithm uses adaptive oversampling: In a first path, a few rays per pixel (according to a user defined parameter) are shot into the scene, creating an approximate image. This image is used to estimate the variance in an $n \times n$ neighborhood and shot additional rays. The rays are stratified using jittered sampling on a regular subpixel grid. In the end, the set of all ray colors is considered; for each pixel, a weighted average of the neighboring ray colors is computed using a Gaussian filter kernel. This implementation is of course still a bit simplistic (we could use quasi-random grids for stratification, care should be taken to avoid bias in adaptive oversampling, and the user chosen parameters are somehow arbitrary). However, it models the main techniques used in current raytracing approaches and thus should allow a rough comparative performance characterization.

In addition to distributed raytracing, we also compare the algorithm with classic cone tracing. For this comparison, we just use the point-based multi-resolution algorithm and deactivate the usage of point primitives. This creates a cone-tracing implementation for anisotropic ray cones as described by our linear ray model.

In addition to these comparisons, we also take a look at the rendering results for approximate rendering effects supported by our technique such as soft shadows and blurry reflections.

7.5.1 Performance and Image Quality

In order to compare the raytracing techniques with former, well known techniques, we employ two test scenes: The first test scene is a low resolution image, containing only few areas with high frequency details. The scene consists of a low resolution chess board and two spheres (one reflective, one refractive). The second test case is a worst case scenario for conventional raytracing techniques: It contains large areas with structured and unstructured high resolution, high contrast content. The scene shows a large, high frequency chessboard with highly detailed vegetation models placed on it [Lapré 2002]. Additionally, some well known meshes (the Stanford bunny [Stanford 2004] and the well-known cow model; data taken from [Garland 2003]) consisting of reflective and refractive material are placed on the chess board to create complex secondary and higher order rays. All scene details are modeled as geometry, no textures are used. This scene is hard to handle for a distributed raytracing algorithm because the unstructured high frequency content cannot be integrated efficiently using stratification or quasi-random sampling techniques. The outcome of ray samples is mostly random for large areas of the scene, voiding the benefits of stratified sampling techniques. Therefore, we obtain (slow) convergence according to the central limit theorem. Additionally, importance sampling and adaptive oversampling do not help much as the problematic areas cover large regions of the image. The scene is also bad for conventional ray tracing techniques that use extended rays (cone tracing, beam tracing): As the scene consists of highly complex geometry, intersection calculations of large ray cones with the geometry become very expensive, underlining the need for a multi-resolution approach.

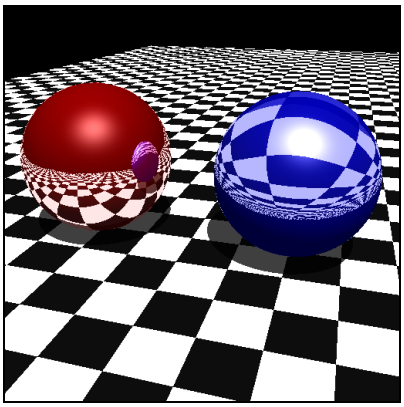
7.5.1.1 Low Complexity Benchmark

We start with the low complexity benchmark. We apply the following candidate algorithms to the scene: Conventional raytracing, point-based multi-resolution raytracing (denoted by PBMR in the following), point-based multi-resolution raytracing with subpixel mask compositing instead of alpha blending (SPM), adaptive distributed raytracing (DRT), and cone tracing. The results are shown in Figure 112. Table 11 shows the required preprocessing times.

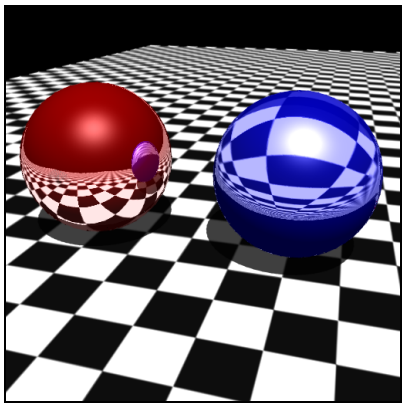
The conventional raytracing image (a) (computed in 30 seconds) contains some aliasing artifacts at borders and in the images obtained by reflection/refraction. Applying the PBMR algorithm removes most artifacts safely, increasing the computation time by a factor of 2.6. Using subpixel masks does not improve the image quality but leads to a longer rendering time. 2×2 subpixel masks create some artifacts at the borders between different objects, for 4×4 masks, the quality is acceptable.

	low complexity scene point hierarchy	high complexity scene point hierarchy	low complexity scene triangle hierarchy	high complexity scene triangle hierarchy
preproc. time [sec]	3 sec.	558 sec.	1 sec.	9 sec.
memory [MB]	0.93 MB	96 MB	0.65 MB	143 MB

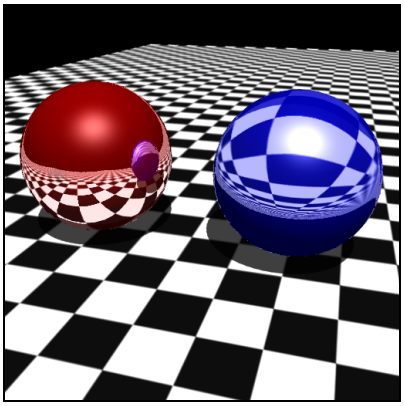
Table 11: Preprocessing costs for the benchmark scenes



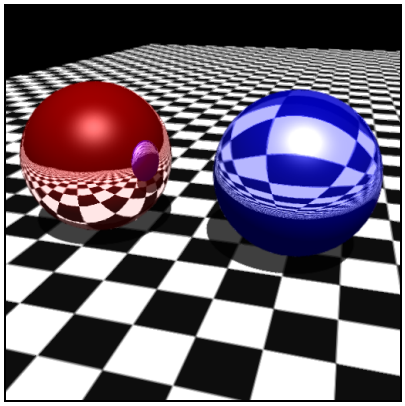
(a) conventional raytracing (30 sec.)



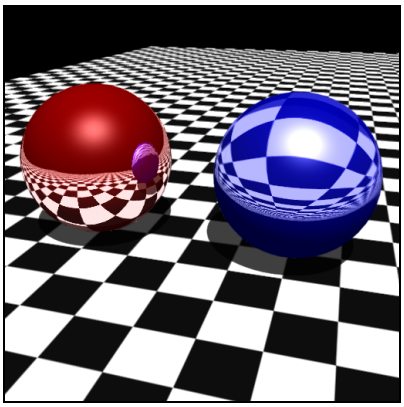
(b) cone tracing (82 sec.)



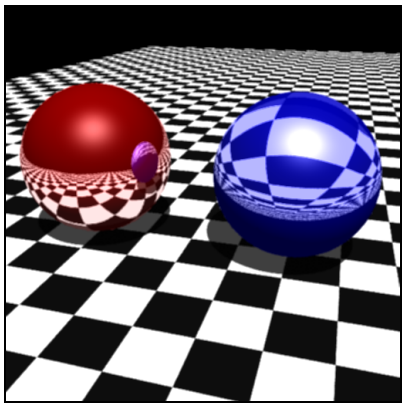
(c) point-based multi-resolution raytracing, alpha compositing (79 sec.)



(d) point-based multi-resolution raytracing, 2×2 subpixel masks compositing (97 sec.)

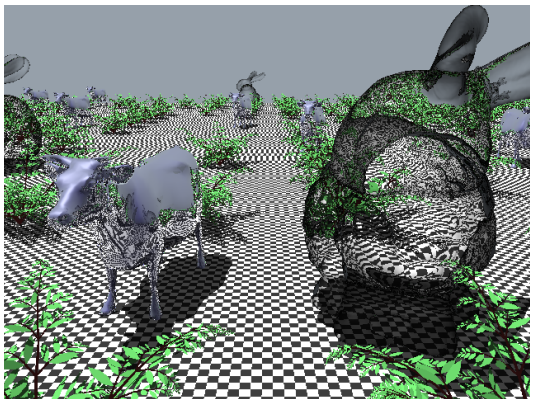


(e) point-based multi-resolution raytracing, 4×4 subpixel masks compositing (104 sec.)

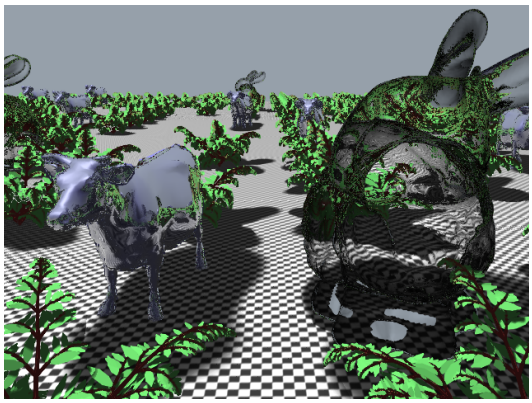


(f) distributed raytracing (94 sec.)

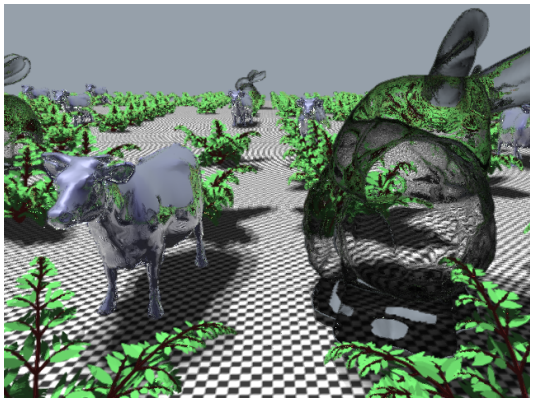
Figure 112: Performance and rendering quality comparison for different raytracing techniques, low complexity benchmark scene (512×512 pixels).



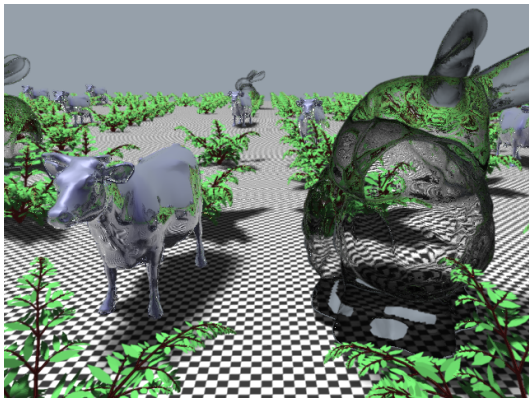
(a) conventional raytracing (215 sec.)



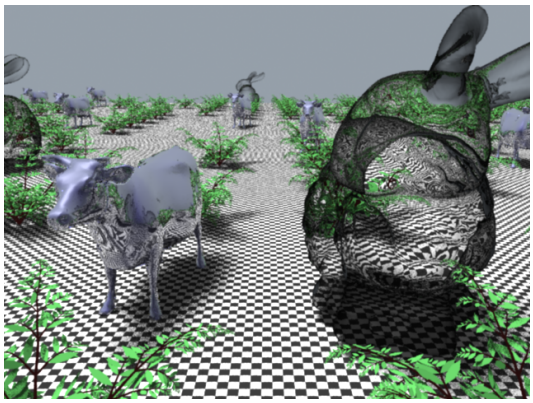
(b) point-based multi-resolution raytracing, alpha compositing (1332 sec.)



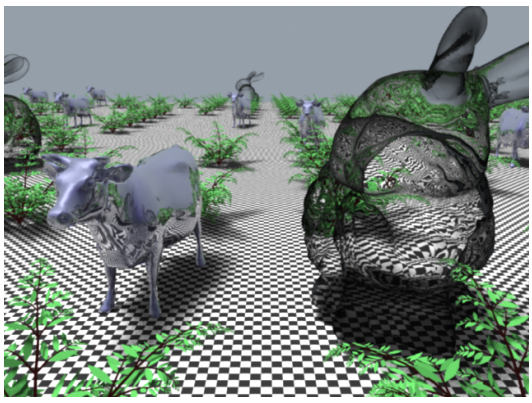
(c) point-based multi-resolution raytracing, 2×2 subpixel masks compositing (4062 sec.)



(d) point-based multi-resolution raytracing, 4×4 subpixel masks compositing (7244 sec.)



(e) distributed raytracing, parameters adjusted for rendering time similar to image (b) (1334 sec.)



(f) distributed raytracing, parameters adjusted for rendering time similar to image (d) (6961 sec.)

Figure 113: Performance and rendering quality comparison for different raytracing techniques (640×480 pixels).

Using the PBMR algorithm without a point hierarchy (i.e. performing cone tracing) leads to a slightly increased rendering time. However, for low complexity scenes, the advantage of the multi-resolution technique is rather small. The distributed raytracing algorithm produces a reference solution that is mostly identical to the PBMR solution; only some minor noise artifacts in the DRT solution remain visible. We have chosen the oversampling parameters to obtain a rendering time similar to that of the PBMR and SPM renderings.

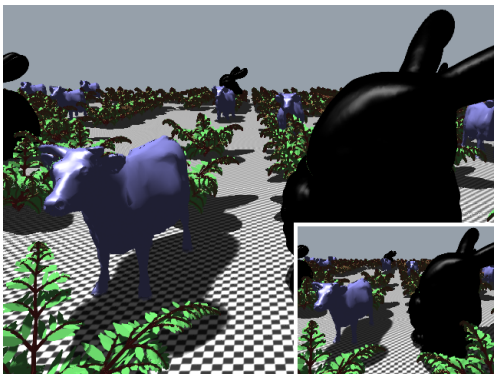
As a result, the different antialiasing techniques lead to similar results for a low complexity scene, both in terms of image quality and rendering time.

7.5.1.2 High Complexity Benchmark

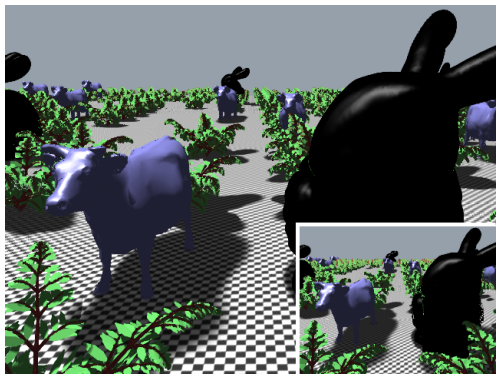
Figure 113 shows the rendering results for the high complexity benchmark scene. As expected, the conventional raytracing rendering shows severe aliasing artifacts. Point-based multi-resolution raytracing takes 6 times longer but removes most aliasing artifacts[41]. We have also computed a distributed raytracing image using oversampling parameters to match the rendering time of the PBMR image as close as possible (it has not been possible to find a perfect match due to the large running times, making parameter experiments quite expensive). Comparing the two results (image (b) and (e)), two differences are apparent: First, the PBMR-solution leads to a strong overestimation of the silhouette opacities of the objects. In contrast to the previous low complexity example, this scene contains many fine granular silhouette details so that the effect predicted in Section 6.2.7 now becomes obvious. This effect does not only exaggerate the opacity of object borders. It is especially unpleasant for secondary rays: The shadows from the area light source are strongly overestimated. The second observation is the remaining noise in the DRT-solution: Using roughly the same amount of computation time as the PBMR-algorithm, the adaptive DRT-algorithm is not capable of removing the noise artifacts from the image. In the previous example, only a few border regions contained high frequency details so that adaptive ray sampling permitted a sufficient oversampling in problematic regions. For the high complexity benchmark scene, this is not possible to the same extend. The PBMR technique with alpha compositing offers good antialiasing of inner regions of a complexly structure objects. Secondary rays are handled correctly. However, in border regions, the foremost object borders are overestimated and the reproduction of distribution raytracing effects such as soft shadows is not yet satisfactory. An additional artifact of the PBMR technique becomes visible at the bottom of the foremost bunny model: In the point-based images, a hole appears in regions where the transparent bunny and the floor meet. This is a problem of the surface merging strategy: A front-facing and a back-facing surface are close to each other so that they are merged to one fragment with wrong normal direction so that it is culled[42]. Such artifacts could be avoided by a refined merging criterion, e.g. taking into account the normal directions of the fragments. Additionally, the PBMR-image also shows some black spots at the border of the reflective cow object which are due to early intersections of the outgoing ray with the emitting surface.

To overcome the silhouette overestimation problems, we have combined the alpha-blending heuristic [Zwicker et al. 2001a] with subpixel masks [Carpenter 84], as described in Section 6.2.7. The results for 2×2 and 4×4 subpixel masks are shown in Figure 113(c) and (d), respectively. The results are not equivalent to the reference DRT-solution (Figure 113f) but the image quality is acceptable, at least for the 4×4 version. The early-intersection artifacts are also reduced (compare the back of the reflective cow in image b and d) as the coverage of the ray is determined more

[41] Some minor artifacts are still visible, especially on the floor, because the performed band-limiting is always a compromise between base-band attenuation and remaining aliasing.
[42] Inside refractive objects, the orientation for backface-culling is reversed.



(a) Cone tracing, primary and shadow rays only. (640×480: 1427 sec., 256×192: 709 sec.)



(b) Point-based multi-resolution raytracing, primary and shadow rays only. (640×480: 478 sec., 256×192: 86 sec.)

Figure 114: Comparison with classic cone tracing, primary and shadow rays only. Handling reflections and refractions using classic cone tracing is practically infeasible for this scene.

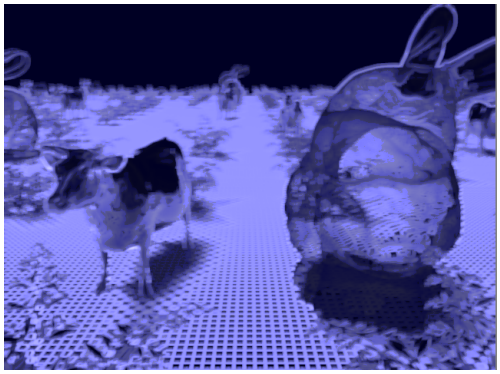


Figure 115: Variance map used to control the oversampling in Figure 113e, f.

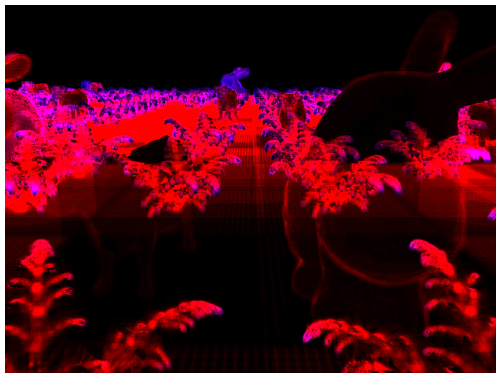
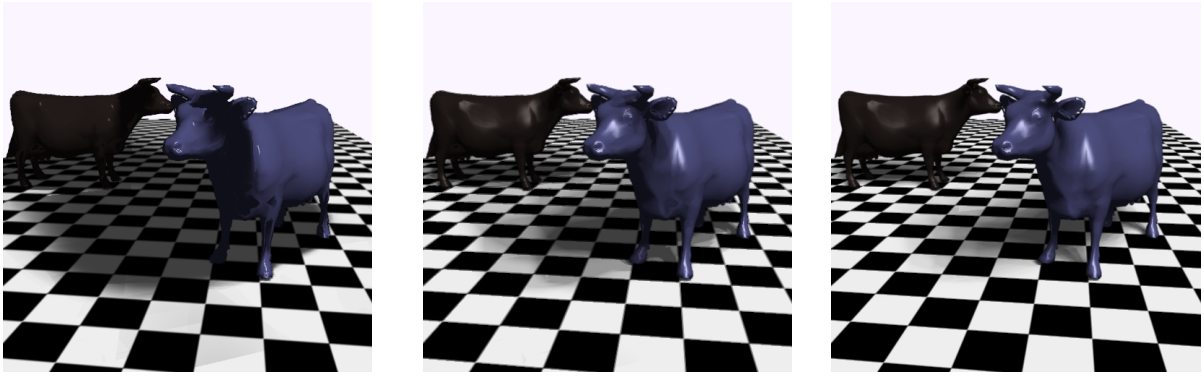


Figure 116: Raytracing costs (Figure 114, primary rays only). Red = triangle intersection tests, blue = point intersection tests. Full intensity corresponds to 2000 test.

accurately. The subpixel masks technique also has some drawbacks: First, some aliasing artifacts are reintroduced, although being rather weak for the 4×4 version. The remaining aliasing artifacts could be reduced by adjusting the filter support of the sub-pixels. Choosing this value is a trade-off between silhouette overestimation and aliasing. Additionally, the rendering time is increased due to the overhead of the subpixel computations (3× for the smaller and 5.4× for the larger subpixel masks).

Please note that using subpixel masks is still much cheaper than brute-force supersampling at a higher resolution (i.e. 4×/16× more sample rays) as a repeated search for intersections is avoided. Nonetheless, the additional costs are significant. Thus, we have computed a second DRT-solution, now allowing (roughly) the same rendering time as used for the SPM-4 solution. As expected, this solution is much better in terms of remaining noise artifacts than the version that used a fifth of the sampling rays. However, the image still contains visible noise artifacts. For a still image this may be of minor importance, but in animations, such artifacts are still distracting. Therefore, an application of the point-based raytracing technique with subpixel masks may be useful in such cases.



(a) alpha blending compositing (65 sec.)

(b) 2×2 subpixel masks (109 sec.)

(c) 4×4 subpixel masks (131 sec.)

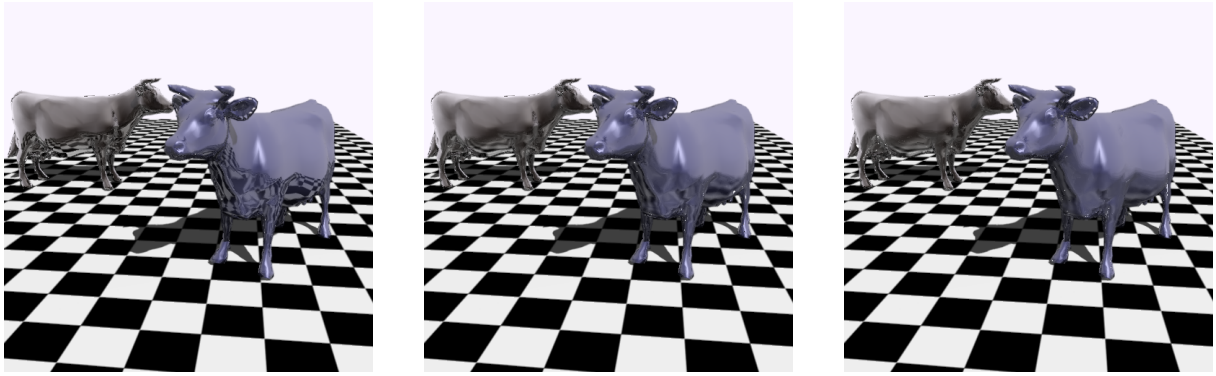
(e) 8×8 subpixel masks (204 sec.)

(f) reference solution: distributed raytracing (259 sec.)

Figure 117: Soft shadows (512×512 pixels).

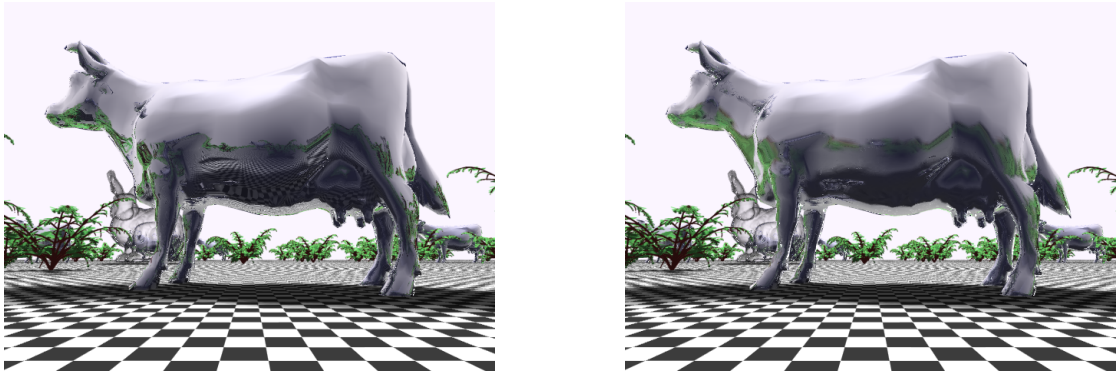
In conclusion, the proposed novel raytracing technique is able to render images of acceptable quality. Even in adverse scenes, it yields scenes with little noise and aliasing artifacts. For a faithful reproduction of border regions, subpixel masks must be employed. The costs are then rather high so that the performance advantage over distribution raytracing techniques is not that large. However, a lot of optimizations seem to be possible to avoid the cost penalties. First, the subpixel mask rendering code itself could be optimized more thoroughly. Additionally, we can think of using it adaptively, i.e. increasing the number of subpixels if borders are detected. Using such optimizations, the rendering costs probably become more competitive.

After comparing the novel technique with distribution raytracing, we compare it with classic cone tracing. Here, the results are obvious: When trying to render the scene using classic cone tracing, the algorithm spend several minutes at the first few pixels of the refractive bunny. Even after waiting a considerable amount of time, no progress was visible so that the computation had to be interrupted. The secondary rays created by reflection and refraction are typically broadening rapidly so that an enormous number of triangle intersection tests becomes necessary. As a result, cone tracing of reflections and refractions in complex scenes is practically infeasible. Thus, we have restricted the comparison to primary rays only. In this case, the algorithm is able to finish rendering an image; however, the rendering time is much longer (Figure 114a,b). The ratio becomes worse if the resolution is reduced: The PBMR technique shows an output-sensitive rendering time due to the multi-resolution approach; the rendering time is roughly proportional to the image resolution. For classic cone tracing, the rendering time does not decrease as strongly as



(a) ideally specular reflections (148 sec.) (b) blurry reflections (176 sec.) (c) more blurring (186 sec.)

Figure 118: Blurry Reflections (512×512 pixels).



(a) ideal specular reflection (2652 sec.) (b) blurry reflection (2814 sec.)

Figure 119: Blurry Reflections in a more complex scene (cf. Figure 113, 640×480 pixels). In the left image, it can be seen nicely that linearly interpolated normals lead to piecewise constant second derivatives, i.e. constant footprint increments and thus a constant resampling filter on each triangle. In the right image, the ray footprint increments have been increased to simulate blurry reflections.

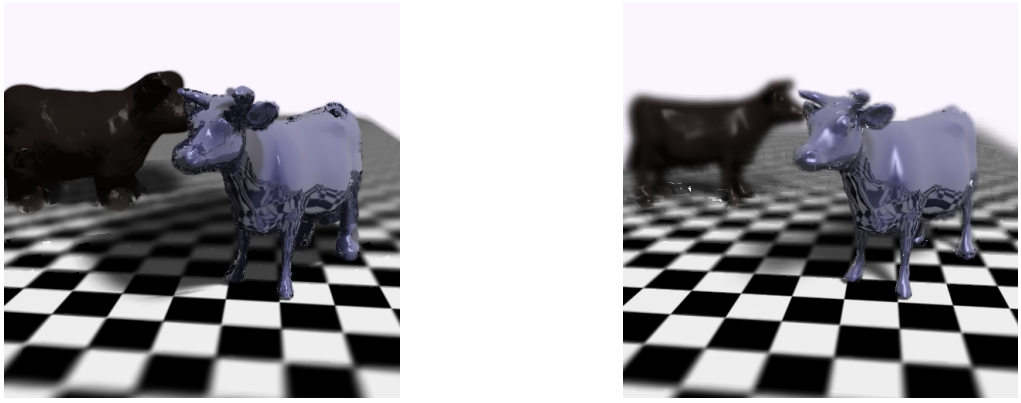
the number of triangle intersection tests remains roughly constant. The speed-up for using a point hierarchy is a factor of about 3 at 640×480. At 256×192, the speedup factor is already more than 8. Correspondingly, we expect larger speedups for more detailed scenes. For our benchmark scene, rendering using primary rays only at 640×480 pixels still uses mostly triangles, only a few highly detailed areas employ point based replacements (see Figure 116).

7.5.2 Special Effects

The extended ray volumes of the multi-resolution raytracing approach also allow for an approximation of typical distributed raytracing effects such as soft shadows, blurry reflections, and depth-of-field. In this section, we will examine the rendering quality obtained in practice.

7.5.2.1 Soft Shadows

Soft shadows are an important rendering effect for realistic image synthesis: In natural scenes, usually no sharp shadow boundaries are observed. The extended ray cone model allows an approximation of soft shadow effects by shooting rays with increasing footprint diameter towards



(a) alpha blending compositing (263 sec.) **(b) 8×8 subpixel masks (793 sec.)**

Figure 120: Depth of field (512×512 pixels).

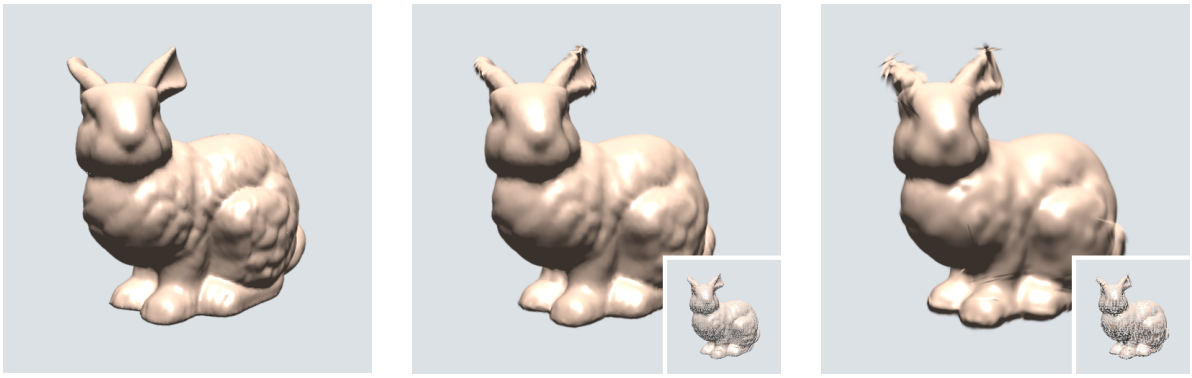
the light sources. Figure 117 shows the results: Using the PBMR algorithm with alpha compositing creates images with strongly overestimated shadow bounds. Thus, subpixel masks are mandatory for a good image quality. The same effect can also be seen in Figure 113; the scene also contains soft shadows. Using subpixel masks with 2×2 subpixels, the overestimation can be reduced. However, now, discretization artifacts are visible. Using 4×4 subpixel masks already yields a good image quality and 8×8 subpixels create an image with very little artifacts. In comparison to distributed raytracing (Figure 117f), the shadow region is still overestimated a bit but this should be no problem for most applications. The distributed raytracing solution takes more computation time than the subpixel mask rendering, but still shows substantial noise artifacts[43].

7.5.2.2 Blurry Reflections

A second effect is blurry reflections: By increasing the increments of the reflected ray cones, semi-diffuse surfaces with blurry reflections can be simulated. Figure 118 shows an example of a test scene: Figure 118a shows the original ideally specular reflections, without additional blurring. Figure 118b and c show the results for increasing the ray increments. Figure 119 shows the same method applied to the more complex scene of Figure 113. The results appear qualitatively correct (some aliasing is left on the floor; again, the sub-pixel filter radius for the 4×4 subpixel masks has not been adjusted optimally). The rendering time does not decrease for increasing blurriness but increases slightly: This is behavior by design, as the multi-resolution algorithm tries to keep the number of primitives that are tested for each ray constant. In low resolution areas, this can even increase the rendering time, as a single triangle is less expensive than an (at most fixed sized) approximating point set of a larger ray cone. The blurring technique also has limits: If the increase of the rays become too large (more than 30% per unit length, which is already very blurry), the algorithm tends to yield self-intersections with the emitting surfaces, leading to artifacts in the image. To prevent theses issues, better heuristics to avoid self intersections are necessary.

In Figure 119a, an interesting observation (apart from blurry reflections) can be made: For ideal specular reflections, the ray increments are only determined by the local surface curvature. As we use normal interpolated triangles with linearly interpolated normals, the normal derivatives are constant for each triangle. Thus, we obtain discontinuities in the “blurriness” of the re-

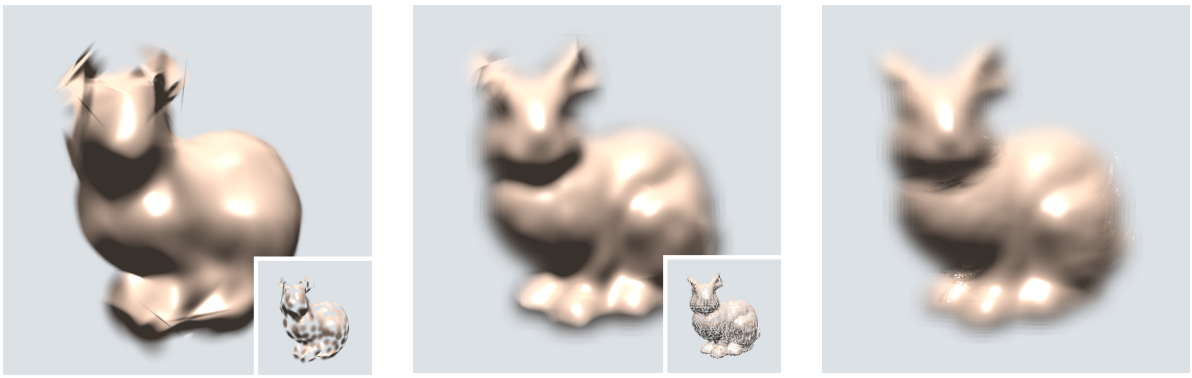
[43] It should be noted that pure random sampling has been used for distributed raytracing of area light sources. The convergence rate could be enhanced by using stratification.



(a) rendering with pixel resolution (23 sec.)

(b) simplified to 10 pixels sized sample points (64 sec.)

(c) simplified to 20 pixels sized sample points (60 sec.)



(d) simplified to 50 pixels sized sample points (52 sec.)

(e) blurred rendering with ray footprint according to 20 pixels (703 sec)

(f) blurred rendering with ray footprint according to 20 pixels, but using data at 1 pixel resolution (3961 sec.)

Figure 121: Point-based simplification – rendering results for varying parameters for the ratio between ray diameter and point spacing (512×512 pixels). The small pictures at the lower right show the underlying point cloud representation (rendering with shrunk local filter support).

flections at triangle edges as well as first order discontinuities in the mirrored images. In order to obtain a better quality for the reflections, a higher order normal interpolation scheme should be used. This is a modeling problem rather than a restriction of the raytracing approach.

7.5.2.3 Depth-of-Field

Extended ray cones can also be augmented to approximate depth-of-field effects (Section 6.2.4). Figure 120 shows the result: Rendering such partially blurred images with alpha blending compositing leads to unacceptable images. The silhouettes are strongly overestimated. Thus, the objects appear geometrically thickened instead of out-of-focus. With subpixel masks, this artifact can be avoided. However, a large number of subpixels are necessary to obtain acceptable results. Accordingly, the rendering times are rather long.

7.5.2.4 Varying Level-of-Detail / Visual Convergence

An additional operation provided by point-based multi-resolution raytracing is simplification of objects by controlling the level-of-detail used for rendering. This feature is unlikely to be used as

rendering effect. Nevertheless, it is interesting to examine the simplification results as they show how the algorithm adapts the resolution during rendering.

Figure 121 shows rendering results for different ratios between point spacing and (minimum) ray diameter. As the model shows a smooth mesh (the Stanford bunny [Stanford 2004]), we can perform quite strong simplification (increasing the sample spacing by a factor of 10) while obtaining only moderate artifacts. Most artifacts occur at sharp edges (the ears) that cannot be modeled adequately using flat (differential) sample points. The rendering time does not decrease significantly for the simplified renderings; indeed, rendering becomes slower when starting to simplify the mesh. Again, this is due to the fact that the algorithm tries to use a similar number of primitives for each ray, independent of its diameter. For high resolutions, the number of accessible primitives is bound by the leaf node triangles, yielding shorter rendering times. The motivation for simplifying models is not to speed up rendering. Instead, we use lower resolution versions to represent renderings with larger ray footprint. Thus, usually not the simplified model itself (such as Figure 121c) is shown but a version blurred due to a larger ray footprint (Figure 121e). Rendering the blurred version is more expensive than rendering the simplified model. As the ray footprint is enlarged additionally, more primitives have to be tested per ray. Nevertheless, using a simplified model is much faster than rendering at the original level-of-detail (Figure 121f). Although both renderings look mostly similar, the rendering at the original resolution takes considerably more time.

Chapter 8
Extensions

In this chapter, we describe some extensions to the basic rendering techniques described before. The first two subsections describe techniques that have been developed in cooperation with colleagues from University of Tübingen and University of Paderborn. The author of this thesis was only involved in the design of the algorithms and data structures but not in the implementation. Therefore, only a brief overview of the main ideas is given here. For details, please refer to the corresponding publications [Guthe et al. 2002, Klein et al. 2002]. The next two subsections describe techniques developed by the author of this thesis, but which do not deal with multi-resolution image rendering techniques. These topics are included to show the applicability of the developed concepts to other problems in computer graphics, not dealing with multi-resolution rendering of surface models. Again, please refer to the corresponding publications [Wand and Straßer 2003b, Wand and Straßer 2003c] for details.

8.1 Volume Rendering

8.1.1 Overview

Efficient rendering of complex surface models, as discussed up to now, is an important issue in computer graphics. However, there are several disciplines that have to deal with large three-dimensional data sets with different characteristics. An important problem in scientific visualization is the visualization of volumetric data sets. In this section, we will focus on scalar data sets sampled on a regular grid, i.e. large three-dimensional arrays of scalar values. Such data sets occur for example frequently in medical imaging (tomography scanners, CT/MRT/PET) or as a result of physical simulations (e.g. using finite differencing approaches).

The problem of regularly sampled volume data sets is their sheer size: For example, modern computer tomography scanners offer resolutions of more than some thousand voxels squared, creating data sets of several gigabytes. Using traditional visualization techniques that inspect the complete data set for rendering, it is not possible to display such scenes in real-time. A multi-resolution rendering strategy similar to the techniques proposed for surface rendering can potentially avoid the complexity problems. Multi-resolution rendering has been applied to volume rendering by several authors, see Section 3.2.1.5 for details. The technique proposed in this section

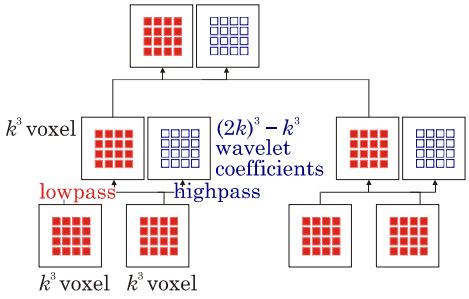


Figure 122: Constructing a volume octree analogous to point hierarchies using wavelet filters.

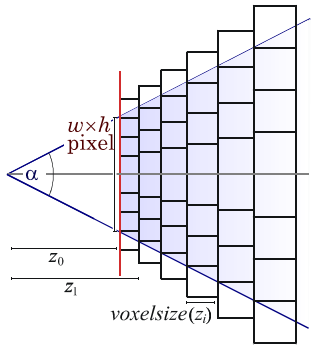


Figure 123: Analysis of the rendering costs (similar to Figure 45)

combines an efficient texture mapping-based multi-resolution approach [LaMar et al. 99, Weiler et al. 2000] with an object space error metric [Boada et al. 2001].

The first problem for rendering large volume data sets is the efficient representation of the data set. Of course, a multi-gigabyte data set does not fit into the main memory of conventional PCs. In order to represent the data more efficiently, a lossy wavelet compression is employed. This encoding scheme offers both a strong reduction of the memory requirements (about 1:30 without visible artifacts) and a spatial hierarchy needed for multi-resolution rendering. The wavelet hierarchy is constructed as follows: First, the data set is divided into cubes of k^3 voxels (usually $k = 16$). Then, groups of 8 adjacent blocks become siblings: A parent node for these 8 nodes is constructed by applying a wavelet lowpass and highpass filter to the group of 8 blocks. The lowpass filtered data is represented again by k^3 voxels, the additional $(2k)^3$-k^3 values of the children are represented by wavelet coefficients obtained by applying the highpass filter to the original data. This scheme (Figure 122) is carried on hierarchically until only one root node is left, yielding an octree representation (null-pointers are used in areas where not all 8 children are covered with data). Each block of highpass coefficients is encoded using entropy and redundancy coding, creating a very compact representation. Even large data sets, like one of the well-known visible human data sets [National Library of Medicine 2002], can be stored completely in the main memory of a conventional PC.

This data structure is used by the rendering algorithm similarly to static point-based multi-resolution rendering [Algorithm 3]: The algorithm traverses the hierarchy downwards, starting at the root node. At each node, it checks whether the node is contained in the view frustum. If not, the node is skipped. Otherwise, the algorithm checks the projected size of the contained voxels. If it is still larger than a single pixel, the traversal is continued at the children in order to search for a more detailed representation. After all octree nodes with matching resolution are determined, their content is converted to OpenGL texture slices and rendered by drawing alpha blended slices in back to front order.

Whenever the algorithm wishes to access the children of a node, a wavelet decompression is necessary. Additionally, a texture object has to be created for each node to be drawn on the screen. Performing these computations from scratch for every frame is quite expensive. Thus, a caching scheme, as discussed in Section 4.2.6 is employed: Both decompressed data and textures are stored in separate LRU-caches. Additionally, the graphics driver decides to upload a working set of recently used textures to the graphics board, creating a third caching level. For data sets that do not fit into main memory, this scheme could easily be extended by caching nodes that have been swapped in from harddisc. However, this has not yet been implemented.

8.1.2 Analysis and Consequences

How efficient is this rendering procedure? In order to find out whether the proposed technique will be capable of rendering large data sets on conventional PCs, we perform a formal analysis of the costs of the algorithm. It will turn out that the technique described so far already shows a good asymptotical behavior; however, it must still be augmented to be applicable to real-time rendering on the desired platform.

The structure of the proposed algorithm is similar to that used for point-based multiresolution rendering (forward mapping). Therefore, the same analysis can be applied (Section 5.2.3): The view frustum is covered with $O(\log \tau)$ boxes where each box leads to constant costs (with τ being the maximum relative depth range of the scene). Thus, we expect rendering costs of $O(\log \tau)$ for a data set with an arbitrary number of voxels, being output-sensitive. For the case of regularly sampled volume data sets, the (worst case) costs can also be expressed in terms of voxels (similar to [Chamberlain et al. 96]). Additionally, we can also count the voxels used by the rendering algorithm in order to predict its performance:

For the quantitative analysis, we chose a z-distance of the image plane so that the voxel resolution of the original data set and the pixel size match up (Figure 123). We fix an image resolution of $w \times h$ pixels and a vertical viewing angle of α. At first, we assume that the volume can be resampled to arbitrary voxel spacing, always allowing an exact voxel spacing of a single pixel. We cover the image plane at depth z_0 with $w \times h$ resampled voxels (these voxels have the same spacing as in the original data set). Then, we subsequently add more layers of (resampled) voxels at depths z_i, with a voxel spacing projecting to the pixel spacing on the screen. The depth z_i of voxel layer i is given by the recurrence:

$$z_{i+1} = z_i + voxelsize(i)$$

with

$$voxelsize(i) = \frac{\tan\frac{\alpha}{2}}{\frac{h}{2}} z_i$$

(see Figure 123). Solving the recurrence, we obtain

$$z_i = z_0 \underbrace{\left(1 + \frac{\tan\frac{\alpha}{2}}{\frac{h}{2}}\right)}_{=:q}{}^i = z_0 q^i \quad \text{with} \quad q := \left(1 + \frac{2\tan(\alpha/2)}{h}\right)$$

With z_{far} denoting the largest depth in the data set relative to the viewer, we obtain a bound for the number of voxel layers of

$$layers_{max} = \frac{\log\left(z_{far}/z_0\right)}{\log q}.$$

Currently, we have only considered the voxels behind the plane at which the voxel resolution matches the display resolution. Voxels in front of that plane cannot be adapted to the screen reso-

lution as the maximum level of resolution is already reached[44]. Therefore, we do not find more than $w \cdot h \cdot \cot(\alpha/2) \in O(1)$ voxels in this area (using $z_0 = \cot(\alpha/2)$).

In order to limit the ratio z_{far}/z_0 by the size of the data set, we consider the ratio

$$R(z) = \frac{z + \Delta z}{\max(z, z_0)}$$

with Δz denoting the depth of the data set in world coordinates. It describes the relative depth range for varying distance to the data set. In case of $z \geq z_0$, we obtain a derivative of

$$R'(z) = \frac{-\Delta z}{z^2},$$

being strictly negative. Thus, the largest ratio is found at $z = z_0$. For $z < z_0$, choosing $z = z_0$ obviously maximizes the ratio $R(z)$. This means that the maximum relative depth range is obtained if the data set is located at a distance of z_0 and oriented to yield a maximum depth difference Δz. If we assume a cube voxel data set of n^3 voxels, we know that Δz (and thus also z_{far}) cannot be larger than $\sqrt{3}n$ voxels. The size of a voxel in world coordinates is *voxelsize*(0), i.e.

$$voxelsize(i) = \frac{2\tan(\alpha/2) \cdot \cot(\alpha/2)}{h} = \frac{2}{h}$$

This limits the ratio z_{far}/z_0 to

$$\frac{z_{far}}{z_0} = \frac{2\sqrt{3}n}{h\cot(\alpha/2)} = 2\sqrt{3}\frac{n}{h}\tan(\alpha/2).$$

Hence, the number of voxels in the view frustum cannot exceed:

$$v_{max} = w \cdot h \cdot \left(\frac{\log\left(2\sqrt{3}(n/h)\tan(\alpha/2)\right)}{\log\left(1 + 2\tan(\alpha/2)/h\right)} + \cot(\alpha/2) \right) \tag{17}$$

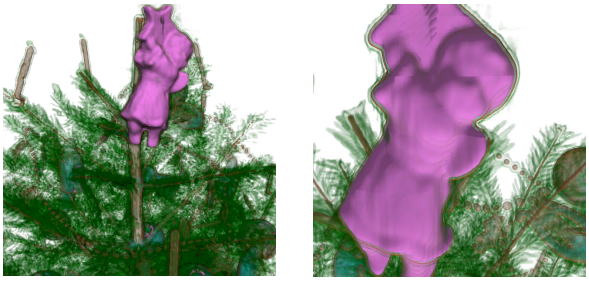
Overall, we obtain a number of voxels of $O(\log n)$ for a cube voxel grid of n^3 voxels. This estimation does not yet consider the underlying octree structure. The octree has three side effects: First, we can choose the resolution only in powers of two. Second, at each resolution level, boxes must be aligned at a grid of powers of two (corresponding to the root cube). Third, each box contains a grid of k^3 voxels, leading to an approximation of the ideal sampling density. These are the same effects as discussed for point-based multi-resolution rendering: As described in Section 4.2.3.5, choosing the resolution in powers of two only leads to an average oversampling of

$$\int_1^2 s^3 ds = 3\frac{3}{4}$$

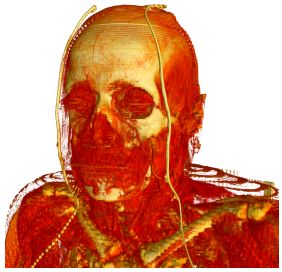
in the volumetric case. The alignment of boxes to a grid in powers of two does not change the asymptotic complexity (see Section 5.2.3). The quantitative overhead cannot easily be quantified. However, for small values of k (the number of voxels per octree box), the octree boxes are small in respect to the size of the regions of constant resolution (created by a voxel spacing that can only change in powers of two). Therefore, we expect only a small overhead. The overestimation of the projection factor[45] leads to the same maximum overestimation as analyzed in Section 5.2.3.2, and

[44] In contrast to surface rendering, we do not need a near clipping plane to limit the rendering efforts (due the finite voxel resolution).

[45] Please note that the approach described here again tries to approximate the depth factor of the perspective projection.



(a) Images from a walkthrough of the "Christmas-tree" data set[46] ($512^2 \times 999$ voxels, 12bit). The average frame rate is 9.1 fps (viewport size 256^2).



(b) Image from a walkthrough of the "visible human female" CT data set ($2048 \times 1216 \times 1877$ voxels, 12bit) [National Library of Medicine 2002]. The average frame rate is 4 fps at a viewport size of 800^2 pixels.

Figure 124: Example renderings of the multi-resolution volume rendering system.

the bounds for view frustum overestimation (Section 5.2.3.3) and overestimation of the distortion factor (Section 5.2.3.4) hold, too.

Putting these results together, we can estimate the number of voxels from the multi-resolution hierarchy that is needed for rendering a typical data set. According to our applications, we assume a view port size of 640×480 pixels, 60° viewing angle and a data set of 2048^3 voxels. Rendering the original data set would require processing 8 billion of voxels. The example case leads to a maximum relative depth range z_0/z_{far} of 8.53, yielding 892 voxel layers behind z_0. According to equation 17, we obtain 275 million voxels. For a smaller viewport of 320×200 pixels, we still obtain 46 million voxels. Now we take into account that we can only perform an approximate resampling using octree boxes of $k = 16^3$ voxels. The bounds for the overhead of the octree-based multi-resolution approach (see Table 2) yield an additional oversampling factor of about 1.1 (depth) $\times 1.2$ (view frustum) $\times$ 1.03 (distortion) $\times$ 3.75 (discrete resolution levels) = 5.1 (factor 7.4 for the 320 $\times$ 240 view port), creating up to 1.4 billion (340 million for the small view port) voxels[47].

This brief analysis shows two important aspects: First, the multi-resolution algorithm will scale very well, the rendering time grows only very weakly for larger data sets. However, the constants in the "$O(\log n)$"-behavior are considerable: Even for small display resolutions, the algorithm will not be able to run in real-time on conventional PC-hardware. Note that contemporary PC graphics boards offer local video memory of up to 256MB, which is less than the demanded number of voxels to process. Even with enough memory, the processing costs for alpha blending some hundred million voxels are too large.

Due to this analysis, the design of the algorithm has been augmented: Instead of performing simple depth-based multi-resolution rendering only, an object space error metric has been included. Many data sets contain large regions of empty space or at least regions with little variation, i.e. low frequency content. Thus, we adapt the resolution to the frequency spectrum of the volume in addition to the projected size: In the precomputed wavelet-hierarchy, the average difference (according to a suitable norm) between the values in an inner node and the values at

---

[46] "*The CT-dataset XMasTree was generated from a real world Christmas Tree by the Department of Radiology, University of Vienna and the Institute of Computer Graphics and Algorithms, Vienna University of Technology.*" The data set is available at: `http://ringlotte.cg.tuwien.ac.at/datasets/XMasTree/XMasTree.html`

[47] The estimated overhead factors for the projection factor and view frustum are only conservative bounds (see Section 5.2.3). Nevertheless, the overhead is dominated by the large factor due to the discrete (power of two) resolution levels anyway. This factor could be reduced by creating intermediate resolution levels, e.g. by resampling decompressed data on-the-fly to one or two coarser levels of resolution. This has not yet been implemented.

maximum resolution indicate the amount of high-frequency content that has not been represented faithfully. Thus, we form a composite metric of projective foreshortening and frequency content by multiplying both error estimates. The hierarchy traversal is then guided by this error metric. Instead of the simple depth-first search, we employ a greed algorithm that collects subsequently nodes with the worst error from the hierarchy using a priority queue. When the (user defined) maximum number of voxels is reached, the traversal is stopped. This modification leads to a considerable reduction of processed voxels while maintaining a comparable image quality. As the distribution of high frequency regions throughout the volume is very uneven in most data sets (only a few regions contain highly detailed data), we are able to cut the rendering costs substantially, allowing an interactive visualization on conventional PC hardware.

8.1.3 Implementation and Results

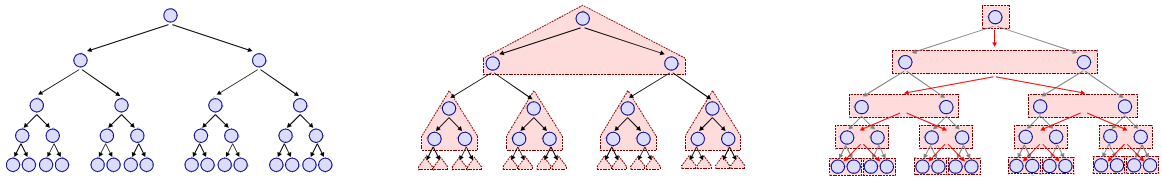
Using the modified multi-resolution algorithm, we were able to render large data sets such as the visible human data sets (6.5 GB) or the Christmas tree data set (375MB) at interactive framerates. Figure 124 shows two examples rendered using a 2.6GHz Pentium 4 PC equipped with 2GB Rambus memory and a Radeon 9700Pro graphics board. The data sets have been rendered applying a classification function $[0, 1] \rightarrow \text{RGBA}$ to the scalar data. The classification has been taken into account for weighting the frequency content. A 30% performance gain is obtained by additionally applying an occlusion culling heuristic. For more details, see [Guthe et al. 2002, Guthe and Straßer 2004] or Stefan Guthe's PhD thesis [Guthe 2004].

8.2 Out-of-Core Storage

8.2.1 Overview

Up to now, we have used hierarchical instantiation to encode scenes of high complexity for point-based surface rendering. This is not possible in all applications. Often, no instantiation structure is known, in some cases the data is irregular so that no redundancy can be exploited (at least not easily). This is a typical problem in scientific visualization, where we have to deal with measured data. In such cases, we need an alternative strategy to cope with the restrictions of main memory. One option is to store the data set on an external memory device such as a harddisc or probably even an array of such devices. In this section, we discuss how the data structure for point-based multi-resolution rendering can be generalized to support our-of-core storage.

The main problem of out-of-core storage is due to latency times for random access. Typically, accessing a random position in a file of data is more expensive than sequential access by several orders of magnitude. In addition, a sequential transfer of data is (usually) still substantially slower than performing the same operation in main memory. Therefore, it is common to use an augmented cost model to analyze the performance of *external-memory* algorithms (see e.g. [Silva et al. 2002] for a detailed survey). First, we have to count the number of random access operations an algorithm has to perform; second, we consider the amount of data that is transferred from the external device. In many cases, processing costs in main memory are of minor importance.



(a) original tree (b) local subtrees (c) neighbors

Figure 125: Blocking schemes for out-of-core storage.

8.2.2 Modifications to the Static Sampling Data Structure

Given these problems, we immediately see that the dynamic sampling data structure is not well-suited for external storage: The algorithm mostly performs nothing but random access operations, degrading the performance seriously in external settings. However, we can generalize the static sampling approach. To do this, we first need an augmented construction algorithm with sufficient external efficiency.

The construction algorithm reads the data linearly but it uses multiple passes in which the complete input data is re-read. Thus, the number of these passes is decisive for its performance. The idea of the algorithm is based on *spatial partitioning,* which is a well known paradigm for out-of-core processing [Silva et al. 2002]. We assume that the input triangles are given as one large file with randomly ordered triangles. In a first pass, the algorithm reads all triangles, counts them and computes a bounding box for them. If the number of triangles is small enough to be processed in-core, we use the conventional construction algorithm as discussed in Section 4.2.2. Otherwise, we create a map in main memory that describes the distribution of the primitives in the scene in a second pass. This map is a three-dimensional array of cells within the bounding box of the scene. Each cell stores an integer counter for the number of primitives that intersect with that cell. The map should be made as large as possible, using as much main memory as possible. If we know a priori that the primitives are not randomly distributed within the bounding box but e.g. only on a surface, we can also replace the array with a hash table. For a surface like structure, only $O(k^2)$ of the potential k^3 cells are occupied. Thus, a much larger map can be fit into main memory.

After the distribution of the primitives has been determined, the corresponding cells are sorted into an octree. Leaf nodes of this octree are sequentially merged into inner nodes. Using a priority queue, we ensure that children that contain the least number of primitives are merged first. Afterwards, we have groups of primitives with roughly an equal number of primitives in each group. Then we create a set of files, according to the created number of groups. For each file, we also create a write buffer in main memory (this limits the number of groups that can be created in one pass). We go through the input data again and place each triangle in the write buffer of the region it belongs to. If a write buffer is full, we write it to disc in one linear write operation, thus avoiding costly seeks.

After this procedure, we call the construction algorithm recursively for the created files. Then we take the returned file handles that point to complete data structures and build the top of the tree in-core. To do this, we can use exactly the same algorithm as used for instantiation: Whole subtrees are treated as single primitives and inserted in the higher level tree (see Section 4.2.5 for details). Then, we write the data to disc and return the file handle for the created data structure to be included in higher level data structures. If we want to obtain one single file, we can gather all created files afterwards and copy them for concatenation if desired.

The number of seeks for this algorithm mostly depends on the size of the write buffer for partitioning the data. If we use large write buffers, we obtain only very few seeks in one pass. However, the adaptivity of the division process is reduced so that additional passes might be necessary. In addition, the distribution of the primitives in the scene is important: If the geometry is concentrated in a few very small spots, multiple passes are also necessary. However, at each pass, the resolution of the map is increased by a large constant (a grid size of 128^3 is no problem, even with simple arrays). Thus, we rapidly increase the map resolution. The number of passes depends only on the logarithm of the ratio between the size of a local cluster of geometry and the scene size to a very large basis. In practice, we can expect to handle most scenes with only one mapping pass unless they are really very large or ill formed.

The rendering algorithm uses a caching approach to access the external data structure: We can employ the same rendering algorithm as before but replace the in-core pointers by proxy objects that load the nodes from file if they are not present in main memory. LRU-scheduling is used to replace cached nodes if they have not been used for some time. To speed up this swapping algorithm, we can perform suitable blocking of the data structure. Instead of swapping in one node at a time, several nodes could be fetched in order to minimize random access. In order to group adjacent nodes, typical options are to group local subtrees (a node and children up to a certain level) or to group subsequently siblings in the octree (Figure 125). To analyze the cost implications of these options, we determine an oversampling factor, similar to Sections 4.2.3.5 and 5.2.3.2: We assume that we want to render a single image and determine the overhead of being forced to swap in more data than necessary. Grouping local subtrees increases the branching factor of the hierarchy. Effectively, multiple resolution steps are swapped in at once. Grouping siblings corresponds to enlarging the number of sample points per hierarchy node (points per box side length k). For forward mapping, this leads to a larger constant ε describing the spatial approximation accuracy. Typically, the average oversampling due to large resolution steps in the hierarchy is worse than the oversampling due to inaccurate spatial adaptation. Therefore, it is grouping siblings in the hierarchy is probably the more promising strategy.

8.2.3 Preliminary Results

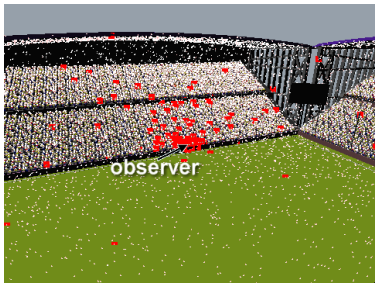
The out-of core rendering strategy depicted above has been evaluated using a prototype implementation. Using the external construction procedure, large scenes could be processed efficiently (examples up to 30GB have been tested). The data structure allowed rendering at interactive frame rates. The results show that the strategy proposed theoretically can be used in practice. However, the quantitative results are still preliminary. Aspects like blocking, stratification, and prefiltering still have to be examined more in detail. For more details, please refer to [Klein et al. 2002, Klein et al. 2004].

8.3 Sound Rendering

8.3.1 Overview

Multi-resolution point hierarchies cannot only be used for visualizing complex scenes. An interesting option is to apply the strategy to sound rendering [Funkhouser et al. 2002]: We are given a three-dimensional scene containing a large number of sound sources. The task of the rendering algorithm is to reproduce the resulting sound at a given observer position. This problem is of some relevance for virtual reality and gaming applications. Consider for example the football sta-

(a) observer in a complex scenes, containing many sound sources (football stadium)



(b) rendering strategy: white: sound sources (primary and phantom sources) red: selected sample sources

Figure 126: Visualization of the sound rendering strategy.

dium scene from Section 7.3.3, which could appear in a sports computer game. For a realistic impression, we could whish to render the sound created by the crowd of shouting and singing football fans. If we want to *auralize* (the acoustic equivalent to visualize) the soundscape, we have to combine the sound waves emitted by the numerous sound sources (football fans). Additionally, "phantom" sound sources can be computed that approximate the effect of indirect, global sound propagation. The sound rendering algorithm then has to determine the sum of all sound waves as they arrive at the observer. An approximation can be determined using a hierarchical sampling approach, similar to visual point-based rendering.

8.3.2 The Sound Rendering Algorithm

We build a hierarchy of sound sources similar to the nested sampling data structure described in Section 4.2. Each sound source is characterized by its position and volume and optionally with a directional emission characteristic. The sound sources are first sorted into the octree according to their position. Then, the inner nodes obtain one representative sound source each. They are chosen by choosing a random sound source from the children with probability proportional to their volume.

The sound rendering algorithm descends into the hierarchy searching for the most important sound sources, i.e. it subsequently fetches the loudest representative sources according to the current observer position. This is done using a priority queue of nodes weighted by the received volume (similar to the volume rendering algorithm of Section 8.1). This strategy performs an importance sampling similar to the visual rendering algorithms, choosing nodes that appear louder at the receiver with higher probability. It accounts for volume and position of the sound sources as well as for "view frustum culling", i.e. the source selection can be weighted by a directional characteristic of the receiver (e.g. higher received volume for sources in front of the observer). However, the algorithm cannot account for directional characteristics of the sound sources. Sound emitters that send sound waves with a pronounced directional characteristic (such as phantom sound sources that approximate specular reflections) will still lead to a high variance in the stochastic estimate. Therefore, we perform a two step sampling: First, we collect sound sources from the hierarchy, but fetching more than necessary (typically 10 times more sound sources). Next, a linear time selection algorithm is run on the candidate set, selecting sources that emit in the direction of the observer with higher probability. This technique works quite well for moderately directional emission characteristics.

After selecting sample sound sources, they are handed over to a mixer thread, which displays the result by mixing the corresponding sound data in real-time, typically at a sampling rate of 44.1 KHz. The mixer parameters are updated periodically by the hierarchical sampling algorithm, but at a much slower sampling rate of typically 20 Hz. The mixer receives "key frames" that describe volume and phase information and interpolates smoothly between these parameters to avoid discontinuity artifacts. The sound rendering quality depends on the number of sound sources the mixer thread is able to handle in real-time. In practice, about 2000 instances of a smaller set of prototype sound sources can be handled on a Pentium 4 2GHz (44.1 KHz, mono output, i.e. 1000 sources for stereo reproduction).

8.3.3 Results

We have implemented a prototype sound rendering algorithm and integrated it into the framework for point-based visual rendering. As test case, we have used the football stadium scene from Section 7.3.3 (Figure 126). Each football fan has been assigned one of a set of prototype sound files with slightly varying phase. Additionally, phantom sound sources were placed automatically by a "photon tracing" algorithm simulating global reverberations. Using a dual processor Xeon 2.2 GHz system, we were able to render both sound and images in real-time during interactive walkthroughs. Using 2000 sound sources, the sound rendering quality was acceptable, at least for gaming oriented applications. For more details, see [Wand and Straßer 2003c].

8.4 Caustics Rendering

8.4.1 Overview

The last example in this "extensions" chapter does not describe a multi-resolution rendering technique. However, it shows that dynamic surface sampling algorithms are useful for other rendering problems, too.

The problem that we want to solve here is rendering caustics of extended light sources in real-time. Caustics (in computer graphics terminology) are lighting effects that occur if light is transported via one or more specular patches (reflective, refractive) and received by a diffuse (or glossy diffuse) receiver surface. The specular patches potentially focus the light, creating structured patterns on the receiving surface. The traditional technique for rendering these effects is photon tracing [Arvo 86] (see also [Jensen et al. 2001, Jensen et al. 2003] for a recent survey of related rendering techniques). These techniques require a lot of ray queries and an expensive density estimation step. Often, they are not applicable in real-time.

In this section, we propose an alternative technique that can render single bounce caustics (i.e. one specular interaction along the light path) of extended light emitters efficiently. The basic idea is fairly simple and can be explained by considering an analogy, a disco ball: A disco ball is a small facetted sphere that is lit by a spot light, creating several small light spots on the walls of a room (typically a discotheque). A closer look at the spots on the wall reveals that these are just projections of the light source. The facets of the disco ball act as reflecting pinhole cameras, projecting images of the light source at the receiving surfaces.

This is a single bounce caustic effect. The same idea can also be used to render caustics from more complex objects: We discretize the surface of the reflecting (or refracting) object into sample points, using one of the sampling techniques discussed before (Section 4.2.3). Then, an

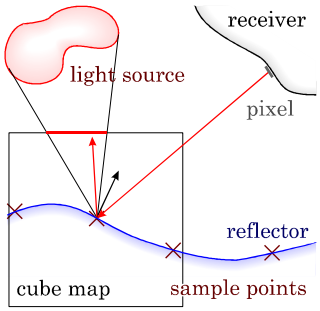
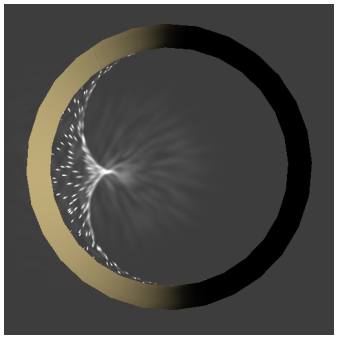
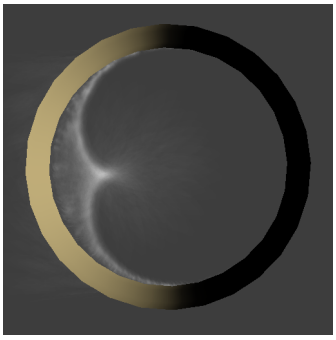


Figure 127: Implementing caustics rendering using texture mapping.

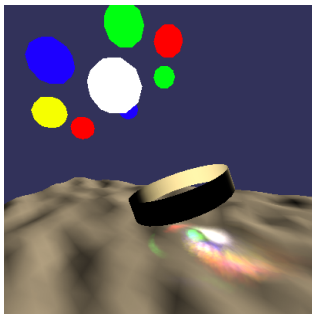


(a) no antialiasing

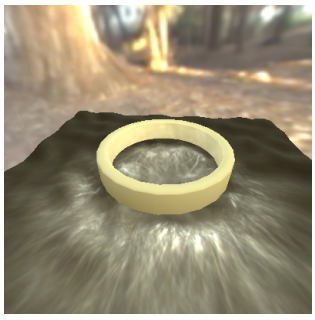


(b) filtering according to surface curvature and point spacing

Figure 128: Aliasing and Antialiasing (1000 sample points).



(a) Dynamic scene, the light source and the reflector are moved interactively (200 sample, 400×400 pixels, points, no AA, / 9.6 fps).



(b) real-time rendering (154 sample points, 400×400 pixels, no AA, 6.4fps)

(c) antialiased rendering (330 sample points, 400×400 pixels, software rendering)

Figure 129: Rendering examples (ATI Radeon 9700Pro, Pentium 4 2GHz). The high dynamic range natural light environment map in (b), (c) is taken from [Debevec 98].

image of the light sources is projected onto the receiving surfaces where each sample points act as a pinhole camera.

8.4.2 Problems and Solutions

To implement the proposed method efficiently, we assume that the light sources are far away so that the incoming light at the reflector can be represented as an environment map (the implementation uses a cube map). Then we rasterize the receiver once for each reflector sample point. For each rasterized fragment, a texture lookup in the cube map is computed based according to the projection of the virtual pinhole camera (Figure 127). The texture mapping process is implemented as a DirectX 9 pixel shader.

If we apply this algorithm to different application scenes (Figure 128a, Figure 129a,b), we obtain renderings that converge towards the correct solution. However, the visual convergence is bad for light environments with high frequency details (such as the point light source in Figure 128). Therefore, we need an antialiasing strategy. Obviously, the difference between a smooth surface, which creates continuous caustics, and a facetted surface, which creates spots like a disco ball, is the local surface curvature. Hence, we compute differential sample points instead of simple sample points [Kalaiah and Varshney 2001]. Then, we compute reflected ray frustums at the sample points instead of sample rays, similar to computing reflected rays for point-based raytrac-

ing described in Section 6.2.4. As a result, we obtain not only a lookup direction for the cube map but additionally two derivative vectors that describe a first-order approximation of the footprint of all lookup vectors for the area around the sample point. These three vectors are then used to perform an anisotropic texture lookup offered by programmable graphics hardware (according to the DirectX 9 pixel shader 2.x model). The hardware estimates the integral over the given region using a footprint assembly technique [Schilling et al. 96].

8.4.3 Results

The implementation of the algorithm has been tested on a Pentium 4 PC equipped with an ATI Radeon 9700Pro graphics board. Using this configuration, non-antialiased renderings were possible at interactive frame rates (see Figure 129). For antialiased rendering, a pixel shader 2.x compliant hardware is necessary. At the time of writing, only nVidia boards of the GeForceFX architecture offered these capabilities. However, we were not able yet to use programmable anisotropic texture lookup instructions for cube maps, probably due to restrictions of the current drivers. Therefore, the results show in Figure 128b and Figure 129c are still non-interactive software renderings. For more details on the caustics rendering technique, see [Wand and Straßer 2003b].

Chapter 9
Conclusions and Future Work

This last chapter of the thesis summarizes the main results and concludes with some ideas for future work.

9.1 Conclusions

9.1.1 Summary

In this thesis, different point-based multi-resolution rendering strategies have been presented. Two alternative data structures have been proposed: Dynamic and static sampling. Different variants of the data structure are possible: Dynamic sampling can be modified to allow for dynamic modification, area classes can be used to identify large triangles. Static sampling can be implemented as nested or full sampling data structure. Additionally, different sampling and stratification techniques can be chosen. A further important choice is the representation of sample points: Pure point sample representations yield infinitesimally small sample points from the surfaces. Static (full) sampling additionally allows for prefiltering, storing average, band limited color and differential surface properties [Pfister et al. 2000, Kalaiah and Varshney 2001].

Two main rendering algorithms have been used to generate images from these data structures: forward mapping and backward mapping. Forward mapping techniques aim at real-time applications. Reconstruction techniques varying from simple opaque splatting to band-limited filtered reconstructions can be employed to generated images. The point-based backward mapping techniques described in this thesis are not interactive. Instead they offer rendering of antialiased images and classic distributed raytracing effects at moderate costs.

In addition to these two rendering techniques, the concepts of point-based multi-resolution rendering can also be generalized to visualize large volume data sets interactively, visualize data sets that do not fit into main memory, and might be even useful for sound rendering of complex soundscapes. We have also employed surface sampling techniques to speed up rendering of caustics for interactive global illumination.

9.1.2 Main Results

The sampling data structures can be constructed efficiently with little memory overhead: The dynamic sampling data structure can be constructed in $O(n \log n)$ time for n triangles. It uses $O(n)$ memory. In practice, the construction is very efficient: Typically, preprocessing takes only a few seconds for models consisting of a hundred thousand triangles and instances in the underlying scene graph.

Nested static sampling can be constructed within the same time and memory bounds. For static sampling with full sampling, we obtain $O(nh)$ construction time (with h denoting the height of the octree). The memory demands are theoretically unbound, but in practice, no superlinear behavior is expected in most relevant cases. The preprocessing times are significantly larger than for dynamic sampling, especially if prefiltering is employed.

Rendering using forward mapping consists of three steps: Approximation of the projection factor (that scales area in a perspective projection), sampling and image reconstruction. To approximate the projection factor, the spatial hierarchy constructed during preprocessing is traversed. The process takes $O(h + \log \tau)$ time, τ being the maximum relative depth range, i.e. the ratio between near and far clipping plane. The selection process also performs an approximate view frustum culling, enlarging the cross-sectional area of the view frustum at most by a constant factor. If we assume that the orientation of the triangles of the scene is random, with normals distributed uniformly on the unit sphere and geometry distributed evenly within the (extended) view frustum, we are able to approximate the integral projection factor up to a constant value. Orientation classes can be used to improve the approximation of the orientations of the triangles. However, in adverse cases, they cannot guarantee a constant overestimation. In practice, only moderate benefits are observed when orientation classes are employed (20% performance improvement).

The sampling step determines a number of sample points according to the projected area. The number of sample points is determined by the product of the actual area of the geometry and the estimated (approximated) projection factor. Thus, the rendering costs increase with the overestimation of this factor. Dynamic sampling leads to costs of $O(\log n)$ for each selected sample point, static sampling needs $O(1)$ time. Sample caching can be used in order to reduce the additional costs of dynamic sampling. The concrete number of sample points that have to be processed depends on the structure of the point sets: Stratified sampling patterns yield a number of sample points linear in the estimated projected area. Random sampling patterns are larger by a logarithmic factor, i.e. $O(\bar{a} \log \bar{a})$ with $\bar{a}$ denoting the estimated projected area.

Different sampling and stratification techniques lead to different rendering costs: random sampling needs the largest sample sizes. Grid stratification and quantized grid stratification can reduce these costs, removing the logarithmic factor and improving the constants. Neighborhood-based point removal is the most efficient of the proposed techniques, both by theoretical analysis and empirical evidence.

Using static stratified sampling, we obtain a rendering time linear in the estimated projected area and logarithmic in the relative depth range, i.e. demanding overall $O(\bar{a} + h + \log \tau)$ time. For average scenes, this means especially a rendering time linear in the actual projected area. For dynamic sampling, the costs increase by two logarithmic factors, yielding $O(\bar{a} \log \bar{a} \log n + h + \log \tau)$ rendering time.

Different image reconstruction techniques offer a trade-off between quality and rendering time. Although all approaches are linear time algorithms, the rendering time can vary substan-

tially in practice. Per-pixel reconstruction and opaque splatting are the most efficient techniques. Using an implementation accelerated by contemporary graphics hardware, walkthroughs of highly complex scenes are possible at real-time frame rates. A Gaussian filtering technique can be employed to render images with little noise and aliasing artifacts but at non-interactive framerates. The best trade-off is probably obtained using splatting techniques: Alpha-blended splats with an area based transparency heuristic yield good results for displaying vegetation scenes, but still have to deal with aliasing issues for regularly structured image content. Surface splatting as proposed by [Zwicker et al. 2001a] (which has been examined here only in an extended variant based on raytracing) yields better antialiasing properties. A subpixel-mask heuristic can be employed to increase the resolution of occlusion in problematic regions such as complex boarders at additional run-time costs.

The static sampling technique can be applied to animated scenes (keyframe animations), too. Assuming some coherency in the input animations, the good rendering times can be retained. In practice, large cloud animations such as a football stadium or a flock of animals with several thousand individuals can be displayed at real-time framerates.

The backward mapping (raytracing) techniques, also relying on a variant of the static sampling data structure, yield anti-aliased images and can approximate classic distributed raytracing effects such as soft shadows, blurry reflection and depth-of-field. The image quality is quite close to reference solutions obtained from real distributed raytracing but still distinguishable. The main problem is an overestimation of silhouette opacities in complex situations. Even with subpixel masks, this effect cannot be avoided completely. This is a general property of all examined point-based multi-resolution rendering techniques: As we resample the scene to the pixels of the screen prior to rendering, complex subpixel occlusion effects (which cannot be modeled by tangent discs or an area-based opacity estimate) cannot be resolved faithfully. This is an inherent restriction of the approach itself. However, for many scenes in practice (such as most of those showed in this thesis), the artifacts are of minor relevance.

The performance of the point-based multi-resolution raytracing algorithm is much better than that of classic cone tracing. In contrast to the classic approach, highly complex scenes can be handled, due to the output-sensitive running time. In comparison with distributed raytracing, the novel approach still provides a performance benefit for scenes showing large areas of high variance in the image. If no complex silhouettes are contained in the image, the performance advantage is substantial: The algorithm yields virtually noise free solutions. At the same rendering time, distribution raytracing still leads to considerable noise artifacts. For scenes with complex silhouettes, subpixel masks have to be employed. Using the current implementation, this reduces the performance. Distributed raytracing is then able to render images with reduced, but still visible, noise artifacts within the same time. The drawback of the multi-resolution approach is bias: The estimated value of the result is not the correct image but only an approximation.

Some extensions to the basic multi-resolution rendering framework have been proposed: A multi-resolution volume rendering approach is able to render approximate images of very large data sets at interactive framerates on a conventional PC. The formal analysis of the multi-resolution approach for point-based rendering of surface models is applicable to volume rendering, too. The theoretical costs predictions have been crucial for the design of an efficient volume rendering technique. Further results also show that the proposed point-based rendering techniques can also be used in out-of-core settings, for data sets not fitting into main memory. The same hierarchical data structure that is used for visual rendering can also be used for sound rendering. Approximate renderings of scenes with a large number of sound emitters are possible in real-time. The rendering quality is (subjectively) sufficient, at least for VR and entertainment

applications. In addition, a real-time rendering technique for anti-aliased, single-bounce specular to diffuse light transport (caustics) has been discussed that relies on surface sampling and programmable texture mapping.

9.1.3 Discussion

It has been shown that point-based multi-resolution techniques can increase the efficiency for rendering complex scenes. The forward mapping techniques are able to display highly complex scenes in real-time. In contrast to many former mesh simplification techniques, the point-based techniques are applicable independently of the topological structure of the scene. In contrast to image-based techniques, the precomputation demands are small and no problems with parallax errors are observed. The rendering time depends (mostly) on the projected area, including occluded area. Therefore, a combination with an occlusion culling technique might be necessary in some cases to sustain real-time performance.

Several algorithms and data structures have been proposed. Which one should be chosen for a given application? In most cases, static sampling with full sampling sample sets and neighborhood-based stratification is the most efficient choice. Scene and lighting model permitting, prefiltering should be employed as it reduces aliasing and noise problems drastically. However, in some situations, prefiltering can lead to artifacts (as for the forest scenes). In this case, one can either design a suitable heuristic to avoid the artifacts, or rely on simple point samples, which are often less problematic. In the latter case, a suitable supersampling technique (Gaussian reconstruction, surface splatting, averaging) should be employed to remove noise artifacts from the image. Static nested sampling is more of theoretical interest, allowing for optimal preprocessing costs. In practice, it provides only minor advantages.

Rendering with dynamic sampling is slower than rendering with static sampling for several reasons: First, random dynamic sampling itself is more expensive. Second, prefiltering cannot be employed so that expensive oversampling is necessary for a high image quality. However, the technique also provides some advantages. A major advantage is moderate preprocessing costs: The preprocessing time is much shorter than for stratified, prefiltered static point sample hierarchies. Additionally, interactive dynamic modifications are possible. Dynamic updates could probably also be devised for the static case but presumably at higher costs, especially if prefiltering is demanded. Additionally, the concept of dynamic, hierarchical random sampling is quite general. Probably, the technique could be generalized more easily to more general modeling techniques such as procedural models.

The backward mapping rendering algorithm, point-based multi-resolution raytracing, should be compared with distributed raytracing (cone tracing, the second alternative, is of minor importance today): The major advantage of the point-based technique is the avoidance of noise artifacts: As filtering is already done during preprocessing, low-noise images can be obtained at smaller costs than in the distributed raytracing case. For some applications, such as rendering of animations, this might be an important aspect. The drawback is the approximate representation. Images rendered with distributed raytracing converge towards an accurate solution while the point-based technique may produce artifacts (overestimated silhouette opacities, prefiltering artifacts etc.). Please note that the performance relation between the two techniques depend strongly on optimizations of the implementation. Therefore, our current conclusions concerning performance can only be preliminary as optimization of a point-based multi-resolution raytracing technique has not yet been studied extensively.

For rendering complex keyframe animations, especially crowd animations, the proposed point-based multi-resolution technique is an interesting option: Mesh simplification techniques for animated scenes are rather involved and image-based simplification techniques suffer particularly from discretization problems in the animated case. For this reason, only few alternative techniques for rendering complex animated scenes have been published yet. The point-based technique is easy to implement and quite effective, allowing for real-time applications. Thus, it may be an interesting result of this thesis for use in practical applications.

9.1.4 Conclusions

In conclusion, point-based multi-resolution rendering is not a perfect rendering technique, solving all rendering problems: The most important problem concerning image quality is probably the heuristic nature of the representation of subpixel details. This appears to be an inherent restriction of the approach and presumably of any efficient hierarchical approximation technique [Chamberlain et al. 96]. In terms of performance, we have at least three problems: First, point-based multi-resolution rendering is only a simplification technique. This means, that we need an additional occlusion culling strategy for efficient forward mapping rendering in densely occluded environments. Second, we cannot guarantee linear $O(a)$ (or $O^{\sim}(a)$ for dynamic/random sampling) running time in terms of the exact projected area a. We still need some additional assumptions such as a uniform distribution of normals and restricted relative depth range. However, this is mostly a theoretical issue and only a minor problem in practice. The third problem is the large base costs: In contrast to mesh simplification, point-based simplification starts to accelerate rendering only at the point where the scene contains subpixel details. Therefore, a high geometry processing rate is necessary for efficient rendering. For low complexity scenes, the technique does not provide significant advantages (neither for forward nor for backward mapping). However, current graphics hardware is fast enough to permit real-time rendering (forward mapping) at least at moderate image resolutions.

Despite these inherent problems, point-based multi-resolution rendering is an important complement to current rendering techniques for large scenes, substantially extending the range of scenes that can be rendered efficiently: Complex scenes such as landscapes or animated crowds consisting of billions of primitives and more can now be rendered in real-time. Even for simple, real-time reconstruction techniques, the image quality is not worse than that of conventional z-buffer rendering, which would have taken years to finish a single image, in extreme cases. The underlying techniques are also applicable to other approximation problems such as antialiasing in raytracing, volume rendering, and sound rendering. Whether the advantages or the drawbacks of the technique are dominant depends on the target application. This thesis should facilitate this decision by the presented analytical and empirical facts.

9.2 Future Work

Point-based multi-resolution techniques offer a new approach to many problems. Only recently, the capabilities of widely available hardware systems have become powerful enough to allow for an elementary, lightweight data representation such as sample point clouds. Therefore, it is likely that similar representations and techniques will receive growing attention in the near future. In the following, we will discuss some ideas for future work in the area of point-based multi-resolution techniques, derived from the experience with the techniques described in this thesis.

Modeling: The most important problem left to be solved in the area of rendering complex scenes is probably a modeling problem. As shown in this thesis (and by many other authors, too), it is nowadays possible to render many highly complex models in real-time. The main problem is now a representation problem. We can render approximations of complex landscape with mountains, trees, grass if we were are able to provide a suitable model of such a scene. Currently, this is done using simple instantiation of a few base models. Using such a technique, some remarkable results are possible [Deussen et al. 98], but it might not be sufficient in general: The visual complexity of the scene is fixed in advance and the redundant structure of the scene may be visible, even if more sophisticated instantiation techniques were used. Alternatively, out-of-core storage can be employed. However, this approach only shifts the limits from the limits of main memory to the storage limits of an external storage device.

Therefore, an interesting direction for future research could be the integration of modeling with rendering. Hierarchical modeling techniques such as fractal subdivision could probably be combined with point-based multi-resolution rendering in order to generate details on-the-fly (e.g. based on hierarchical subdivision, similar to the REYES architecture [Cook et al. 87]). Another option to obtain more realistic models is the use of measured real-world data. Point-based representations have been shown to be useful for 3D-photography, i.e. as representation of the appearance of real-world objects [Matusik et al. 2002, Poulin et al. 2003]. Probably, both techniques could even be combined. A generalization of current texture synthesis techniques [Wei and Levoy 2000] to three-dimensional point clouds in addition to a suitable acquisition method for three-dimensional real-world objects could probably be used to synthesize photo-realistic objects on-the-fly, during rendering.

Besides this, the usage of points as modeling primitive (without focus to complex scenes) has gained some attention recently [Zwicker et al. 2002a, Pauly et al. 2003a].

Rendering efficiency and image quality: The efficiency of the techniques proposed in this thesis could be improved further in future work. For forward mapping, a high geometry processing rate is essential. This can currently only be achieved by utilizing programmable graphics hardware for geometry processing. This leads to several optimization problems, such as compact storage and transfer of geometry data and the problem of hiding hardware latencies. In addition, the mixture of triangle-based and point-based rendering could be improved. For locally flat regions, triangles could be used more efficiently for approximating the surface. Some first steps in investigation such dual simplification strategies have already been made [Cohen et al. 2001, Guthe et al. 2003].

To improve the image quality of the current techniques, two main problems have to be considered: The first problem is the representation problem for sample points. It is still not clear which kind of attribute sets (assuming a prefiltering approach) are best to use for different applications. Although an optimal solution with fixed costs and guaranteed quality may be impossible, there is probably still some room for improvements. The second problem is the reconstruction problem. Current techniques work well for smooth surfaces. However, better techniques are necessary to deal with general cases. A combination of volume rendering and splatting techniques (such as reconstructing subpixel ray volumes using volume splats, as a generalization of subpixel masks) seems to be promising.

Global illumination: The raytracing techniques described in this thesis are only a first step into the direction of using point-based multi-resolution techniques for global illumination problems. The current technique is still rather expensive, but there are several options to speed up the rendering process: First, the technique itself could be optimized by approximating expensive operations such as eigenvalue computations by more lightweight techniques based on inter-

polations or precomputed tables. Second, the algorithm itself could be altered. One idea could be to avoid the expensive cone queries. Instead, infinitesimal rays could be shot and interpolation due to local information at the sample points (color gradients, normals, curvature) could be used for generating high-quality images. We could also combine distributed raytracing and prefiltering: Shooting multiple infinitesimal rays at prefiltered, pixel sized points we could avoid noise and aliasing in the color channel and only use stochastic integration to estimate the opacity. Another option could be to perform only a rough raytracing with a few rays and compute the image by projecting larger point clouds that intersect with the rays. For secondary rays, the mapping between adjacent point cloud intersections could be interpolated smoothly. Adaptive sampling could control the image quality.

Aside from point-based raytracing, it would also be interesting to apply point-based multiresolution techniques to speed up other global illumination calculations such as radiosity techniques.

Software architecture: A predominant problem in all areas of computer science is the design of a suitable software architecture for employing new techniques in practice. At least two important aspects need further attention: First, we have to consider the modeling problem. Even for the current instantiation-based technique, more flexibility and orthogonality would be desirable. More involved procedural modeling techniques would also pose architectural problems: What is a good abstraction for general hierarchical modeling? Which framework should be provided to facilitate the development of application specific modeling modules? A second problem is modeling of surface appearance. In this thesis, we have only used simple shading models such as Phong shading or even constant color shading. This might be sufficient to demonstrate the effects of geometric simplification. However, for practical use, more advanced shading and lighting models are mandatory. Multi-texturing and programmable shader scripts are state-of-the art both in real-time and offline rendering. The integration of a general shading architecture with point-based rendering is a complex problem. Aspects such as hardware requirements (batch processing, minimizing state changes) have to be considered as well as approaches for generalizing point attributes for different, programmable material models to allow for general prefiltering.

Appendix A: Tables / Measurements

Dynamic Sampling

epsilon	0.0025	0.005	0.01	0.02	0.03	0.04	0.05	0.1
near view	25.0	9.99	5.52	4.10	3.72	3.69	3.72	3.94
medium distance	22.4	8.17	4.35	2.97	2.54	2.49	2.46	2.59
far view	7.76	3.46	2.10	0.90	0.8	0.77	0.76	0.76

epsilon	0.19	0.41	0.68	1.00	1.83	2.36	3.00	7.00
near view	4.39	5.47	6.74	7.99	11.4	13.8	16.3	16.3
medium distance	2.84	3.36	3.94	4.50	5.92	6.56	7.15	7.23
far view	0.80	0.99	1.09	1.18	1.39	1.49	1.67	1.67

Table 12: Rendering time [sec] in dependence of the approximation accuracy ε (Figure 80).

epsilon	0.0025	0.005	0.01	0.02	0.03	0.04	0.05	0.10
rendering time [sec]	25.01	9.99	5.52	4.10	3.72	3.69	3.72	3.94
points	1,008,000	1,428,700	1,845,000	2,270,900	2,496,600	2,633,380	2,703,800	2,842,150
boxes	3,940,003	1,633,461	679,863	269,946	141,700	90,787	64,676	23,650
triangles	13,328,332	3,351,266	849,685	224,845	115,067	75,836	60,457	48,516

epsilon	0.19	0.41	0.68	1.00	1.83	2.36	3.00	7.00
rendering time [sec]	4.39	5.47	6.74	7.99	11.36	13.77	16.33	16.34
points	3,032,630	3,495,300	4,037,600	4,675,240	6,206,500	7,498,900	8,780,000	8,780,000
boxes	10,447	3,977	2,062	1,285	690	593	461	461
triangles	52,026	59,036	61,511	62,238	71,725	75,695	47,492	47,492

Table 13: Number of processed primitives for varying approximation accuracies ε (near view, Figure 81).

epsilon	0.0025	0.005	0.01	0.02	0.03	0.04	0.05	0.10
rendering time [sec]	7.76	3.46	2.10	0.90	0.80	0.77	0.76	0.76
points	393,026	393,972	394,560	395,856	397,251	398,957	400,673	408,770
boxes	2,071,217	742,986	363,331	78,063	38,524	26,001	18,985	8,027
triangles	0	0	0	0	0	0	0	0

epsilon	0.19	0.41	0.68	1.00	1.83	2.36	3.00	7.00
rendering time [sec]	0.99	1.09	1.18	1.39	1.49	1.67	1.67	0.99
points	453,954	491,660	531,094	623,079	663,561	745,460	745,461	453,954
boxes	628	213	121	54	40	30	30	628
triangles	0	0	0	0	0	0	0	0

Table 14: Number of processed primitives for varying approximation accuracies ε (far view, Figure 82).

epsilon / rel. sampl. density	0.0025	0.005	0.01	0.02	0.03	0.04	0.05
2² pts / pixel	341.89	113.07	41.07	15.00	7.82	5.11	3.79
1 pts / pixel	153.72	50.87	18.40	6.84	3.78	2.56	2.01
(1/2)² pts / pixel	55.71	18.47	6.79	2.83	1.76	1.37	1.19
(1/4)² pts / pixel	19.10	6.31	2.65	1.43	1.14	1.04	1.00
(1/8)² pts / pixel	6.09	2.35	1.33	1.04	1.00	1.01	1.03
(1/16)² pts / pixel	2.25	1.22	1.00	1.02	1.07	1.12	1.17
boxes	3,940,565	1,634,023	680,312	267,385	138,525	88,312	62,676
points (max dens.)	191,953	999,841	1,668,320	2,207,950	2,472,930	2,661,150	2,814,790

epsilon / rel. sampl. density	0.19	0.41	0.68	1.00	1.83	2.36	3.00
2² pts / pixel	1.68	1.17	1.00	1.03	1.10	1.29	1.57
1 pts / pixel	1.18	1.00	1.02	1.16	1.33	1.68	2.19
(1/2)² pts / pixel	1.00	1.02	1.16	1.37	1.62	2.17	2.94
(1/4)² pts / pixel	1.06	1.18	1.42	1.70	2.05	2.84	3.83
(1/8)² pts / pixel	1.20	1.39	1.72	2.10	2.55	3.54	4.82
(1/16)² pts / pixel	1.43	1.71	2.15	2.64	3.22	4.50	6.12
boxes	21,913	8,991	3,189	1,607	987	514	337
points (max dens.)	3,588,190	4,120,030	4,766,190	5,480,680	6,417,200	8,234,580	10,471,900

Table 15: Trade-off between the rendering cost components for varying ε and different sampling densities (Figure 83). Rendering time in [sec].

subdivision depth	0	1	2	3	4	5	6
#orientation classes	1	2	4	8	16	32	64
near view	1,00	0,81	0,95	0,91	0,96	1,03	1,18
medium distance	1,00	0,77	0,89	0,84	0,87	1,04	1,14
far view	1,00	0,93	1,03	1,10	1,12	1,22	1,30

Table 16: Relative rendering time [factor in respect to rendering time with no orientation classes] in dependence of the number of orientation classes (Figure 84).

area factor	1.09	1.19	1.41	1.7	1.8	1.9	1.95	2
rendering time [sec]	4.101	3.899	3.516	3.351	3.399	3.381	3.300	3.722
opt. ε value	0.18	0.125	0.08	0.08	0.08	0.08	0.08	0.08

area factor	2.25	2.5	3	4	8	16	32	64
rendering time [sec]	3.413	3.550	3.508	3.503	3.364	3.411	3.516	3.719
opt. ε value	0.08	0.08	0.07	0.06	0.04	0.04	0.04	0.04

Table 17: Rendering time for differently sized area classes (Figure 85).

max. sampled area	0.00195	0.0039	0.0078	0.0156	0.0313	0.0625	0.125	0.25
ε = 1.04, density = 1	15.663	8.946	5.640	4.231	3.656	3.480	3.488	3.588
ε = 1.5, density = 1	17.715	10.540	7.189	5.536	5.182	5.012	5.259	5.483
ε = 1.5, dens. = 1/16	15.778	8.271	4.366	2.213	1.317	0.808	0.582	0.492

max. sampled area	0,5	1	2	4	8	16	32	64
ε = 1.04, density = 1	3.733	3.837	3.856	3.883	3.904	3.889	3.897	3.889
ε = 1.5, density = 1	5.679	6.033	6.216	6.689	7.168	7.436	7.439	7.456
ε = 1.5, dens. = 1/16	0.465	0.458	0.463	0.506	0.507	0.523	0.520	0.523

Table 18: Transition between triangle rendering and point-based rendering (Figure 86).

inst. level	subdivision	1	2	3	4	5	6	7	8
0	#triangles	2	24	130	600	2,562	10,584	43,010	173,400
	time [sec]	0.712	0.768	0.809	0.838	0.877	1.042	1.220	1.505
1	#triangles	200	2,400	13,000	60,000	256,200	1,058,400	4,301,000	1.73E+7
	time [sec]	1.088	1.136	1.162	1.206	1.234	1.409	1.613	1.912
2	#triangles	20,000	240,000	1,300,000	6,000,000	2.56E+7	1.06E+08	4.30E+08	1.73E+09
	time [sec]	1.483	1.539	1.542	1.610	1.631	1.824	2.004	2.300
3	#triangles	2,000,000	2.40E+7	1.30E+08	6.00E+08	2.56E+09	1.06E+10	4.30E+10	1.73E+11
	time [sec]	1.866	1.917	1.920	1.962	2.001	2.184	2.410	2.738
4	#triangles	2.00E+08	2.40E+09	1.30E+10	6.00E+10	2.56E+11	1.05E+12	4.30E+12	1.73E+13
	time [sec]	2.218	2.281	2.341	2.380	2.445	2.636	2.879	3.268
5	#triangles	2.00E+10	2.40E+11	1.30E+12	6.00E+12	2.56E+13	1.06E+14	4.30E+14	1.73E+15
	time [sec]	2.675	2.726	2.724	2.806	2.836	3.030	3.295	3.528

Table 19: Rendering time of dynamic sampling for varying scene complexities (Figure 87).

update compl. [triangles]	500	1,000	2,000	5,000	10,000	20,000	50,000	100,000
time [sec]	0.050	0.080	0.135	0.300	0.581	1.170	2.947	5.999
time/triangle [µsec]	100	80	68	60	58	59	59	60

scene compl. [triangles]	33,950	135,800	305,550	543,200	848,750
time [sec]	1.035	1.079	1.127	1.123	1.170
time/triangle [µsec]	52	54	56	56	59

Table 20: Update times for dynamic modifications of the dynamic sampling data structure (Figure 88). The first table shows update times for a varying number of changed triangles in a 848,750 triangle scene. The second table shows update times for a varying scene complexity, updating 20,000 triangles in each case.

Static Sampling

k = pts / box side length	8	12	16	32	48	64	96	128
epsilon(*k*)	0.048	0.073	0.098	0.149	0.200	0.308	0.419	0.656
near view	2.49	1.19	0.74	0.43	0.31	0.25	0.23	0.25
medium distance	1.94	0.72	0.35	0.18	0.13	0.12	0.11	0.11
far view	0.071	0.05	0.041	0.03	0.03	0.02	0.03	0.02

Table 21: Rendering time [sec] in dependence of the approximation accuracy, given implicitly by the number of points per side of an octree box (Figure 89).

k = pts / box side length	8	12	16	32	48	64	96	128
epsilon(*k*)	0.048	0.073	0.098	0.149	0.200	0.308	0.419	0.656
rendering time [sec]	2.49	1.19	0.74	0.43	0.31	0.25	0.23	0.25
Points	1,134,270	1,292,855	1,443,502	1,664,067	1,797,372	2,006,868	2,111,018	2,249,912
Boxes	9,850,741	4,441,097	2,509,804	1,174,450	675,757	375,339	285,649	280,042
Triangles	90,077	46,783	30,326	16,602	10,515	5,774	3,650	2,036

Table 22: Number of processed primitives for varying approximation accuracies (near view, Figure 90).

k = pts / box side length	8	12	16	32	48	64	96	128
epsilon(*k*)	0.048	0.073	0.098	0.149	0.200	0.308	0.419	0.656
1 pts / pixel	2.606	1.247	0.771	0.458	0.310	0.268	0.251	0.264
$(1/2)^2$ pts / pixel	0.698	0.363	0.231	0.151	0.105	0.098	0.089	0.103
$(1/4)^2$ pts / pixel	0.209	0.118	0.086	0.061	0.050	0.048	0.051	0.052
$(1/8)^2$ pts / pixel	0.079	0.053	0.045	0.031	0.023	0.019	0.021	0.026
$(1/16)^2$ pts / pixel	0.035	0.025	0.015	0.009	0.007	0.007	0.010	0.015

Table 23: Trade-off between the rendering cost components for varying approximation accuracies and different sampling densities (Figure 91). The sample spacing has been chosen proportionally to the inverse splat diameter. Rendering time in [sec].

k = pts / box side length	8	12	16	32	48	64	96	128
epsilon(*k*)	0.048	0.073	0.098	0.149	0.200	0.308	0.419	0.656
OpenGL std. v.arr. [sec]	2,070	0,775	0,433	0,262	0,216	0,185	0,175	0,165
DirectX man. v.b. [sec]	0,876	0,328	0,183	0,111	0,091	0,078	0,074	0,070

Table 24: Average rendering time (camera path near to far) for different rendering APIs (Figure 92). The OpenGL implementation uses OpenGL 1.1 standard vertex buffers (client memory) while the DirectX version uses managed vertex buffers (uploaded automatically to local video memory). This leads to a higher primitive throughput. However, the fixed costs per rendering call are higher for the DirectX-based technique.

k = pts / b.s.l.	8	12	16	24	32	48	64	96	128
time [sec]	3.4	6.6	12.0	31.6	66.2	171.5	296.8	466.7	743.6
memory [MB]	28	29	30	32	34	40	48	67	90
boxes	706	706	706	706	706	686	668	600	550
points	7,433	18,750	33,048	70,086	118,049	235,283	379,044	730,062	1,165,057

Table 25: Preprocessing costs for varying approximation accuracies (static sampling, Figure 94).

tri / leaf	64	128	256	512	1024	2048	4096	8192
rendering time	0.223	0.217	0.216	0.216	0.232	0.267	0.372	0.758
data struct. size (points)	489,560	403,908	337.208	276,382	230,450	197,663	189,113	174,228
data struct. size (boxes)	2,690	1,965	1.414	978	686	504	413	337
points / inner node	1,256	985	962	975	816	721	608	975

Table 26: Rendering time and data structure size for a varying number of triangles per leaf node (for *k* = 48 pts / box side length, Figure 96). The number of points per inner node is an average for all inner nodes of the octree of the lowest instantiation level (i.e. the octree containing the triangles, no instances).

time intervals	1	2	3	4	5	7	10	20
shrinking sphere: av. #points	10,064	6,588	5,820	5,316	4,968	4,612	4,315	4,060
reduction	0%	-53%	-73%	-89%	-103%	-118%	-133%	-148%
walking man: av. #points	5,345	5,224	5,259	5,157	5,251	5,181	5,231	5,139
reduction	0,0%	-2,3%	-1,6%	-3,6%	-1,8%	-3,2%	-2,2%	-4,0%

Table 27: The influence of time discretization for preprocessing animated geometry (Figure 97).

k = pts / b.s.l.	1	2	3	4	5	6	7	8	10	12	14	16
1 tri/leaf	21.4	16.3	18.1	20.6	21.1	19.7	22.5	21.8	24.5	24.0	29.0	29.4
2 tri/leaf	21.4	16.6	18.4	19.9	21.3	20.1	23.3	21.1	24.7	24.6	29.5	28.9
4 tri/leaf	21.5	16.4	18.2	19.8	19.9	19.8	23.6	21.9	24.5	24.4	29.0	29.0
8 tri/leaf	24.3	16.4	18.5	19.3	21.5	20.4	23.7	22.3	24.8	24.4	29.7	29.4

Table 28: Rendering time for point-based multi-resolution raytracing for a varying number of points per octree node and different numbers of triangles in leaf nodes (near view, Figure 99).

k = pts / b.s.l.	1	2	3	4	5	6
time	21.5	17.0	18.2	19.8	19.9	19.8
Boxes	4,977,869	2,612,269	2,076,975	1,732,979	1,516,659	1,376,813
triangles	3,226,238	1,341,287	784,107	316,475	274,027	164,419
Points	1,672,718	2,602,813	4,319,492	6,516,030	8,232,991	9,959,157

k = pts / b.s.l.	7	8	10	12	14	16
Time	23.6	21.9	24.5	24.4	29.0	29.0
boxes	1,284,482	1,189,630	877,366	796,987	739,874	678,828
triangles	17,262	11,404	9,035	243,252	6,572	1,638
points	14,370,714	15,780,875	22,699,226	29,012,277	43,001,024	51,612,532

Table 29: The number of points, triangles and boxes that have been tested for intersection by the point-based multi-resolution raytracing algorithm, in dependence of the number of points per octree box side length (Figure 100).

References

[Abrash 97] **Abrash, M.:** *Michael Abrash's Graphics Programming Black Book, Special Edition*, Coriolis, 1997.

[Adams and Dutré 2003] **Adams, B., Dutré, P.:** Interactive Boolean Operations on Surfel-Bounded Solids. In: *SIGGRAPH 2003 Conference Proceedings*, 2003.

[Adamson and Alexa 2003] **Adamson, A., Alexa, M.:** Ray Tracing Point Set Surfaces. In: *Shape Modeling International 2003*.

[Adelson and Bergen 91] **Adelson, E. H., and J. R. Bergen:** The Plenoptic Function and the Elements of Early Vision. In: Landy, M., Movshon, J.A. (Editors) *Computational Models of Visual Processing*, MIT Press, 1991.

[Agrawala et al. 2000] **Agrawala, M., Ramamoorthi, R., Heirich, A., Moll, L.:** Efficient Image-Based Methods for Rendering Soft Shadows. In: *SIGGRAPH 2000 Conference Proceedings*, 2000.

[Akeley 93] **Akeley, K:** RealityEngine graphics. In: *Siggraph 93 Conference Procedings*, 109-116, 1993.

[Alexa 2002] **Alexa, M.:** Recent Advances in Mesh Morphing. In: *Computer Graphics Forum*, 21(2), 2002.

[Alexa et al. 2001] **Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, C.T.:** Point Set Surfaces. In: *Visualization 2001 Conference Proceedings*, 2001.

[Amanatides 84] **Amanatides, J.:** Ray Tracing with Cones. In: *SIGGRAPH 84 Proceedings*, 129-135, 1984.

[Amanatides 96] **Amanatides, J.:** Object-Space Variance Estimators. In: *Proceedings of the Seventh Western Computer Graphics Symposium*, 85-87, 1996.

[Amor et al. 2003] **Amor, M., Boo, M., del Rio, A., Wand, M., Straßer, W.:** A New Algorithm for High-Speed Projection in Point Rendering Applications. In: *DSD-Euromicro Conference*, IEEE Computer Society, 2003.

[Apodaca and Gritz 99] **Apodaca, A., Gritz, L.:** *Advanced RenderMan, Creating CGI for Motion Pictures*. Morgan Kaufmann, 1999.

[Appel 68] **Appel, A.** Some Techniques for Shading Mashine Renderings of Solids. In: *Proceedings of the Spring Joint Computer Conference,* 37-45, 1968.

[Arnold and Gosling 96] **Arnold, K., Gosling, J.:** *The Java Programming Language*. Addison-Wesley, 1996.

[Arvo 86] **Arvo, J.:** Backward Ray Tracing. In: *Developments in Ray Tracing, SIGGRAPH 86 Course Notes*, 1986.

[Arvo and Kirk 91] **Arvo, J., Kirk, D.:** A Survey of Ray Tracing Acceleration Techniques. In: Glassner, A. (editor): *An Introduction to Ray Tracing,* Academic Press, 4th printing, pp. 201-262, 1991.

[ATI 2004] **ATI Developers Relations:** `http://www.ati.com/developer`, 2004.

[Aubel et al. 2000] **Aubel, A., Boulic, R., Thalmann, D.:** Real-time Display of Virtual Humans: Levels of Detail and Impostors. In: *IEEE Transactions on Circuits and Systems for Video Technology*, 2000.

[Bala et al. 2003] **Bala, K., Walter, B., Greenberg, D.P.:** Combining Edges and Points for Interactive High-Quality Rendering. In: *SIGGRAPH 2003 Conference Proceedings*, 2003.

[Baraff et al. 2003] **Baraff, D., Witkin, A., Anderson, J., Kass, M.:** Physically Based Modeling. In: *SIGGRAPH 2003 Course Notes*, 2003.

[Bartz et al. 99] **Bartz, D., Meißner, M., Hüttner, T.:** OpenGL-assisted Occlusion Culling of Large Polygonal Models. In: *Computer and Graphics - Special Issue on Visibility - Techniques and Applicatons*, 1999.

[Bartz et al. 2001] **Bartz, D., Staneker, D., Straßer, W., Cripe, B., Gaskins, T., Orton, K. Carter, M., Johannsen, A., Trom, J.:** Jupiter: A Toolkit for Interactive Large Model Visualization. In: *Proc. of Symposium on Parallel and Large Data Visualization and Graphics*, 129 – 134, 2001.

[Bern et al. 90] **Bern, M., Eppstein, D., Gilbert, J.** Provably good mesh generation. In: *Proc. 31st Annu. IEEE Sympos. Found. Compt. Sci.,* 231 – 241, 1990.

[Blinn 78a] **Blinn, J.F.:** *Simulation of Wrinkled Surfaces*. In: *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12, 286 – 292, 1978.

[Blinn 78b] **Blinn, J.F.:** *Computer display of curved surfaces*. Ph.d. thesis, University of Utah, 1978.

[Blinn 82] **Blinn, J.F.:** Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. In: *Computer Graphics (SIGGRAPH 82 Proceedings)*, 16(3), 21 – 29, 1982.

[Blinn and Newell 76] **Blinn, J.F., Newell,M.E.:** Texture and Reflection in Computer Generated Images. In: *Communications of the ACM*, 19:542–546, 1976.

[Boada et al. 2001] **Boada, I., Navazo, I., Scopigno, R.:** Multiresolution volume visualization with a texture-based octree. In: *The Visual Computer, 17(3)*, 185 – 197, Springer, 2001

[Bogomjakov and Gotsman 2002] **Bogomjakov, A., Gotsman, C.:** Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes. In: *Computer Graphics Forum* 21(2):137-148, 2002.

[Borland 95] **Borland Software Corporation:** Delphi 1.0, 1995.

[Botch et al. 2003] **Mario Botsch, Leif Kobbelt:** High-Quality Point-Based Rendering on Modern GPUs. In: *Pacific Graphics 2003 Conference Proceedings*, 2003.

[Botsch et al. 2002] **Botsch, M., Wiratanaya, A., Kobbelt, L.P.:** Efficient high quality rendering of point sampled geometry. In: *Rendering Techniques 2002*, Springer, 2002.

[Bronstein et al. 97] **Bronstein, I.N., Semendjajew, K.A., Musiol, G., Mühlig, H.:** *Taschenbuch der Mathematik, 3. Auflage*, Verlag Harri Deutsch, 1997.

[Callahan 95] **Callahan, P.B.** *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications.* Ph.D. thesis, John Hopkins University, 1995.

[Carpenter 84] **Carpenter, L.:** The A-buffer, an Antialiased Hidden Surface Method. In: *SIGGRAPH 84 Conference Proceedings*, 103-108, 1984.

[Catmull 74] **Catmull, E.** *A Subdivision Algorithm for Computer Display of Curved Surfaces.* Ph.D. Thesis, Report UTEC-CSc-74-133, Computer Science Department, University of Utah, Salt Lake City, UT, 1974.

[Chamberlain et al. 95] **Chamberlain, B., DeRose, T., Lischinski, D., Salesin, D., Snyder, J.:** *Fast Rendering of Complex Environments Using a Spatial Hierarchy*. Technical Report, UW-CSE-95-05-02, University of Washington, 1995.

[Chamberlain et al. 96] **Chamberlain, B., DeRose, T., Lischinski, D., Salesin, D., Snyder, J.:** Fast Rendering of Complex Environments Using a Spatial Hierarchy. In: *Proc. Graphics Interface '96*, 132-141, 1996.

[Chen and Nguyen 2001] **Chen,B., Nguyen, M.X.:** POP:A Hybrid Point and Polygon Rendering System for Large Data. In: V*isualization 2001 Conference Proceedings*, 2001.

[Chen and Williams 93] **Chen, S.E., Williams, L.:** View Interpolation for Image Synthesis. In: *Computer Graphics (SIGGRAPH '93 Proceedings)*, 27, 279–288, 1993.

[Christensen et al. 2003] **Christensen, P.H., Laur, D.M., Fong, J., Wooten, W.L., Batali, D.:** Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. In: *EUROGRAPHICS 2003 Conference Proceedings*, 2003.

[Cignoni et al. 2003] **Cignoni, P., Rocchini, C., Montani, C., Scopigno, R.:** External Memory Management and Simplification of Huge Meshes. In: *IEEE Trans. on Visualization and Computer Graphics* 9(4):525-537, 2003.

[Clark 76] **Clark, J.H.:** Hierarchical geometric models for visible surface algorithms. In: *Communications of the ACM, 19(10)*, 547-554, 1976.

[Coconu et al. 2002] **Coconu, L., Hege, H.:** Hardware-Accelerated Point-Based Rendering of Complex Scenes. In: *Rendering Techniques 2002*, Springer, 2002.

[Cohen et al. 98] **Cohen, J., Olano, M., Manocha D.:** Appearance-Preserving Simplfication. In: *SIGGRAPH 98 Proceedings, Annual Conference Series*, 1998.

[Cohen et al. 2001] **Cohen, J.D., Aliaga, D.G., Zhang, W.:** Hybrid Simplification: Combining Multi-Resolution Polygon and Point Rendering. In: V*isualization 2001 Conference Proceedings*, 2001.

[Cohen-Or et al. 2001] **Cohen-Or,D., Chrysanthou,Y., Koltun,V., Durand,F., Greene,N., Silva, C.T.:** Visibility, Problems, Techniques, and Applications. In: *SIGGRAPH 2001 Course Notes,* 2001.

[Collins 92] **Collins, S.:** Adaptive Splatting for Specular to Diffuse Light Transport. In: *Proceedings of the Fifth Eurographics Workshop on Rendering*, 119-135, Springer, 1992.

[Cook et al. 84a] **Cook, R.L., Porter, T., Carpenter, L.:** Distributed Ray-tracing. In: *SIGGRAPH 84 Proceedings*, 137-145, 1984.

[Cook 84b] **Cook, R.L.:** Shade Trees. In: *SIGGRAPH 84 Proceedings*, 223-231, 1984.

[Cook 86] **Cook, R.:** Stochastic sampling in computer graphics. In: *ACM Transactions on Graphics*, 5(1), 51-72, 1986.

[Cook et al. 87] **Cook, R.L., Carpenter, L., Catmull, E.:** The Reyes image rendering architecture, In: *SIGGRAPH 87 Proceedings*, 95-102, 1987.

[Csébfalvi and Szirmay-Kalos 2003] **Csébfalvi, B. Szirmay-Kalos, L.:** Monte Carlo Volume Rendering. In: *Visualization 2003 Conference Proceedings*, 2003.

[Csuri et al. 79] **Csuri, C., Hackathorn, R., Parent, R., Carlson, W., Howard, M.:** Towards an Interactive High Visual Complexity Animation System. In: *Computer Graphics (SIGGRAPH 79 Proceedings)*, 13(2), 289-298, 1979.

[Curious Labs 2001] **Curios Labs:** Poser 4.0, 2001. See: `http://www.curiouslabs.com/`

[Dachsbacher et al. 2003] **Dachsbacher, C., Vogelgsang, C., Stamminger, M.:** Sequential Point Trees. In: *SIGGRAPH 2003 Conference Proceedings*, 2003.

[de Berg et al. 94] **de Berg, M., Halperin, D., Overmars, M., Snoeyink, J., van Kreveld, M.:** Efficient Ray Shooting and Hidden Surface Removal. In: *Algorithmica, 12*, 30 – 53, 1994.

[de Berg et al. 97] **de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.:** *Computational Geometry - Algorithms and Applications,* Springer-Verlag 1997.

[Debevec et al. 96] **Debevec, P.E., Taylor, C.J., Malik, J.:** Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach. In: *SIGGRAPH 96 Proceedings*, 11-20, 1996.

[Debevec 98] **Debevec, P.:** Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics With Global Illumination and High Dynamic Range Photography. In: *SIGGRAPH 98 Proceedings*, 189-198, 1998.

High dynamic range images taken from: `http://www.debevec.org/Probes`

[Décoret et al. 2003] **Décoret, X., Durand, F., Sillion, F.X., Dorsey, J.:** Billboard Clouds for Extreme Model Simplification. In: *SIGGRAPH 2003 Conference Proceedings*, 2003.

[Demers and Malenfant 95] **Demers, F.N., Malenfant, J.:** Reflection in logic, functional and object-oriented programming: a short comparative study. In: *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, 29-38, 1995.

[Deussen et al. 98] **Deussen, O., Hanrahan, P., Lintermann, B., Mech, R., Pharr, M., Prusinkiewicz, P.:** Realistic Modeling and Rendering of Plant Ecosystems. In: *SIGGRAPH 98 Proceedings*, 1998.

[Deussen et al. 2002] **Deussen, O., Colditz, C., Stamminger, M., Drettakis, G.:** Interactive Visualization of Complex Plant Ecosystems. In: *Visualization 2002 Conference Proceedings*, 2002.

[Dey and Hudson 2002] **Dey, T.K., Hudson, J.:** PMR: Point to Mesh Rendering, A Feature-Based Approach. In: *Visualization 2002 Conference Proceedings*, 2002.

[Disney 94] **Disney Pictures:** *The Lion King*, 1994.

[Durand et al. 2000] **Durand,F., Drettakis, G., Thollot, J., Puech, C.:** Conservative Visibility Preprocessing using Extended Projections. In: SIGGRAPH 2000 Conference Proceedings, 2000.

[Eck et al. 95] **Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., Stuetzle, W.** Multiresolution Analysis of Arbitrary Meshes. In: *ACM Computer Graphics Proc., Annual Conference Series (SIGGRAPH '95 Proceedings),* 173-182, 1995.

[Encarnação et al. 96] **Encarnação, J., Straßer, W., Klein, R.:** *Graphische Datenverarbeitung I*, Oldenbourg Verlag, 1996.

[Encarnação et al. 97] **Encarnação, J., Straßer, W., Klein, R.:** *Graphische Datenverarbeitung II*, Oldenbourg Verlag, 1997.

[Figueiredo et al. 92] **Figueiredo, L.H., Gomes, J.M, Terzopoulos, D., Velho, L.:** Physically-Based Methods for Polygonization of Implicit Surfaces. In: *Graphics Interface '92 Proceedings*, 1992.

[Fischer et al. 98] **Fischer, M., Lukovszki, T., Ziegler, M.:** Geometric Searching in Walkthrough Animations with Weak Spanners in Real Time. In: *Proceedings of the 6th Annual European Symposium on Algorithms (ESA'98)*, LNCS, Springer, 163 – 174, 1998.

[Fleishman et al. 2003] **Fleishman, S., Cohen-Or, D., Alexa, M., Silva, C.T..** Progressive Point Set Surfaces. In: *ACM Transactions on Graphics*, 22(4), 2003.

[Foley et al. 96] **Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F.:** *Computer Graphics: principals and practice, second edition in C*. Addison-Wesley, 1996.

[Freeman 96] **Freeman, K.G.:** *Technique and System for the Real-Time Generation of Perspective Images*. United States Patent No. *5,550,959*, 1996.

[Friedrich et al. 98] **Friedrich, A., Polthier, K., Schmies, M.:** Interpolation of Triangle Hierarchies. In: *IEEE Visualization '98 Proceedings*, 1998

[Fujimoto et al. 1986] **Fujimoto, A., Tanaka, T., Iwata, K.:** ARTS: Accelerated Ray-Tracing System. In: *IEEE Computer Graphics and Applications* 6(4), 16-26, 1986.

[Funkhouser et al. 2002] **Funkhouser, T., Jot, J., Tsingos, N.:** "Sounds Good to Me!", Computational Sound for Graphics, Virtual Reality, and Interactive Systems. In: *SIGGRAPH 2002 Course Notes*, 2002.

[Gardner 85] **Gardner, G.Y.:** Visual Simulation of Clouds. In: *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19, 297–303, 1985.

[Garland and Heckbert 97] **Garland, M., Heckbert, P.S.:** Surface simplification using quadric error metrics. In: *SIGGRAPH 97 Proceedings*, 209 - 216, 1997.

Implementation ("QSlim") available online:
`http://graphics.cs.uiuc.edu/~garland/software/qslim.html`

[Garland 99] **Garland, M.:** Multiresolution Modeling: Survey & Future Opportunities. In: *EUROGRAPHICS 99 Proceedings, State of the Art Report*, 1999.

[Garland 2003] **Garland, M:** Sample Data (model collection), 2003.
`http://graphics.cs.uiuc.edu/~garland/research/quadrics.html`

[Genetti et al. 98] **Genetti, J., Gordon, D, Williams, G.:** Adaptive Supersampling in Object Space Using Pyramidal Rays. In: *Computer Graphics Forum*, 17(1), 29-54, 1998.

[Ghanzanfarpour and Hasenfratz 98] **Ghanzanfarpour, D., Hasenfratz, J.M.:** A Beam Tracing Method with Precise Antialiasing for Polyhedral Scenes. In: *Computer & Graphics*, 22(1), 103-115, 1998.

[Glassner 84] **Glassner, A.S.:** Space subdivision for fast ray tracing. In: *IEEE Computer Graphics and Applications*, 4(10), 15-22, 1984.

[Glassner 95] **Glassner, A. S.** *Principles of Digital Image Synthesis.* Morgen Kaufmann Publishers, 1995.

[Gortler et al. 96] **Gortler, S.J., Grzeszczuk, R., Szeliski, R., Cohen, M.F.:** The Lumigraph. In: *SIGGRAPH 96 Conference Proceedings*, 43-54, 1996.

[Gotsman et al. 2002] **Gotsman, C., Gumhold, S., Leif, K.:** Simplification and Compression of 3D Meshes. In: Iske, Armin and Quak, Ewald and Floater, Michael S. (editors): *Tutorials on Multiresolution in Geometric Modelling*, 319-362, Springer, 2002.

[Gotsman et al. 99] **Gotsman, C., Sudarsky, O., Fayman, J.:** Optimized occlusion culling. In: *Computer & Graphics*, 23(5):645–654, 1999.

[Greene et al. 93] **Greene, N., Kass, M., Miller, G.** Hierarchical Z-Buffer Visibility. In: *ACM Computer Graphics, Annual Conference Series (SIGGRAPH '93 Proceedings)*, 231-238, 1993.

[Grigoryan and Rheingans 2002] **Grigoryan, G., Rheingans, P.:** Probabilistic Surfaces: Point Based Primitives to Show Surface Uncertainty. In: *Visualization 2002 Conference Proceedings*, 2002.

[Grossman and Dally 98] **Grossman, J. P., Dally, W. J.:** Point Sample Rendering. In: *Rendering Techniques '98*, 181–192, Springer, 1998.

[Gu et al. 2002] **Gu, X., Gortler, S.J., Hoppe, H.:** Geometry Images. In: *SIGGRAPH 2002 Proceedings*, 2002.

[Gumhold and Straßer 98] **Gumhold, S. Straßer, W.:** Real time compression of triangle mesh connectivity. In: SIGGRAPH 98 Conference Proceedings, Annual Conference Series, pages 133 – 140. ACM SIGGRAPH, Addison Wesley, July 1998.

[Guthe et al. 2002] **Guthe, S., Wand, M., Gonser, J., Straßer, W.:** Interactive Rendering of Large Volume Data Sets. In: *Visualization 2002 Conference Proceedings*, 2002.

[Guthe et al. 2003] **Guthe, M., Borodin, P., Klein, R.:** Efficient View-Dependent Out-of-Core Visualization. In: *Proceedings of The 4th International Conference on Virtual Reality and its Application in Industry*, 2003.

[Guthe and Straßer 2004] **Guthe, S., Strasser, W.:** Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. In: *Computers & Graphics 28*, 51 – 58, 2004.

[Guthe 2004] **Guthe, S.:** *Compression and Visualization of Large and Animated Volume Data*. PhD thesis, University of Tübingen, to appear.

[Haumont et al. 2003] **Haumont, D., Debeir, O., Sillion, F.:** Volumetric Cell-and-Portal Generation. In: *Computer Graphics forum (EUROGRAPHICS 2003 Proceedings)*, 22(3), 2003.

[He et al. 95] **He, T., Hong, L., Kaufman, A., Varshney, A., Wang, S.** Voxel Based Object Simplification. In: *Visualization '95 Conference Proceedings,* 296-303, 1995.

[Heckbert and Hanrahan 84] **Heckbert, P., Hanrahan, P.:** Beam Tracing Polygonal Objects. In: *SIGGRAPH 84 Proceedings*, 119-127, 1984.

[Heckbert 86] **Heckbert, P.S.:** Survey of Texture Mapping. In: *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986.

[Heckbert 89] **Heckbert, P.S.:** *Fundamentals of Texture Mapping and Image Warping.* M.sc. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June 1989.

[Heckbert and Garland 94] **Heckbert, P., Garland, M.:** Multiresolution Modeling for Fast Rendering. In: *Proceedings of Graphics Interface '94*, 1994

[Heckbert and Garland 97] **Heckbert, P.S., Garland, M.** Survey of Polygonal Surface Simplification Algorithms, *SIGGRAPH 97 course notes*, 1997.

[Heidrich and Seidel 99] **Heidrich, W., Seidel, H.P.:** Realistic, Hardware-accellerated Shading and Lighting. In: *SIGGRAPH 99 Conference Proceedings*, 1999.

[Hopf and Ertl 2003] **Hopf, M., Ertl, T.:** Hierarchical Splatting of Scattered Data. In: *Visualization 2003 Conference Proceedings*, 2003.

[Hoppe et al. 93] **Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., Stuetzle, W.:** Mesh Optimization. In: *SIGGRAPH 93 Proceedings*, 1993.

[Hoppe 96] **Hoppe, H.:** Progressive meshes. In: *SIGGRAPH 96 Proceedings,* 99 – 108, 1996.

[Hoppe 97] **Hoppe, H.:** View-dependent refinement of progressive meshes. In: *SIGGRAPH 97 Proceedings*, 189 – 198, 1997.

[Hoppe 99] **Hoppe, H.:** Optimization of mesh locality for transparent vertex caching. In: *SIGGRAPH 99 Proceedings*, 269 – 276, 1999.

[Igehy 99] **Igehy, H.:** Tracing Ray Differentials. In: *SIGGRAPH 99 Proceedings*, 179 – 186, 1999.

[Infogrames 99] **Infogrames:** *Outcast,* 1999.

[Jensen et al. 2001] **Jensen, H.W., Christensen, P.H., Suykens, F.:** A Practical Guide to Global Illumination Using Photon Mapping. In: *SIGGRAPH 2001 Course Notes*, 2001.

[Jensen et al. 2003] **Jensen, H.W., Arvo, J., Dutré, P., Keller, A., Owen, A., Pharr, M., Shirley, P.:** Monte Carlo Ray Tracing. In: *SIGGRAPH 2003 Course Notes*, 2003.

[Kajiya 84] **Kajiya, J.T.:** Ray Tracing Volume Densities. In: *SIGGRAPH 84 Conference Proceedings*, 165–174, 1984.

[Kajiya 86] **J. Kajiya.** The rendering equation. In: *Proc. of SIGGRAPH 86*, pages 143–150. ACM SIGGRAPH, 1986.

[Kalaiah and Varshney 2001] **Kalaiah, A., Varshney, A.:** Differential Point Rendering. In: *Rendering Techniques 2001*, Springer, 2001.

[Kalaiah and Varshney 2003] **Kalaiah, A. Varshney, A.:** Statistical Point Geometry. In: *Proceedings of Eurographics Symposium on Geometry Processing*, 2003.

[Karabassi et al. 2003] **Karabassi, E.A., Papaioannou, G., Fretzagias, C., Theoharis, T.:** Exploiting multiresolution models to accelerate ray tracing. In: *Computer & Graphics (27)*, 91-98, 2003.

[Kay 79] **Kay, D. S.** *Transparency, Refraction and Ray Tracing for Computer Synthesized Images*. M.S. Thesis, Program of Computer Graphics, Cornell University, Ithaca, NY, 1979.

[Khodakovsky et al. 2003] **Khodakovsky, A., Litke, N., Schröder, P.:** Globally Smooth Parameterizations with Low Distortion. In: *SIGGRAPH 2003 Proceedings*, 2003.

[Kirk 87] **Kirk, D., The Simulation of Natural Features Using Cone Tracing.** In: *The Visual Computer*, 3(2), 63 – 71, Springer, 1987.

[Klein et al. 96] **Klein, R., Liebich, G., Straßer, W.:** Mesh Reduction with Error Control. In: *Proceedings of IEEE Visualization 96*, 311 – 318, 1996.

[Klein et al. 98a] **Klein, R., Cohen-Or, D., Hüttner, T.:** Incremental View-dependent Multiresolution Triangulation of Terrain. In: *The Journal of Visualization and Computer Animation*, 129 – 143, 1998.

[Klein et al. 98b] **Klein, R., Schilling, A.G., Straßer, W.:** Illumination Dependent Refinement of Multiresolution Meshes. In: *Proceedings of Computer Graphics International*, 680 – 687, IEEE Comput. Soc., 1998.

[Klein 99] **Klein, R.:** *Handling of Large 3D-Surface-Datasets Using Mesh Simplification and Multiresolution Modeling*, Habilitation, Wilhelm-Schickard-Institut für Informatik, Graphisch-Interaktive Systeme (WSI/GRIS), Universität Tübingen, 1999.

[Klein et al. 2002] **Klein, J., Krokowski, J., Fischer, M., Wand, M., Wanka, R. Meyer auf der Heide, F.:** The Randomized Sample Tree: A Data Structure for Interactive Walkthroughs in Externally Stored Virtual Environments. In: *Proceedings of ACM Symposium on Virtual Reality Software and Technology (VRST)*, 2002.

[Klein et al. 2004] **Klein, J., Krokowski, J., Fischer, M., Wand, M., Wanka, R., Meyer auf der Heide, F.:** The Randomized Sample Tree: A Data Structure for Externally Stored Virtual Environments In: *Presence 13(6)*, MIT Press, December 2004, to appear.

[Köckler 94] **Köckler, N.:** *Numerical Methods and Scientific Computing*, Clarendon Press, 1994.

[Lacroute 94] **Lacroute, P., Levoy, M.:** Fast volume rendering using a shear-warp factorization of the viewing transformation. In: *Computer Graphics, 28 (Annual Conference Series)*, 451 – 458, 1994.

[LaMar et al. 99] **LaMar, E.C., Hamann, B., Joy, K.I.:** Multiresolution techniques for interactive texture-based volume visualization. In: *IEEE Visualization '99* Proceedings, 355 – 362, 1999.

[Lane et al. 80] **Lane, J.M., Carpenter, L.C., Whitted, T., Blinn, J.F.:** Scan Line Methods for Displaying Parametrically Defined Surfaces. In: *Communications of the ACM*, 23(1), 23 – 34, 1980.

[Lapré 2002] **Lapré, L.:** "*lparser*" freeware L-systems parsing package

Available online:
`http://home.wanadoo.nl/laurens.lapre/lparser.htm`

[Laur and Hanrahan 91] **Laur, D., Hanrahan, P.:** Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In: *Computer Graphics (SIGGRAPH 91 Proceedings)*, 25(4), 285 – 288, 1991.

[Lee et al. 85] **Lee, M.E., Redner, R.A., Uselton, S.P.:** Statistically Optimized Sampling for Distributed Ray Tracing. In: *SIGGRAPH 85 Proceedings*, 61-67, 1985.

[Levoy et al. 2000] **Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., Fulk, D.:** The Digital Michelangelo Project: 3D Scanning of Large Statues. In: *SIGGRAPH 2000 Proceedings,* 2000.

[Levoy and Hanrahan 96] **Levoy, M., Hanrahan, P.:** Light Field Rendering. In: *SIGGRAPH 96 Proceedings, Annual Conference Series*, 31-42, 1996.

[Levoy and Whitted 85] **Levoy, M., Whitted, T.:** *The Use of Points as a Display Primitive.* Technical report, University of North Carolina at Chapel Hill, 1985.

[Leyvand et al. 2003] **Leyvand, T., Sorkine, O., Cohen-Or, D.:** Ray Space Factorization for From-Region Visibility. In: *SIGGRAPH 2003 Conference Proceedings*, 2003.

[Lindholm et al. 2001] **Lindholm, E., Kilgard, M.J., Moreton, H.:** A User-Programmable Vertex Engine. In: *SIGRAPH 2001 Proceedings*, 2001.

[Lindstrom 2000] **Lindstrom, P.:** Out-of-Core Simplification of Large Polygonal Models. In: *SIGGRAPH 2000 Proceedings*, 2000.

[Lischinski and Rappoport 98] **Lischinski, D., Rappoport, A.:** Image-based rendering for non-diffuse synthetic scenes. In: *Rendering Techniques '98*, 301 – 314, Springer, 1998.

[Lorensen and Cline 87] **Lorensen, W.E., Cline, H.E.** Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: *ACM Computer Graphics Proc., Annual Conference Series (SIGGRAPH 87 Proceedings),* 21 (4), 163 – 169, 1987.

[Lucasfilm 84] **Lucas Film Ldt.:** *The Adventures of André and Wally B.*, 1984.
Available online:
http://www.pixar.com/shorts/awb/index.html

[Lucente and Galyean 95] **Lucente, M., Galyean, T.A.:** Rendering Interactive Holographic Images. In: *Computer Graphics, Annual Conference Series (SIGGRAPH 95 Proceedings)*, 387–394, ACM SIGGRAPH, 1995.

[Luebke and Georges 95] **Luebke, D.P., Georges, C.:** Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In: *Proceedings of the 1995 Symposium on Interactive 3-D Graphics*, ACM, 1995.

[Luebke et al. 2003] **Luebke, D., Reddy, M., Cohen, J.D., Varshney, A., Watson, B., Huebner, R.:** *Level of Detail for 3D Graphics*, Morgan Kaufmann Publishers, 2003.

[Maciel and Shirley 95] **Maciel, P.W.C., Shirley, P.:** Visual Navigation of Large Environments Using Textured Clusters. In: *Proceedings of the 1995 ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 1995.

[Mao 96] **MAO, X.:** Splatting of non-rectilinear volumes through stochastic resampling. In: *IEEE Transactions on Visualization and Computer Graphics*, 2(2), 156–170, 1996.

[Mark et al. 97] **Mark, W.R., McMillan, L., Bishop, G.:** Post-Rendering 3D Warping. In: *Symposium on Interactive 3D Graphics*, 7–16, 1997.

[Matusik et al. 2002] **Matusik, W., Pfister, H., Ngan, A., Beardsley, P., Ziegler, R., McMillan, L.:** Image-Based 3D Photography using Opacity Hulls. In: *SIGGRAPH 2002 Conference Proceedings*, 2002.

[Max 95a] **Max, N.:** Optical Models for Direct Volume Rendering. In: *IEEE Transactions on Visualization and Computer Graphics 1(2)*, 99–108, June 1995.

[Max and Ohsaki 95b] **Max, N., Ohsaki, K.:** Rendering Trees from Precomputed Z-Buffer Views. In: *Rendering Techniques '95*, 74–81. Springer, 1995.

[Max 96] **Max, N.:** Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. In: *Rendering Techniques '96*, 165–174, Springer, 1996.

[McMillan and Bishop 95] **McMillan; L., Bishop, G.:** Plenoptic Modelling : An Image-Based Rendering System. In: *SIGGRAPH 95 Conference Proceedings*, 39-46, 1995.

[Meyer et al. 2002] **Meyer, M., Desbrun, M., Schröder, P., Barr, A.H.:** Discrete Differential-Geometry Operators for Triangulated 2-Manifolds. In: *Proceedings of the International Workshop on Visualization and Mathematics*, 2002.

[Mitchell 87] **Mitchell, D.P.:** Generating antialiased images at low sampling densities. In: *SIGGRAPH 87 Proceedings*, 65-72, 1987.

[Mitchell and Netravali 88] **Mitchell, D.P., Netravali, A.N.:** Reconstruction Filters in Computer Graphics. In: *SIGGRAPH 88 Conference Proceedings*, 221-228, 1988.

[Mitchell 96] **Mitchell, D.P.:** Consequences Of Stratified Sampling In Graphics. In: *SIGGRAPH 96 Proceedings*, 277-280, 1996.

[Motwani and Raghavan 95] **Motwani, R., Raghavan, P.** *Randomized Algorithms*. Cambridge University Press, 1995.

[National Library of Medicine 2002] **The National Library of Medicine:** The Visible Human Project. `http://www.nlm.nih.gov/research/visible/visible_human.html`

[New Line 2003] **New Line Productions:** *The Lord of the Rings*, 2003.

[Neyret 96] **Neyret, F.:** Synthesizing Verdant Landscapes using Volumetric Textures. In: *Rendering Techniques '96*, 215-224, Springer, 1996.

[Nirenstein et al. 2002] **Nirenstein, S., Blake, E., Gain, J.:** Exact From-Region Visibility Culling. In: *Rendering Techniques 2002*, Springer 2002.

[Novalogic 92] **Novalogic Inc.:** *Comanche,* 1992.

[nVidia 2004a] **NVidia Developers Relations:** `http://developers.nvidia.com`, 2004.

[nVidia 2004b] **NVidia Cooperation:** CineFX 3.0, The Next Wave of Stunning Visual Effects, Technical Brief, 2004. `http://www.nvidia.com/object/feature_cinefx3.0.html`

[Olieveira et al. 2000] **Oliveira, M.M., Bishop, G., McAllister, D.:** Relief Texture Mapping. In: *SIGGRAPH 2000 Conference Proceedings*, 2000.

[Papadimitriou 94] **Papadimitriou, C.H.:** *Computational Complexity*. Addison-Wesley, 1994.

[Paramount 82] **Paramount Pictures:** *Start Trek II: The Wrath of Khan*, 1982.

[Pauly and Gross 2001] **Pauly, M., Gross, M.:** Spectral Processing of Point Sampled Geometry. In: *SIGGRAPH 2001 Conference Proceedings*, 2001.

[Pauly et al. 2002] **Pauly, M., Gross, M., Kobbelt, L.P.:** Efficient Simplification of Point-Sampled Surfaces. In: *Visualization 2002 Conference Proceedings*, 2002.

[Pauly et al. 2003a] **Pauly, M., Keiser, R., Kobbelt, L.P., Gross, M.:** Shape Modeling with Point-Sampled Geometry. In: *SIGGRAPH 2003 Conference Proceedings*, 2003.

[Pauly et al. 2003b] **Pauly, M., Keiser, R., Gross, M.:** Multi-scale Feature Extraction on Point-Sampled Surfaces. In: *EUROGRAPHICS 2003 Conference Proceedings*, 2003.

[Peter and Straßer 2001] **Peter, I., Straßer, W.:** The Wavelet Stream: Interactive Multi Resolution Light Field Rendering. In: *Rendering Techniques 2001*, 127–138. Springer, 2001.

[Pfister et al. 2000] **Pfister, H., Zwicker, M., van Baar, J., Gross, M.: Surfels:** Surface Elements as Rendering Primitives. In: *SIGGRAPH 2000 Proceedings, Annual Conference Series*, 335-342, 2000.

[Popescu et al. 2000] **Popescu, V., Eyles, J., Lastra, A., Steinhurst, J., England, N., Nyland, L.:** The WarpEngine: An Architecture for the Post-Polygonal Age. In: *SIGGRAPH 2000 Conference Proceedings*, 2000.

[Popovic and Hoppe 97] **Popovic, J., Hoppe, H.** Progressive simplicial complexes. In: *ACM Computer Graphics Proc., Annual Conference Series (SIGGRAPH 97 Proceedings)*, 217-224, 1997.

[Poulin et al. 2003] **Poulin, P., Stamminger, M., Duranleau, F., Frasson, M.C., Drettakis, G.:** Interactive Point-based Modeling of Complex Objects from Images. In: Graphics Interface 2003 Conference Proceedings, 2003.

[Praun et al. 2003] **Praun, E., Hoppe, H.:** Spherical Parameterization and Remeshing. In: *SIGGRAPH 2003 Proceedings*, 2003.

[Press et al. 95] **Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.:** *Numerical Recipes in C, Second Edition*, Cambridge University Press, 1995.

[Puppo and Scopigno 97] **Puppo, E., Scopigno, R.** Simplification, LOD and Multiresolution Principals and Applications. In: *EUROGRAPHICS '97 Tutorial Notes, 1997.*

[Rafferty et al. 98] **Rafferty, M.M., Aliaga,D.G., Lastra, A.A.:** 3D Image Warping in Architectural Walkthroughs. In: *Proceedings of VRAIS '98*, 228–233, 1998.

[Reeves 83] **Reeves, W. T.:** Particle Systems – A Technique for Modeling a Class of Fuzzy Objects. In: *Computer Graphics (SIGGRAPH 83 Proceedings)*, 17(3), 359-376, 1983.

[Reeves and Blau 85] **Reeves, W.T., Blau, R.:** Approximate and probabilistic algorithms for shading and rendering structured particle systems. In: *SIGGRAPH 85 Proceedings*, 313 – 322, 1985.

[Regan and Pose 94] **Regan, M., Pose, R.:** Priority rendering with a virtual reality address recalculation pipeline. In: *SIGGRAPH 94 Conference Proceedings*, 155 – 162, 1994.

[Ren et al. 2002] **Liu Ren, Hanspeter Pfister, Matthias Zwicker:** Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering. In: *EUROGRAPHICS 2002 Conference Proceedings*, 2002.

[Reynolds 87] **Reynolds, C.W.:** Flocks, herds and schools: A distributed behavioral model. In: *SIGGRAPH 87 Proceedings*, 25 – 34, 1987.

[Rezk-Salama 2000] **Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G., Ertl, T.:** Interactive Volume Rendering on Standard PC Graphics Hardware using Multi-Textures and Multi-Stage Rasterization. In: *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2000.

[Rivara 84] **Rivara, M.:** Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. In: *International Journal for Numerical Methods in Engineering*, 20, 1984.

[Rohlf and Helman 94] **Rohlf, J., Helman, J.:** IRIS performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In: *Siggraph 94 Proceedings*, 381 – 394, 1994.

[Ronfard and Rossignac 96] **Ronfard, R., Rossignac, J.:** Full-range approximation of triangulated polyhedra. In: *EUROGRAPHICS '96 Proceedings*, 1996.

[Rossignac and Borrel 93] **Rossignac, J., Borrel, P.** Multi-resolution 3D approximations for rendering complex scenes. In: B. Falcidieno and T.Kunii, editors, *Modeling in Computer Graphics: Methods and Applications,* pages 455-465, Springer-Verlag 1993.

[Rubin and Whitted 80] **Rubin, S., Whitted, T.:** A three-dimensional representation for fast rendering of complex scenes. In: *Computer Graphics*, 14(3), 110 – 116, 1980.

[Rusinkiewicz and Levoy 2000] **Rusinkiewicz, S., Levoy, M.:** Qsplat: A Multiresolution Point Rendering System for Large Meshes. In: *SIGGRAPH 2000 Proceedings*, 343 – 352, 2000.

[Samet 89] **Samet, H.:** *The Design and Analysis of Spatial Data Structures*. Addison-Wesley 1989.

[Schaufler 98] **Schaufler, G.:** Per-Object Image Warping with Layered Impostors. In: *Rendering Techniques '98*, 145–156, Springer, 1998.

[Schaufler and Jensen 2000] **Schaufler, G., Jensen, H.:** Ray Tracing Point Sampled Geometry. In: *Rendering Techniques 2000*, Springer, 2000.

[Schaufler et al. 2000] **Schaufler, G., Dorsey, J., Decoret, X., Sillion, F.X.:** Conservative Volumetric Visibility with Occluder Fusion. In: SIGGRAPH 2000 Conference Proceedings, 2000.

[Schilling and Straßer 93] **Schilling, A.G., Straßer, W.**: EXACT: Algorithm and Hardware Architecture for an Improved A-buffer. In: *SIGGRAPH 93 Proceedings*, 85 – 92, 1993.

[Schilling et al. 96] **Schilling, A.G., Knittel, G., Straßer, W.:** Texram: A Smart Memory for Texturing. In: *IEEE Computer Graphics & Applications*, 32-41, 1996.

[Schilling and Klein 98] **Schilling, A.G, Klein, R.:** Rendering of multiresolution models with texture. In: *Computers & Graphics*, 667 – 674, 1998.

[Schilling 2001] **Schilling, A.:** Antialiasing of Environment-Maps, In: *Computer Graphics Forum*, 20(1), 5-11, 2001.

[Schroeder et al. 92] **Schroeder, W. J., Zarge, J. A., Lorensen, W.E.** Decimation of Triangle Meshes. In: *ACM Computer Graphics Proc., Annual Conference Series (SIGGRAPH '92 Proceedings), 26(2),* 65 – 70, 1992

[SGI 2004] **Sillicon Graphics, Inc:** *OpenGL Extension Registry*, 2004.
`http://oss.sgi.com/projects/ogl-sample/registry/`

[Shade et al. 96] **Shade, J., Lischinski, D., Salesin, D. H., DeRose, T., Snyder, J.** Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In: *ACM Computer Graphics Proc., Annual Conference Series (SIGGRAPH 96 Proceedings),* 1996.

[Shade et al. 98] **Shade, J.W., Gortler, S. J., He, L., Szeliski, R.:** Layered Depth Images. In: *SIGGRAPH 98 Proceedings*, 231–242, 1998.

[Shamir et al. 2000] **Shamir, A., Valerio, P., Chandrajit, B.:** Multi-Resolution Dynamic Meshes with Arbitrary Deformations. In: *IEEE Visualization 2000 Proceedings*, 2000.

[Shinya et al. 87] **Shinya, M., Takahashi, T., Naito, S.:** Principles and applications of pencil tracing. In: *SIGGRAPH 87 Proceedings*, 45 – 54, 1987

[Shirley and Tuchman 91] **Shirley, P., Tuchman,A.:** A Polygonal Approximation for Direct Scalar Volume Rendering. In: *Proceedings of Volume Visualization 91*, 63–70, 1991.

[Sillion et al. 97] **Sillion, F., Drettakis, G., Bodelet, B.** Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery. In: *EUROGRAPHICS '97 Conference Proceedings,* 207 – 218, 1997.

[Silva et al. 2002] **Silva, C.T., Chiang, Y.J., El-Sana, J., Lindstrom, P.:** Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics. In: *Visualization 2002 Courses*, 2002.

[Snedecor and Cochran 67] **Snedecor, G.W., Cochran, W.G.:** *Statistical Methods*, Sixth Edition, Iowa State University Press, 1967.

[Square 2001] **Square Pictures:** *Final Fantasy, The Spirits Within,* 2001.

[Stamminger and Drettakis 2001] **Stamminger, M., Drettakis, G.:** Interactive Sampling and Rendering for Complex and Procedural Geometry. In: *Rendering Techniques 2001*, Springer, 2001.

[Staneker et al. 2003] **Staneker, D., Bartz, D., Meißner, M.:** Improving Occlusion Query Efficiency with Occupancy Maps. In: *Proc. of Symposium on Parallel and Large Data Visualization and Graphics*, 2003.

[Stanford 2004] **Stanford Computer Graphics Laboratory:** The Stanford 3D Scanning Repository, 2004.
`http://graphics.stanford.edu/data/3Dscanrep/`

[Straßer 74] **Straßer, W.:** *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten*. Ph.D. Thesis, Technische Universität Berlin, 1974.

[Sudarsky and Gotsman 96] **Sudarsky, O., Gotsman, C.:** Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. In: *Computer Graphics Forum (EUROGRAPHICS 96 Proceedings)*, 15(3), 249 – 258, 1996.

[Sun 97] **Sun Microsystems:** *Java Beans API Specification*, 1997.
`http://java.sun.com/products/javabeans/`

[Szeliski and Tonnesen 92] **Szeliski, R., Tonnesen, D.:** Surface Modeling with Oriented Particle Systems. In: *SIGGRAPH 92 Proceedings*, 185-194, 1992.

[Szirmay-Kalos and Márton 98] **Szirmay-Kalos, L., Márton, G.** Worst-Case Versus Average Case Complexity of Ray-Shooting. In: *Journal of Computing*, 1998.

[Tecchia and Chrysanthou 2001] **Tecchia, F., Chrysanthou, Y.:** Real-Time Rendering of Densely Populated Urban Environments. In: *Rendering Techniques 2001*.

[Teller and Hohmeyer 93] **Teller, S.J., Hohmeyer, M.:** Stabbing Oriented Convex Polygons in Randomized $O(n^2)$ Time. In: CONM Vol. 178: Proc. Jerusalem Combinatorics '93, American Mathematical Society, 311 – 318, 1993.

[Teller and Séquin 91] **Teller, S.J., Séquin, C.H.** Visibility Preprocessing For Interactive Walkthroughs. In: *Computer Graphics Proc., Annual Conference Series (SIGGRAPH 91 Proceedings), 25 (4)*, 61 – 69, 1991.

[Torborg and Kajiya 96] **Torborg, J., Kajiya, J.** Talisman: Commodity realtime 3D graphics for the PC. In: *ACM Computer Graphics Proc., Annual Conference Series (SIGGRAPH 96 Proceedings),* 353 – 363, 1996.

[Trojansky 2001] **Trojansky, S.:** Destroying a Dam Without Getting Wet: Rule-Based Dynamic Simulation for "Wave of Death". In: *SIGGRAPH 2001 Sketches & Applictions*, 2001.

[Turk 92] **Turk, G.:** Re-Tiling Polygonal Surfaces. In: *ACM Computer Graphics, Annual Conference Series (SIGGRAPH '92 Proceedings)*, 26 (2), 55-64, 1992

[Universal 2001] **Universal Pictures:** *The Mummy Returns*, 2001.

[Varshney 94] **Varshney, A:** *Hierarchical Geometric Approximations*. PhD thesis, Department of Computer Science, University of North Carolina, TR-050, 1994.

[Wald et al. 2001] **Wald, I., Benthin, C., Wagner, M., Slusallek, P.:** Interactive Rendering with Coherent Ray-Tracing. In: *Computer Graphics Forum (EUROGRAPHICS 2001 Proceedings)*, 20(3), 153-164, 2001.

[Wald et al. 2003] **Wald, I., Purcell, T.J., Schmittler, J., Benthin, C., Slusallek, P.:** Realtime Ray Tracing and its use for Interactive Global Illumination. In: *Eurographics State of the Art Reports*, 2003.

[Wand 2000a] **Wand, M.:** *Approximation dreidimensionaler Szenen mit dem randomisierten z-Buffer Algorithmus*. Diploma thesis, Heinz-Nixdorf Institute for computer science, University of Paderborn, March 2000.

Available online:
`http://www.gris.uni-tuebingen.de/areas/pbr/rzbuffer/diplomarbeit.ps.zip`
(in German only, for an english summary, please refer to the technical report version [Wand et al. 2000b])

[Wand et al. 2000b] **Wand, M., Fischer, M., Meyer auf der Heide, F.:** *Randomized Point Sampling for Output-Sensitive Rendering of Complex Dynamic Scenes*. Technical report WSI-2000-20, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, November 2000.

Available online:
`http://www.gris.uni-tuebingen.de/publics/paper/Wand-2000-Randomized.pdf`

[Wand et al. 2001] **Wand, M., Fischer, M., Peter, I., Meyer auf der Heide, F., Straßer, W.:** The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In: *SIGGRAPH 2001 Proceedings, Annual Conference Series*, 361 – 370, 2001.

[Wand and Straßer 2002] **Wand, M., Straßer, W.:** Multi-Resolution Rendering of Complex Animated Scenes, In: *EUROGRAPHICS 2002 Conference Proceedings* (best student paper award), 2002.

[Wand and Straßer 2003a] **Wand, M., Straßer, W.:** Multi-Resolution Point-Sample Raytracing. In: *Graphics Interface 2003 Conference Proceedings*, 2003.

[Wand and Straßer 2003b] **Wand, M., Straßer, W.:** Real-Time Caustics. In: *EUROGRAPHICS 2003 Conference Proceedings*, 2003.

[Wand and Straßer 2003c] **Wand, M., Straßer, W.:** *A Real-Time Sound Rendering Algorithm for Complex Scenes*. Technical report WSI-2003-5, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, 2003.

Available online:
`http://www.gris.uni-tuebingen.de/publics/paper/Wand-2003-ARealTime.pdf`

[Warnock 72] **Warnock, T.T.:** Computational investigation of low-discrepancy point sets. In: S. Zaremba, editor, *Applications of Number Theory to Numerical Analysis*, 319-343, Academic Press, 1972.

[Wei and Levoy 2000] **Wei, L.Y., Levoy, M.:** Fast Texture Synthesis using Tree-structured Vector Quantization. In: *SIGGRAPH 2000 Proceedings*, 2000.

[Weiler et al. 2000] **Weiler, M., Westermann, R., Hansen, C. Zimmerman, K., Ertl, T.:** Level-of-detail volume rendering via 3d textures. In: *IEEE Volume Visualization and Graphics Symposium*, 2000.

[Welsh and Mueller 2003] **Welsh, T., Mueller, K.:** A Frequency-Sensitive Point Hierarchy for Images and Volumes. In: *Visualization 2003 Conference Proceedings*, 2003.

[Wernecke 94] **Wernecke, J.:** *The Inventor Mentor: Programming Object-Oriented 3d Graphics With Open Inventor, Release 2*. Addison Wesley, 1994.

[Westover 90] **Westover, L.:** Footprint Evaluation for Volume Rendering. In: *Computer Graphics (SIGGRAPH 90 Proceedings)*, 24, 367–376, 1990.

[Whitted 80] **Whitted, T.** „An Improved Illumination Model for Shaded Display“. In: *Communications of the ACM,* 23(6), 343–349, 1980.

[Williams 79] **Williams, R.:** *The Geometrical Foundation of Natural Structure: A Source Book of Design*. New York: Dover, 1979.

[Wimmer et al. 2001] **Wimmer, M., Wonka, P., Sillion, F.:** Point-Based Impostors for Real-Time Visualization. In: *Rendering Techniques 2001*, Springer, 2001.

[Winner et al. 97] **Winner, S., Kelley, M., Pease, B., Rivard, B., Yen, A.:** Hardware Accelerated Rendering Of Antialiasing Using A Modified A-buffer Algorithm. In: *SIGGRAPH 97 Conference Proceedings,1997.*

[Witkin and Heckbert 94] **Witkin, A.P., Heckbert, P.S.:** Using particles to sample and control implicit surfaces. In: *SIGGRAPH 94 Proceedings*, 269–277, 1994.

[Wloka and Huddy 2003] **Wloka, M., Huddy, R.:** *DirectX 9 Performance: Where does it come from, and where does it all go?* Game Developers Conference 2003 Presentation. Available online at `http://www.ati.com/developer`.

[Wood et al. 2000] **Wood, D.N., Azuma, D.I., Aldinger, K., Curless, B., Duchamp, T., Salesin, D.H., Stuetzle, W.:** Surface Light Fields for 3D Photography. In: *SIGGRAPH 2000 Proceedings*, 2000.

[Würmlin et al. 2003] **Würmlin, S., Lamboray, E., Gross, M.:** *3D video fragments: Dynamic point samples for real-time free-viewpoint video*. Technical Report No. 397, Institute of Scientific Computing, ETH Zurich.

[Yu et al. 99] **Yu, Y., Debevec, P., Malik, J., Hawkins, T.:** Inverse Global Illumination: Recovering Reflectance Models of Real Scenes From Photographs. In: *SIGGRAPH 99 Conference Proceedings*, 215 – 224, 1999.

[Zaremba 68] **Zaremba, S.:** The mathematical basis of Monte Carlo and quasi-Monte Carlo Methods. In: *SIAM Review*, 10, 303 – 314, 1968.

[Zwicker et al. 2000] **Zwicker, M., Gross, M.H., Pfister, H.:** *A Survey and Classification of Real Time Rendering Methods*, Technical Report 2000-09, Mitsubishi Electric Research Laboratories, March 2000.

[Zwicker et al. 2001a] **Zwicker, M., Pfister, H., van Baar, J., Gross, M.:** Surface Splatting. In: *SIGGRAPH 2001 Proceedings, Annual Conference Series*, 371-378, 2001.

[Zwicker et al. 2001b] **Zwicker, M., Pfister, H., Baar, J., Gross, M.:** EWA Volume Splatting. In: *Visualization 2001 Conference Proceedings*, 2001.

[Zwicker et al. 2002a] **Zwicker, M., Pauly, M., Knoll, O., Gross, M.:** Pointshop 3D: An Interactive System for Point-Based Surface Editing. In: *SIGGRAPH 2002 Conference Proceedings*, 2002.

[Zwicker 2002b] **Zwicker, M.**, *personal communications*, 2002.