Realtime Ray Tracing and Interactive Global Illumination

Ingo Wald

Computer Graphics Group
Saarland University
Saarbrücken, Germany

Dissertation zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Betreuender Hochschullehrer / Supervisor:
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Gutachter / Reviewers:
Philipp Slusallek, Prof. Dr.-Ing., Universität des Saarlandes, Saarbrücken, Germany
Peter Shirley, Associate Professor, University of Utah, Salt Lake City, UT, USA
Philip Dutre, Assistant Professor, Katholieke Universiteit Leuven, Belgium

Dekan / Dean:
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Eingereicht am / Thesis submitted
21. Januar 2004 / Jan 21st, 2004

Datum des Kolloquiums / Date of defense:
3. Mai 2004 / May 3rd, 2004

Ingo Wald
Computer Graphics Group (AG4)
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
`wald@mpi-sb.mpg.de`

Abstract

One of the most sought-for goals in computer graphics is to generate "realism in real time", i.e. the generation of realistically looking images at realtime frame rates. Today, virtually all approaches towards realtime rendering use graphics hardware, which is based almost exclusively on triangle rasterization. Unfortunately, though this technology has seen tremendous progress over the last few years, for many applications it is currently reaching its limits in both model complexity, supported features, and achievable realism.

An alternative to triangle rasterization is the ray tracing algorithm, which is well-known for its higher flexibility, its generally higher achievable realism, and its superior scalability in both model size and compute power. However, ray tracing is also computationally demanding and thus so far is used almost exclusively for high-quality offline rendering tasks.

This dissertation focuses on the question why ray tracing is likely to soon play a larger role for interactive applications, and how this scenario can be reached. To this end, we discuss the RTRT/OpenRT realtime ray tracing system, a software based ray tracing system that achieves interactive to realtime frame rates on todays commodity CPUs. In particular, we discuss the overall system design, the efficient implementation of the core ray tracing algorithms, techniques for handling dynamic scenes, an efficient parallelization framework, and an OpenGL-like low-level API. Taken together, these techniques form a complete realtime rendering engine that supports massively complex scenes, highly realistic and physically correct shading, and even physically based lighting simulation at interactive rates.

In the last part of this thesis we then discuss the implications and potential of realtime ray tracing on global illumination, and how the availability of this new technology can be leveraged to finally achieve interactive global illumination – the physically correct simulation of light transport at interactive rates.

Kurzfassung

Eines der wichtigsten Ziele der Computer-Graphik ist die Generierung von “Realismus in Echtzeit” – die Erzeugung von realistisch wirkenden, computer-generierten Bildern in Echtzeit. Heutige Echtzeit-Graphikanwendungen werden derzeit zum überwiegenden Teil mit schneller Graphik-Hardware realisiert, welche zum aktuellen Stand der Technik fast ausschliesslich auf dem Dreiecksrasterisierungsalgorithmus basiert. Obwohl diese Rasterisierungstechnologie in den letzten Jahren zunehmend beeindruckende Fortschritte gemacht hat, stößt sie heutzutage zusehends an ihre Grenzen, speziell im Hinblick auf Modellkomplexität, unterstützte Beleuchtungseffekte, und erreichbaren Realismus.

Eine Alternative zur Dreiecksrasterisierung ist das “Ray-Tracing” (Stahl-Rückverfolgung), welches weithin bekannt ist für seine höhere Flexibilität, seinen im Großen und Ganzen höheren erreichbaren Realismus, und seine bessere Skalierbarkeit sowohl in Szenengröße als auch in Rechner-Kapazitäten. Allerdings ist Ray-Tracing ebenso bekannt für seinen hohen Rechenbedarf, und wird daher heutzutage fast ausschließlich für die hochqualitative, nicht-interaktive Bildsynthese benutzt.

Diese Dissertation behandelt die Gründe warum Ray-Tracing in näherer Zukunft voraussichtlich eine größere Rolle für interaktive Graphikanwendungen spielen wird, und untersucht, wie dieses Szenario des Echtzeit Ray-Tracing erreicht werden kann. Hierfür stellen wir das RTRT/OpenRT Echtzeit Ray-Tracing System vor, ein software-basiertes Ray-Tracing System, welches es erlaubt, interaktive Performanz auf heutigen Standard-PC-Prozessoren zu erreichen. Speziell diskutieren wir das grundlegende System-Design, die effiziente Implementierung der Kern-Algorithmen, Techniken zur Unterstützung von dynamischen Szenen, ein effizientes Parallelisierungs-Framework, und eine OpenGL-ähnliche Anwendungsschnittstelle. In ihrer Gesamtheit formen diese Techniken ein koplettes Echtzeit-Rendering-System, welches es erlaubt, extrem komplexe Szenen, hochgradig realistische und physikalisch korrekte Effekte, und sogar physikalisch-basierte Beleuchtungssimulation interaktiv zu berechnen.

Im letzten Teil der Dissertation behandeln wir dann die Implikationen und das Potential, welches Echtzeit Ray-Tracing für die Globale Beleuchtungssimulation bietet, und wie die Verfügbarkeit dieser neuen Technologie benutzt werden kann, um letztendlich auch Globale Belechtung – die physikalisch korrekte Simulation des Lichttransports – interaktiv zu berechnen.

Acknowledgements

This thesis would have been impossible without the support of many people:

First of all, I would like to thank my supervisor Philipp Slusallek, for confronting me with the idea of interactive ray tracing and attracting me to the topic of realtime rendering; for continuously pushing me towards new challenges, and for always being available for valuable help and suggestions.

Second, I have to pay credit to Carsten Benthin, who has been an invaluable help in building the RTRT/OpenRT system, especially in (though not limited to) the networking code and in low-level optimizations, and who has played an integral part in many of the projects related to this thesis.

Similarly, I have to thank (in random order) Andreas Dietrich, for always helping out once help was needed; Alexander Keller for originally introducing me to the field of graphics, and for teaching me most of the bells and whistles of ray tracing and global illumination already long before my PhD; Philippe Bekaert for his XRML library, great discussions and help in many cases; Timothy J. Purcell for insight into his ideas, mind-opening discussions, and his invaluable help in the EG2003 Star Report; Alexander Keller and Thomas Kollig for their work on the Instant Global Illumination technique; Jörg Schmittler, Andreas Pomi, Gerd Marmitt, Tim Dahmen and Markus Wagner for the joint projects; and Vlastimil Havran for access to his vast knowledge about previous research on ray tracing. I would also like to thank all those – both former and current – members of the Max-Planck-Institute for Computer Science (MPII) that have provided help, insight into their research and ideas, and tough discussions.

Furthermore, I know very well how much I owe to Georg Demme and his system administration group, without whose continuous day-and-night battle for keeping our hardware resources alive this kind of research would hardly have happened as it did.

Special thanks also go to the people at Intel Corp., especially to James T. Hurley for inviting me over as an intern and thereby introducing me to many people and to even more new ideas, to Alexander Reshetov for his help and teaching on BSP tree construction, to Gordon Stoll for many discussions, and to all the other people at Intel that have certainly broadened my view, taught me a great deal, and made the stay there a great experience.

I would also like to thank my reviewers, Peter Shirley and Phil Dutré, for kindly accepting to review this thesis.

Finally, and most importantly, I would like to thank my family – my wife Gabriele and my little son Janne Lukas – who had to bear the extra stress of having me write this thesis, and without whose great patience and support this thesis would never have been possible.

Contents

List of Figures

List of Tables

Part I

Interactive Ray Tracing

Chapter 1
Introduction

Over the last 20 years, computer graphics has matured from a mostly academic discipline to a practically and commercially important business. The driving forces behind this development are for example virtual reality and visualization applications, the cinema and movie industry, and last but not least the ever more important game and entertainment industry. Looking at todays main uses of computer graphics, there are two fundamentally different kinds of applications: High-quality, offline computer graphics on one side, and fast, interactive graphical applications on the other side.

Interactive computer graphics – as used for example in games and virtual reality applications – today is almost entirely governed by triangle rasterization (like OpenGL and DirectX), running on extremely fast and increasingly sophisticated commodity graphics chips like ATI's Radeon and NVidia's GForce series. Due to several limitations of triangle rasterization, interactive computer graphics usually has to rely on approximations, and makes no claims on physical correctness. Though many interactive applications achieve stunning rendering quality, this is usually due to 'faking' of certain effects, and often requires significant manual effort by artists and game designers for the proper design and tuning of the respective scenes and applications.

On the other side, there is *offline rendering* for high-quality graphics and physically-correct rendering, which has many applications for example in the design, animation, and movie industries, and which allows for creating computer-generated images that can be virtually impossible to distinguish from "real" photographs. Due to strict requirements on quality and correctness, almost all these applications build on ray tracing.

Today, these two approaches to computer graphics are strictly separated, with high-quality rendering limited to offline use, and interactive graphics limited in achievable realism and correctness. Bridging this gap requires to either remove some fundamental limitations of triangle rasterization – which most researchers deem impossible – or to accelerate ray tracing to the point where it allows for realtime applications.

For more than a decade now, different researchers have argued that due to several inherent advantages of ray tracing – coupled with ever-increasing availability of hardware resources and ever-increasing user demands – ray tracing performance should eventually overtake triangle rasterization [Kajiya86, Teller98]. Though these claims are yet unfulfilled for almost two decades, it seems that with some of the recently ongoing developments this scenario is finally taking on shape. In the long term, realtime ray tracing should allow for *both* higher quality *and* higher performance than any other kind of rendering technology.

1.1 Outline of This Thesis

This thesis is structured into three independent parts, reporting on an introduction to interactive ray tracing, the RTRT/OpenRT interactive ray tracing system, and the Instant Global Illumination method, respectively.

Part I starts by introducing the ray tracing method in Chapter 2, and gives a brief survey of ray tracing acceleration techniques in Section 3. Chapter 4 then discusses the benefits of using ray tracing for interactive applications, and addresses the question why in the near future ray tracing is likely to play a larger role in interactive graphics. Part I then ends with an overview over the most important currently ongoing approaches towards realizing realtime ray tracing on different kinds of hardware platforms, which include various software systems, GPU-based ray tracing, and special purpose ray tracing hardware.

The main part of this thesis (Part II) then describes one of these approaches (the RTRT/OpenRT software realtime ray tracing system) in more detail: Chapter 6 starts with a detailed discussion of the issues to be kept in mind when designing a realtime ray tracing system on todays CPUs, and outlines the most fundamental design guidelines of the RTRT core. Chapter 7 then in detail describes the technical aspects of the RTRT core, namely fast ray/triangle intersection, fast BSP traversal, the efficient use of SIMD instructions, and high-quality BSP construction. Chapter 8 then shows how these fast core algorithms can be efficiently parallelized on small clusters of commodity PCs, resulting in near-linear scalability and high ray tracing performance without the need for sophisticated hardware resources. Following this, Chapters 9 and 10 discuss some advanced ray tracing issues, like how to efficiently handle dynamic scenes, and API issues for realtime ray tracing. Together, these chapters describe the different *building blocks* that form a complete rendering engine, the RTRT/OpenRT realtime ray tracing engine, which already enables several new applications. Some examples of such applications are briefly summarized in Section 11.

Part III of this thesis then discusses how realtime ray tracing can eventually be used to finally achieve *interactive global illumination* – the physically-correct simulation of global light transport in a virtual scene at interactive rates. Section 12.1 first briefly summarizes several alternative approaches to interactive global illumination. Following this, Section 12.2 outlines the issues and the constraints arising when trying to map contemporary global illumination algorithms to a realtime ray tracing engine, and concludes that most of these approaches are not suitable to this setting.

Based on this discussion, Section 13 then introduces the *Instant Global Illumination* method, a specially designed method that fits these constraints. Following this, Section 14 discusses some extensions of instant global illumination that aim at also applying the technique to complex and highly occluded scenes.

Finally, this thesis ends with a short summary, and with a brief survey of potential future work in realtime ray tracing and in interactive global illumination.

Chapter 2

An Introduction to Ray Tracing

Before going into any details on ray tracing, it is necessary to first introduce several terms, and to actually define the problem of ray tracing. This may seem superfluous due to the fact that ray tracing is a very commonly known technique taught in about every computer graphics course. However, the term "ray tracing" actually covers a wide range of different meanings, ranging from the basic *concept* of efficiently finding an intersection between a ray and a set of primitives, over the classical recursive ray tracing *rendering algorithm* (and its variants), down to more general *ray tracing based algorithms* that use ray tracing in one or another form. Note that all these terms are fundamentally different. For example, many global illumination algorithms use "ray tracing" for computing images, but otherwise have very few in common with the classical rendering algorithm.

2.1 The Core Concept – Ray Shooting

The core concept of any kind of ray tracing algorithm is to efficiently find intersections of a ray with a scene consisting of a set of geometric primitives. The ray $R(t) = O + tD$ is usually described by its origin O and direction D, and may additionally have a parameter t_{max} that specifies a "maximum distance" up to which the ray is looking for intersecting objects. Only object intersections with distance $t_{hit} < t_{max}$ are considered valid intersections[1].

[1] In practice, rays often originate *on* surfaces. This may lead to the undesired effect of finding this originating geometry as the first "intersection". To avoid this so-called "self-occlusion", valid hit distances have to be strictly greater than zero (i.e. $t_{hit} \in (0, t_{max})$). Due to numerical issues, practical implementations usually limit the valid range to $t_{hit} \in (\epsilon, t_{max})$.

Already at this level, the term “ray tracing” actually covers three different problems that have to be solved: Finding *the closest* intersection to the origin, finding *any* intersection along the ray, and finding *all* intersections along it.

Finding only the closest intersection is the most fundamental operation in any ray tracer. It usually requires to find the closest intersecting primitive P and its distance t_{hit}. Additionally, most ray tracers also determine additional parameters that are later-on used for shading the ray, such as local surface properties or the surface normal.

A slightly simpler problem is to determine whether there is *any* primitive that intersects the ray. This is actually the same as checking for visibility between the two points O and $O + t_{max}D$. As such, this operation is often termed “visibility test” or “occlusion test”. Obviously, checking for *any* intersection is a slightly simpler problem than finding *the closest* intersection. Thus, there are algorithms that are more efficient for this special case than for the general ray shooting case. As checking for occlusion is a very common operation in a ray tracer, most ray tracers have specially optimized routines for this task. Most obviously, “normal” rays might have to perform multiple intersection tests in order to determine which of these intersections in the closest one, whereas intersection testing for shadow rays can be terminated as soon as the first successful intersection test has been performed. Except for this obvious optimization, there are also several optimizations and heuristics for accelerating shadow rays, such as Shadow Caching, Shadow Maps, Adaptive Shadow Testing, Local Illumination Environments, and others (see Section 3.2).

The third sub-problem in ray tracing is finding *all* intersections with a given ray. This is required by some advanced lighting algorithms (such as e.g. [Sbert97]). Except for these special algorithms, however, this special problem is not very common, and few ray tracers have special optimizations for this task (though they obviously exist, see e.g. [Havran01]).

2.1.1 Scene Description

The “scene” to be rendered consists of a list of “geometric primitives”, which are usually simple geometric shapes such as as polygons, spheres, cones, etc. However, ray tracing primitives may also be as complex objects as parametric patches (e.g. Bezier- or NURBS patches), subdivision surfaces, ISO-surfaces, as well as algebraic or implicit surfaces. Ray tracing can even accurately handle such complex constructions as CSG[2], fractals, and recursively and procedurally defined objects. In fact, *any* kind of object can be used as a ray

[2] CSG = Constructive Solid Geometry

tracing primitive as long as it is possible to compute an intersection between a ray and the primitive.

For many common shapes, there actually exist a variety of different analytic, geometric, or numerical intersection algorithms. All these algorithms have different properties with respect to speed, elegance, precision, or numerical robustness, which makes it hard to always identify "the best" algorithm for any primitive. For an excellent overview of different ray-primitive intersection algorithms, see e.g. Glassner's "Introduction to Ray Tracing" [Glassner89].

Triangles vs. General Primitives: Being able to support almost arbitrary primitives is often considered one of the most important features of ray tracing, as even complex primitives can be ray traced directly without the necessity of tesselating it into triangles. Thus they can be handled at full accuracy and without discretization artifacts. Additionally, it may be much more efficient to not require tesselation of the objects. For example, an intersection between a ray and a sphere can be determined quite efficiently, whereas tesselating the sphere would require to store hundreds of triangles for a reasonable accuracy.

On the other hand, supporting only triangles makes it easier to write, maintain, and optimize the ray tracer, and thus greatly simplifies both design and optimized implementation. Furthermore, practically important scenes (i.e. as used in the VR, CAD or movie industry) usually contain few "perfect spheres", nor other high-level primitives[3], as most programs today are exclusively based on triangular meshes, anyway.

Finally, supporting only triangles does not severely limit the kinds of scenes to be rendered, as usually all of these high-level primitives can be well approximated by triangles.

2.1.2 Ray/Scene Intersection

Finding the closest object hit by a ray requires to intersect the ray with the primitives that make up the scene. Obviously, the naive way of simply intersecting the ray with each geometric primitive is too expensive except for trivial cases. Therefore, accelerating this process usually involves "traversing" some form of an "acceleration structure" – a spatial data structure used to more quickly find objects nearby the ray[4].

[3] An exception are parametric surfaces such as trimmed NURBS surfaces, which are common in CAD industry.

[4] Note that there are also some forms of acceleration structures that are not purely spatial data structures (e.g. Ray Classification [Arvo87]). However, the most common

2.2 The Ray Tracing Rendering Algorithm

Whereas the concept of finding the intersection between a ray and a scene can be used for many different applications (e.g. [Durgin97, Plasi97, Agelet97]), it is most commonly used in the context of the traditional recursive ray tracing algorithm.

2.2.1 Appel: First Approaches to Ray Tracing

The idea of using ray shooting for computing images is as old as 1968, and was first introduced by Arthur Appel [Appel68] as an alternative method for solving the "hidden surface" problem for rendering solid objects. In order to compute a two-dimensional image of a three-dimensional scene, rays are generated from the virtual camera through each pixel, traced into the scene, and the closest object is determined (see Figure 2.1). The color of this ray is then determined based on the properties of this object.

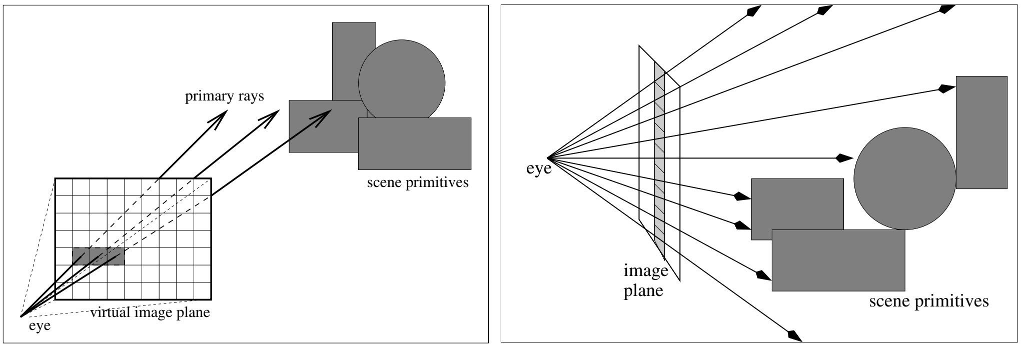


Figure 2.1: The principle of ray casting: In order to compute an image of a scene as seen by a virtual camera, a "primary ray" is shot through each pixel of the virtual image plane, and cast into the scene (left). For each such ray, the closest object hit by this ray is determined by intersecting the ray with the geometric primitives that make up the scene (right).

Additionally, the same concept of shooting rays for determining visibility can be used to shade the hit-point. For example, shadows can be computed by shooting "shadow rays"[5] into the direction of the light sources, to determine whether the respective light source is occluded from the hit surface point, or whether it is occluded. This concept of using shadow rays to influence the

techniques usually are spatial techniques.

[5] Other commonly used names for shadow rays are "illumination rays", "shadow feelers", or "light feelers".

shading of the primary rays hit point was already hinted at by Appel, though he used it in a slightly different form than commonly used today. However, Appel did not yet consider secondary effects like reflections and refraction, and only actually traced eye rays and shadow rays.

2.2.2 Whitted: Recursive Ray Tracing

Today, however, ray tracing is usually used in a recursive manner. In order to compute the color of the "primary" rays (i.e. the first generation of rays that is originating directly at the camera), the recursive ray tracing algorithm casts additional, "secondary" rays to account for indirect effects like shadows, reflection, or refraction[6].

This concept of recursive ray tracing can best be explained by the sketch in Figure 2.2: In this example, the scene to be rendered consists of some geometric primitives (forming a room with some furniture and a glass object on the table), two point light sources, as well as a description of the virtual "camera". In order to compute the color of a pixel on the image plane of the virtual camera, a ray is cast from this camera (through the respective pixel) into the scene, and into the scene. After determining the first hit point of this primary ray, the ray tracer computes the color being reflected into the direction of that ray by first computing the incident illumination at the hit point.

In order to correctly support shadows, light sources only contribute to the incident illumination if the hit point is not occluded from the position of the respective light source, which is checked by tracing a shadow ray towards the direction of the light source.

Additional to direct illumination from light sources, illumination from arbitrary other directions (e.g. from the reflection and refraction directions for specular effects) can be considered by casting a ("secondary") ray into the respective direction and recursively evaluating the light being transported along this rays. This recursive evaluation then proceeds in exactly the same way as for the primary ray. Of course, these secondary rays can in turn trigger another recursion level of new rays, etc.

Using recursive ray tracing, it is easily possible to compute secondary effects like reflection or refraction. Usually all that is required to compute these effects is to determine the direction of the secondary ray (which is

[6] In the context of a rendering algorithm, the term ray *tracing* usually refers to recursive ray tracing. The special case of *not* shooting any secondary rays is typically called ray *casting*. Ray casting is most commonly used in volume rendering, but may also be beneficial for polygonal rendering or other visualization tasks, i.e. for interactively rendering ISO-surfaces, or for rendering massively complex models.

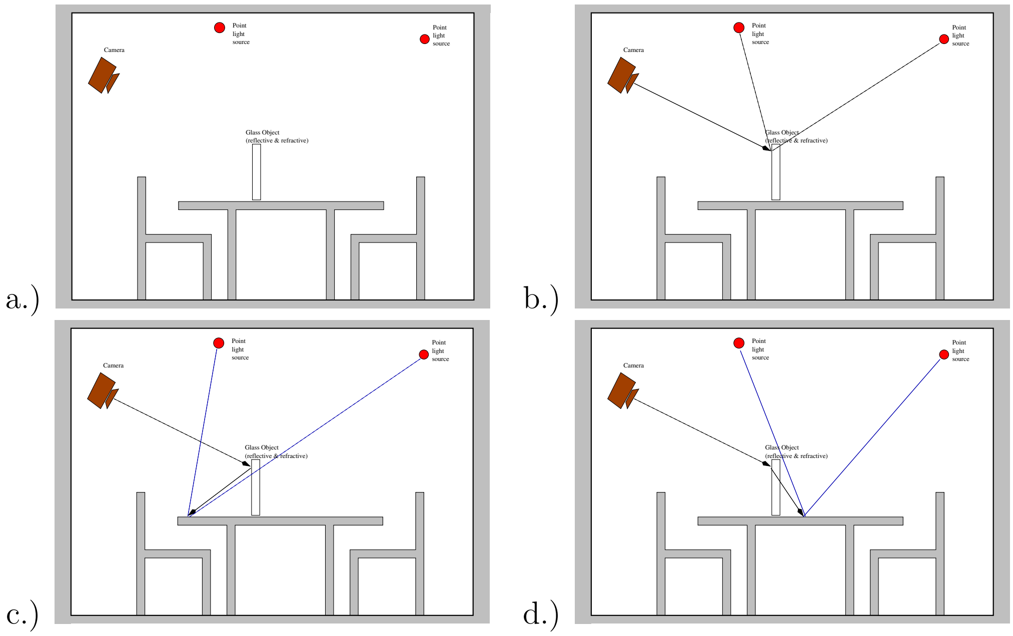


Figure 2.2: Recursive (whitted-style) ray tracing in a simple scene consisting of a camera, some diffuse geometry, a partially specular glass object, and two point light sources (a): A primary ray from the camera hits an object. While "shading" that ray, shadow rays are sent towards the two light source to determine their visibility (b). In order to compute reflection off the glass, a secondary ray is recursively traced into the reflection direction to compute the amount of incoming light. To do this, it can cast new shadow rays, or – if necessary – additional reflection or refraction rays (c). After having finished the reflection ray, the glass shader also computes the refraction direction, and recursively casts a refraction ray (d).

usually goverened by well-known physical formulae), recursively tracing this ray, and taking into account how much of this incoming light is actually reflected into the direction of the incoming ray. By successively performing these operations for each pixel in the virtual image plane, eventually the color of all the pixels are computed, and the rendered image can be displayed.

In this recursive form, ray tracing has first been used by Turner Whitted [Whitted80] in 1980. Therefore, it is often termed "whitted-style" ray tracing, or "recursive" ray tracing[7]. It is this recursive application of ray tracing that is most commonly used, and what most people associate with the term "ray tracing".

[7] Other popular names for whitted-style ray tracing are "classical" ray tracing, or "full-featured ray tracing".

2.2.3 Cook (1984): Distribution Ray Tracing

Whereas whitted-style ray tracing already allowed for computing reflections, refraction, shadows and glossy highlights, it was originally limited to perfectly specular reflections and refraction, point light sources, and instantaneous shutters.

These restrictions have lateron been removed by Cook [Cook84a], who extended the range of ray tracing effects to also include more realistic effects like glossy reflections, smooth shadows from area light sources, motion blur, depth of field, and glossy reflections. This was achieved by modeling all these effects with a probability distribution, which allowed for computing them via stochastic sampling.

Glossy reflections for instance can then be computed by stochastically sampling this probability distribution and recursively shooting rays into the sampled directions[8]. However, in order to achieve a sufficient quality, distribution ray tracing requires to generate a relatively large number of samples, and is usually quite costly.

2.2.4 Programmable Shading

Apart from the ability to accurately compute shadows and reflections, one of the most important aspects of ray tracing is "programmable shading", in the sense that the color of a ray can be computed in an arbitrary way, including the ability to shoot secondary rays.

Whereas Appel, Whitted, and Cook still used simple, fixed lighting models, many researchers have soon noted that ray tracing offers the option to easily change the appearance of an object by modifying the way that a ray is shaded. Over time, a huge toolbox of shading effects have been developed, including different lighting models like e.g. Blinn, Phong, Cook-Torrance, Torrance-Sparrow, Ward, Ashikmin, Lafortune, etc., texture mapping, bump mapping, different camera models, displacement mapping, procedural shading, and many others (for an overview of these techniques, see [Foley97, Akenine-Möller02, Glassner94, Ebert02]).

However, the real breakthrough for programmable shading came with the introduction of programmable "shading languages" like RenderMan [Pixar89, Apodaca90, Hanrahan90, Upstill90, Apodaka00]. Such shading languages

[8] Note that Cooks "distribu*tion* ray tracing" approach was originally called "distribu*ted* ray tracing". As "distributed" ray tracing also refers to a special term in parallel ray tracing (i.e. ray tracing on non-shared memory machines), the name for Cooks concept of considering probability distribution functions has lateron be changed to distribu*tion* ray tracing.

clearly separated the shading process from the actual ray tracing, and allowed to describe the shading process using a convenient, easy-to-use high-level language. Shaders could be written independently of the application and independent of other shaders. The effects of different shaders could be easily combined in a plug-and-play manner.

2.2.4.1 The Shader Concept

Originally, ray tracers usually supported only one kind of "shader" that was attached to all objects at the same time. This function for computing the "color" of a ray typically consisted of a physically motivated lighting/material model that was the same for all primitives, and which could be parameterized by different parameters like material properties or textures.

Today however ray tracers usually follow a much more general concept in which each respective primitive may have a different function for "shading" the ray (i.e. for computing its color). This function can be completely independent of all other surfaces, and does not have to follow any physical properties at all. All that has to be done to implement this concept is to require that each object has one shader, and that this shader alone is responsible for computing the color of each ray hitting that object. Using this concept allows for a "plug and play" mechanism in which different shaders can cooperate in rendering an image, without any shader having to know anything out the other ones.

For example, a scene might contain a sphere with a specular metal shader, as well as another object with a shader simulating wood with procedural methods. In order to shade a ray hitting the metal sphere, the metal shader simply casts a ray into the reflection direction, and recursively calls the shader of the respective hitpoint to compute that reflected ray's color. This way, the wooden object can reflect off the metal sphere without the metal having to know anything about a "wood" shader at all.

This kind of abstraction can also be applied to light sources, the camera, or the environment. Each of these concepts is described by a separate kind of shader, resulting in camera, surface, light, environment, volume, and pixel shaders (also see Figure 2.3).

Camera Shaders: Camera shaders are responsible for generating and casting the primary rays. Typically ray tracers use a perspective pinhole camera (see e.g. [Glassner89]), but other effects like e.g. fish-eye lenses or realistic lens systems are easy to implement, too. More importantly, it is possible to also compute advanced effects like motion blur and depth of field by using camera shaders that simulate real cameras with finite lens aperture and

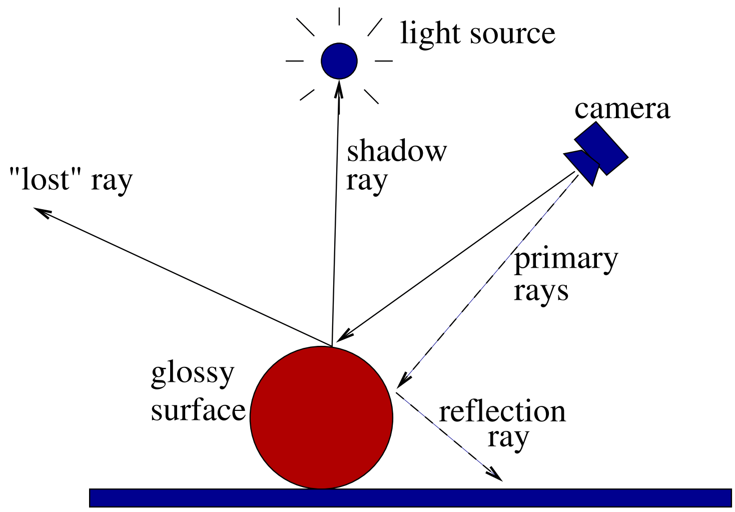


Figure 2.3: A typical ray tracing shader framework with camera, surface, environment, and light shaders: The camera shader generates the primary rays to be cast into the scene. At the points where a ray hits a surface, the respective primitive's surface shader computes the color of this ray, potentially shooting new rays (which in turn get shaded by surface shaders if they hit anything). In order to query the incident illumination from a light source, the surface shader can call back to the respective light's light shader to compute this value, and can then cast a shadow ray to determine visibility. Rays that get "lost" into the environment (i.e. which do not hit any primitive) get shaded by the environment shader.

shutter times [Cook84a, Glassner89, Kolb95].

Surface Shaders: Once a ray hits a geometric primitive, the respective "surface shader" of that primitive is responsible for computing the "color" of this ray. In order to do this, the surface shader first computes the incident light onto the surface by calling the light shaders to supply it with information on the incoming light (so-called "light samples", see below). The surface shader can then cast shadow rays to account for occlusion and computes the light that is reflected into the direction of the incoming ray. Additionally, the surface shader may cast new secondary rays to account for light coming in from other directions, e.g. via reflection or refraction.

Light Shaders: In order not to restrict the ray tracer in what kind of light sources it supports, a flexible shader concept allows for "light shaders". Given a surface hit point, a light shader generates a "light sample" for this surface point, i.e. it returns a vector pointing towards this light source, the distance to this light source, and the "intensity" that this light sample contributes to the surface sample[9].

Environment Shaders: As the scene is often not "closed", it may easily happen that rays are cast into directions in which there is no geometry that they can hit. Such rays get "lost" into the environment. In order to compute their contribution, they can be passed to an "environment shader" which typically looks up a value from a texture representing the distant environment. However, environment shaders can also be used to simulate more complex effects like skylight models.

Volume Shaders, Pixel Shaders, etc.: Obviously, it is possible to extend the just mentioned shader concept even further. For example, it is often common to use "volume shaders" to compute the attenuation that a ray is subject to when traveling between two surfaces. As the volumetric effects are beyond the scope of current interactive ray tracing systems, we will not go into details here. Some ray tracers also support "pixel shaders" or "image shaders" that perform some post-filtering (e.g. tone mapping) on the image.

This "shader concept" allows for a high degree of flexibility and extendibility. All the above mentioned shaders – except for volume shaders – are also supported in the RTRT/OpenRT system (see Part II). For some examples of what is possible with this concept, see Chapter 11.

2.3 General Ray Tracing based Algorithms

Though recursive ray tracing is undoubtedly the most common form of using ray shooting for generating images, there is also a wider range of "ray tracing algorithms" that do not fit the category of recursive ray tracing. For example, many global illumination algorithms only use ray tracing for visibility computations (like radiosity), or also start rays/paths at the light sources (like light path tracing, bidirectional path tracing, photon mapping, and metropolis light transport). Ray tracing is even being used for several applications outside the graphics domain. For example, it can be used for

[9] Note that "intensity" is actually the wrong term. In a physically correct renderer, the value returned by a light shader should actually be "radiance". Still, "color" and "intensity" of the light sample are more commonly used terms in practice. This may stem from the fact that in a general ray tracer light sources do not always have physical meanings (i.e. lights with negative power, or with constant distance falloff), for which the term "radiance" does not make sense.

planning the optimized placement of antennas for wireless communication (e.g. [Agelet97, Dandekar99, Hoppe99, Durgin97, McKown91]), or in computing radar cross sections [Chase00].

Most of these algorithms only use the basic concept of ray tracing, i.e. fast traversal and intersection, but otherwise have few in common with recursive ray tracing. Even most global illumination algorithms – which are often considered a mere "extension" to the ray tracing algorithm – are fundamentally different from ray tracing. For example, the above-mentioned shader concept is typically not flexible enough for most global illumination algorithms, nor do all of these algorithms parallelize as easily as recursive ray tracing. Similarly, the types, order, distribution, and coherency of the rays shot in such algorithms can be drastically different from standard recursive ray tracing. Because of this, it is not yet clear to what degree these kinds of algorithms would benefit from a realtime ray tracing engine, or how well they could be implemented on possible future hardware architectures such as the SaarCOR hardware. However, even if these algorithms can thus not benefit from all advances in realtime ray tracing, it is still likely that they will still benefit from better algorithms and improved implementations of the core ray tracing concepts.

Chapter 3

A Brief Survey of Ray Tracing Acceleration Methods

As discussed previously, the concept of ray shooting has many applications in computer graphics, be it simple ray casting, recursive ray tracing, full ray-tracing based global illumination, or just a toolbox for more advanced algorithms. While most of these applications are quite different from each other, they all have one thing in common: They usually require an enormous amount of rays to be shot (usually in the millions) and thus spend a large fraction of their run time on ray shooting.

Thus, finding ways of accelerating ray tracing has always been of major importance since the very invention of ray tracing[1]. Consequently, a vast amount of different techniques for accelerating ray tracing (and its different variants) have been proposed over the last two decades.

Though the RTRT/OpenRT system uses only very few of all these different concepts[2], any introduction on ray tracing would surely be incomplete without at least summarizing the most often used concepts. Because of the vast amount of research that has already been performed on this topic, any such summary will be incomplete. As such, I will intentionaly only cover the most basic concepts and publications[3]. For a more complete summary and discussion of ray tracing acceleration techniques, I refer to the respective sections in the various books on Ray Tracing and Com-

[1] Already in 1980[Whitted80], Whitted noted that 95% of compute time was spent on intersection calculations.

[2] For example, we make massive use of hierarchical data structures, but few use of adaptive sampling, shadow caches, first hit optimizations, etc.

[3] Furthermore, most techniques have been proposed independently or similarly by many different researchers, or have been proposed in many different variants. Because of brevity of space, I will typically include only the (to my esteem) most common reference.

puter Graphics (like e.g. Glassner [Glassner89], Shirley [Shirley02, Shirley03], Möller-Haines [Akenine-Möller02] or introductory graphics [Foley97]), or to the original publications[4].

Outline

All the different techniques that have been proposed over the last two decades can be structured and ordered in many different ways. Most commonly, this ordering is performed on the kind of "coherence" these different techniques employ[5]. However, though exploiting coherence undoubtedly most often is the key to fast and efficient ray tracing, the term coherence is used inconsistently by many different researchers, and many acceleration techniques actually use a mix of different kinds of coherence. As a consequence, the following summary of techniques will be ordered based on the goals that they try to achieve.

Based on that, the different techniques that have been proposed over the last two decades can be roughly grouped into two categories: One category consists of techniques that aim at reducing the number of rays to be traced, which can be performed either by (re-)constructing an image with less samples on the image plane (e.g. through adaptive sampling), or by reducing the number of rays to be traced for each such sample (e.g. by shooting less shadow rays, or through pruning of the shading tree[6]).

Please note that many of these techniques are applicable to only a small subset of all the variants of ray tracing. For example, shadow caching (see below) can not be used to accelerate ray casting, and first hit optimizations will hardly benefit global illumination algorithms.

[4] Very extensive surveys of ray tracing articles (though usually not including the newer ones) can also be found in [Wilson93]

[5] Coherence refers the degree of similarity between two problems. Obviously, two similar problems can be solved faster if this similarity can be exploited in one or another way. Ray tracing contains many different forms of coherence that can be exploited, like e.g. ray coherence, temporal coherence, image space coherence, object coherence, ray coherence, memory access coherence, frame-to-frame coherence, etc.

[6] Note that the term "shading tree" (or shade tree) also has another, totally different meaning: The term "shade tree" is (as in this context) commonly used to refer to the tree formed by all secondary rays that have recursively been invoked by a given ray. This may *not* be confused with Cook's "shade trees" (see [Cook84b]), which essentially form a way of elegantly expressing the way that a certain (single) ray is shaded, similar to a shading language. Though the term "ray tree" might be a better name for the former concept, the ambiguous term "shade tree" is already widely used in practive.

3.1 Computing less Samples in the Image Plane

Obviously, the rendering time of a frame is closely related to the number of pixels that have to be computed for this frame. As such, one of the most obvious approaches to reducing the compute time for a frame is to reduce the number of primary rays shot for each frame.

3.1.1 Pixel-Selected Ray Tracing and Adaptive Sampling

One of the most (in)famous methods for doing this is to sample the image plane adaptively. Instead of tracing a ray through each pixel, the image plane is subsampled at a fixed spacing. Depending on some heuristics (e.g. if the contrast and depth difference between two neighbouring pixels is small enough), the color of the pixels in-between is either interpolated from the four corner pixels, or the sampling density in this image location is adaptively increased. This process is repeated recursively until either interpolation can be performed, or until all pixels have been traced. For a closer description of this method, see e.g. [Glassner89]. Obviously, it is also possible to trace at least one ray per pixel, and use the described method only for adaptive super-sampling.

However, adaptive methods work best if the features in a scene are quite large. For highly detailed geometry and high-frequency features (such as textures), it is very likely that some of the fine features are missed, and get lost due to the interpolation. Thus, except for trivial scenes these methods break down very quickly, and lead to disturbing artifacts, especially in animations.

3.1.2 Vertex Tracing

Another method for reducing the number of primary samples is "Vertex Tracing" [Ullmann01a, Ullmann01b, Ullman03]: Instead of tracing primary rays through each individual pixel, vertex tracing targets primary rays directly towards the vertices of visible triangles, computes the color of these vertices by recursive ray tracing, and uses graphics hardware to perform the interpolation between the vertices. In order to avoid excessive interpolation, vertex tracing uses several heuristics that adaptively subdivide the triangles (with new rays being shot to the newly created vertices) if certain criteria are met. Additionally, rays are only shot towards objects that are explicitly marked as "ray tracing objects", all other objects are rendered with rasterization hardware.

If the objects to be ray traced are rather simple (i.e. if they have few visible vertices) vertex tracing can greatly reduce the number of rays that have to be traced, and can achieve interactive update rates even on a single desktop PC. The number of rays shot in vertex tracing also depends to a large degree on how often triangles are subdivided. Obviously, coarser subdivision is much faster, but is also likely to miss certain features, and thus to generate disturbing artifacts. In order to increase interactivity, vertex tracing allows for interactively changing the quality parameters: During user interaction, reduced quality parameters are used for fast user feedback, while the image is progressively improved as soon as interaction stops.

However, using reduced quality settings can result in rather poor rendering quality due to excessive interpolation and under-sampling of features. This becomes especially pronounced if the image to be rendered contains many detailed features.

3.1.3 The Render Cache

Another combination of interpolation and sparse sampling of the image plane is Walter et al.'s "Render Cache" approach [Walter99, Walter02]: The render cache keeps a cache of "old" image samples from previous frames, and reprojects those to each new frame. These reprojected samples provide a sparse sampling of the image plane, our of which the full image is then reconstructed using certain heuristics to resolve occlusion, disocclusion, and interpolation inbetween samples.

Asynchronously to this reprojection process, new samples are generated by continuously tracing new rays. To improve performance, the render cache uses heuristics to decide which regions of the image plane need to be sampled most, and preferably generates new samples in those regions. New samples are inserted into the render cache data structure, thereby evicting older samples.

The render cache allows for decoupling frame rate from sample generation speed, and therefore allows interactive frame rates even for very slow renderers. However, the reprojection and image reconstruction steps are quite costly, and only really pay off for extremely costly renderers. As such, the render cache is most beneficial for extremely complex computations, such as global illumination. For simple ray tracing, however, it is often faster to simply ray trace the whole image.

However, the worst problem of the render cache is its limited rendering quality: Even though the render cache uses several heuristics to resolve the worst artifacts, not all artifacts can be avoided: While the render cache quickly converges to a high-quality image when no user interaction occurs, under-sampling of certain objects or image regions, use of outdated samples, and excessive blurring (especially across under-sampled discontinuities) are frequently visible during user interaction. These artifacts become especially apparent for highly detailed scenes with a high degree of user interaction, e.g. with fast camera movement and many moving objects.

3.1.4 Edge-and-Point-Image (EPI)

An extension to the render cache has already been proposed in the form of the EPI ("Edge and Point Image") approach by Bala et al. [Bala03]. Instead of reconstructing the image only based on the sparse pixel samples, the EPI approach also tracks discontinuities such as object silhouettes and shadow borders, and uses those to improve the reconstruction step. Compared to the render cache, the EPI approach avoids interpolation and blurring over these discontinuities, thereby greatly reducing reconstruction artifacts.

Sadly, however, the EPI is not completely orthogonal to a ray tracer. For example, it requires projection of scene features (e.g. object silhouettes) onto others to determine discontinuities, and thus does not follow a "point sampling" approach. As such, it can not easily be layered "on top" of an existing interactive ray tracing system to further increase the performance[7].

3.2 Reducing the Number of Secondary Rays

All the previously discussed techniques have aimed at improving the rendering time by reducing the number of primary rays traced per image. Apart from the number of primary rays per frame, the most obvious factor governing the total rendering time is the average number of rays per pixel, as the product of these two numbers determines the total number of rays shot during each frame.

In practice, this number of secondary rays per pixel can be quite large, especially for scenes with many light sources and with a large amount of specular objects. The number of light sources often has a near-linear impact on rendering time, as each ray cast into the scene (primary rays as well as secondary rays) requires to shoot shadow rays to each light source. Furthermore, highly specular scenes require to shoot many secondary rays per pixel for computing reflection and refraction effects. Especially objects that require to compute *both* reflection and refraction rays usually lead to an excessive number of rays per pixel due to an exponential growth of the ray tree with the number of recursion levels being considered.

[7] Though a fast ray tracer could certainly be used to generate the EPI samples at a faster rate.

As such, practical scenes (which usually contain both many light sources as well as specular objects) require a large number of rays per pixel. Reducing this number can then lead to significant improvements in rendering time.

3.2.1 Shadow Caching

Typically, the majority of all rays shot in a ray tracer are shadow rays, as each primary or secondary ray usually triggers many shadow rays. For shadow rays, however, it is sufficient to find *any* occluding object to guarantee occlusion. Furthermore, many shadow rays are very similar (e.g. all shadow rays to the same point light source), and are often occluded by the same object. “Shadow caching” [Haines86] exploits this coherence by storing at each light source the last occluder. Each new shadow ray to this light source is then first intersected with this cached occluder. If any of these cached object yields a positive intersection, occlusion of the shadow ray is guaranteed, thus the shadow ray does not have to be traced any more. For many light sources with a high degree of occlusion, shadow caching can greatly reduce the number of shadow rays. Furthermore, shadow caching does not require any approximations, does not generate any artifacts, and is orthogonal to most other techniques (i.e. it can easily be combined with other techniques).

However, shadow caching also has some drawbacks: For example, it quickly breaks down if the primary and shadow rays get incoherent, as this leads to “thrashing” of the shadow caches. This is especially true for area light sources and global illumination algorithms. Even worse, by design shadow caches can only accelerate shadow rays that are actually occluded. For non-occluded shadow rays, there is even some overhead due to the cost for intersecting with the cached occluder. Finally, shadow caching often breaks down in complex scenes, as small triangles are less likely to occlude several successive shadow rays. Due to these reasons, shadow caching is generally *not* used in the RTRT/OpenRT engine.

3.2.2 Local Illumination Environments

Recently, a more sophisticated optimization for shadow rays has been proposed in the form of “Local Illumination Environments” [Fernandez02] (also dubbed “LIEs”): LIEs subdivide the scene into a set of “voxels”, each of which stores information on how the different light sources influence that respective region of space. Each voxel stores a list of light sources that will be fully occluded, fully visible, or partially visible with respect to that voxel. With this information, the number of shadow rays to be show can be significantly reduced: Light sources that are fully occluded from the respective voxel do not have to be considered at all, and fully visible light sources can be considered without having to trace any shadow rays. Only partially occluded light sources require shooting of shadow rays. Once the LIE data structure is available, all that has to be done in order to shade a hit point is to look up the respective voxel for the hit point, and use the information stored within that voxel as just described.

However, building the LIE data structure can be quite involved. To be accurate, it would require to solve the visibility problem in object space, which can be quite complicated. Using shadow rays it can however be well approximated. To avoid the preprocessing time and memory consumption for builing the whole data structure in advance, the LIE data structure can be built “on demand”. Furthermore, the data structure can be built hierarchically: A light source that is fully occluded in a certain voxel will also be fully occluded in any of its subvoxels.

Using local illumination environments, only a small fraction of all shadow rays have to be traced. Especially for scenes with many light sources, this can lead to a significant reduction in rendering time.

3.2.3 Adaptive Shadow Testing

Another form of reducing the number of shadow rays has been proposed by Ward [Ward91]: In “adaptive shadow testing”, all light samples are sorted by their contribution to the surface point, and only the most important ones are actually traced. The occlusion of light samples with a small contribution is estimated based on other samples, and tracing these shadow rays is avoided.

3.2.4 Single Sample Soft Shadows

In order to reduce the number of shadow rays for computing smooth shadows, Parker [Parker98b] has proposed a method that approximates smooth shadows with a single shadow ray. The method shrinks the light source to a point, and traces a single shadow ray to this point light, which is then attenuated depending on on how narrowly it misses potential occluders. While the method produces only approximate results, the generated shadows look very convincing.

3.2.5 Pruning the Ray Tree

As ray tracing is well known for its ability to compute reflections and refraction, it is often used in highly reflective scenes. This can easily result in an excessive number of rays to be shot in each frame, due to so-called "explosion of the ray tree": If each surface shader on average shoots more than one secondary ray (e.g. one for reflection and one for refraction), the number of secondary rays grows exponentially with the recursion depth.

However, secondary rays are usually weighed with reflection and refraction coefficients (usually depending on surface properties and incoming ray direction), which may get rather small. Furthermore, these attenuation factors multiply with each further surface interaction, resulting in most rays having a rather small pixel contribution. In order to avoid tracing all these "unimportant" rays, it is common to "prune" the ray tree by tracking each rays "pixel contribution", and to terminate the recursion once this contribution drops below a certain threshold.

However, simply terminating such rays without compensation results in a loss of illumination, and in "biased" images[8] Pruning the ray tree can also be made unbiased by terminating the rays probabilistically with correct reweighting (so-called russian roulette termination [Arvo90b]). This however usually leads to noise in the image, which might be even more disturbing to the viewer.

3.2.6 Reflection Mapping and Shadow Maps

Finally, the number of both secondary and shadow rays can be greatly reduced by approximating shadows and reflections with reflection maps and shadow maps. Both reflection maps and shadow maps are well-known concepts in realtime rendering (see e.g. [Akenine-Möller02]). Though they are typically used in triangle rasterization to approximate effects that can not be computed at all, they can also be used in ray tracing to approximate effects rather than shooting rays for computing them.

However, both methods are infamous for their tendency to generate artifacts. As such, they somewhat contradict the concepts of ray tracing, and should be used with extreme care.

3.2.7 Sample Reuse for Animations

In the special case of ray tracing animations, it is also possible to reuse samples from other frames without re-tracing the respective rays. For example, Formella et al. [Formella94] has proposed to store the complete "ray tree" of each pixel, and to only re-trace those parts of a ray tree that are intersected

[8] Pruning by pixel contribution becomes especially problematic in scenes with strongly varying light source intensity: If connected to a bright light source, even a ray with small attenuation factors may still have a stronger impact than a less attenuated ray illuminated by a dark source.

by a moving object. If a pixel is not affected by a moving object, it can simply be reused in other frames without tracing any new rays. Similar techniques have also been used by others. However, these techniques only apply to ray tracing animations, and are often complicated to use in practice.

3.2.8 Sample Reuse with Interpolation

Another way of reusing samples is to store previously computed samples, and to interpolate new samples from old ones instead of tracing new ones.

For primary rays, this approach has first been proposed by Ward in his "Holodeck" system [Larson98, Ward99]: Rays from previous frames have been stored in a spatial data structure, and new primary rays have been interpolated between those stored samples. New samples are generated asynchronously all the time, thereby improving the image quality when looking at an object for an extended period of time.

The approach has been generalized by Bala et al. [Bala99]. Instead of restricting the approach to primary rays – which are likely to create the most visible artifacts when interpolated – Bala extended the approach to also interpolate between secondary rays. This allowed to restrict the interpolation to regions where it is least likely to result in artifacts. While this still allows for a notable reduction in rays to be shot, it still generates high image quality.

Furthermore, Bala's approach allows for tracking the maximum interpolation error. This in turn makes it possible to render an image – using this approximate technique – with strict error bounds, which is a very important property, especially for practical applications.

3.2.9 First Hit Optimization

A much simpler, and more widely used technique for avoiding the shooting of rays is the so-called "first-hit optimization" (also called "vista buffering" or "ID rendering"), which uses rasterization hardware to speed up tracing the primary rays: In this technique, the scene is first rasterized in "ID rendering" mode, i.e. each primitive is rendered with a unique color that enables to identify it. Then, the primitive hit by a primary ray going through a certain pixel can be identified by looking up that pixels color value from the frame buffer.

However, the impact of the method is rather small. First, it only helps in accelerating the primary rays, which usually make up only a rather small fraction of all rays. For real scenes – with secondary effects and several light sources – the impact is rather small. Furthermore, for realistically complex scenes (and a reasonably fast ray tracer) it is often faster to trace the primary rays than first rasterizing the scene.

Summary

From all the just mentioned techniques, shadow caches are the most commonly used, and are still widely used in practice. Also adaptive sampling is used quite often, especially in the "demo community", which aims more towards artistic and highly impressive ray tracing demos than for a general ray tracing engine. However, most of these techniques require special restrictions (e.g. point lights and spot lights only for efficient shadow caching), or easily break down in practice. As such, the RTRT/OpenRT system (see Part II) does not use any of these techniques, and only concentrates on tracing all rays as quickly as possible[9].

3.3 Accelerating Ray-Scene Intersection

Apart from trying to reduce the number of rays to be shot for an image, another obvious method of accelerating ray tracing is to improve the core ray tracing algorithms such that the rays can be traced faster.

3.3.1 Faster Intersection Tests

Usually a large fraction of the compute time in ray tracing is spent in ray-primitive intersections. This was already noted in 1980 by Whitted [Whitted80], who reported 95% of his compute time to be spent on intersection computations.

3.3.1.1 Different Variants of Primitive Intersection Tests

Obviously, the intersection between a ray and any kind of primitive can be computed in multiple ways. Each of these different algorithms has different properties such as the number of floating point operations vs. integer operation vs. conditionals, memory consumption, or numerical accuracy).

[9] While RTRT/OpenRT does not *in general* use these techniques, some of them have still been used in some special applications: For example, shadow caching has been used in the original "Instant Global Illumination" system [Wald02b] (see Chapter 13), and probabilistic pruning of the ray tree has been used for visualizing the "Headlight" model [Benthin02] (see Chapter 11.2)

Different applications favor different of these properties, and varying availability of certain hardware features (e.g. a fast floating point unit) further shifts the respective advantages and disadvantages.

Consequently, a large number of different algorithms are known for most kinds of primitives. For triangles alone, many different algorithms exist (e.g. [Möller97, Badouel92, Erickson97, Shoemake98, Woo90]). Additionally, there exist dozens of variants of different algorithms (see e.g. [Möller, Held97]) also for other kinds of primitives, most of which are considered "common knowledge" and which thus are not even sufficiently documented anywhere. The same essentially is true for other kinds of primitives, as virtually each ray tracing system has its own special implementation for each kind of primitive.

However, RTRT supports only triangles anyway, and uses its own specially designed ray-triangle intersection test (see Section 7.1). As such, we will not go into detail on any of the different intersection tests. Good overviews of the different techniques can be found in most introductory books (see e.g. [Glassner89]).

3.3.1.2 Bounding Volumes

For ray tracers that support complex primitive types (such as e.g. parametric surfaces), many costly ray-primitive objects can be avoided by tightly enclosing these costly primitives with "bounding volumes" (see e.g. [Rubin80, Kay86]). A bounding volume is a simple geometric primitive (usually a box or a sphere) that can be intersected very quickly. If a ray misses the bounding volume (which can be checked quite cheaply), it does not have to be intersected with the complex primitive at all. Only rays hitting the bounding volume have to be checked *also* against the original primitive. Bounding volumes are a standard-technique in ray tracing, but unfortunately pay off only for complex primitive types. Even though RTRT only supports triangles, it uses bounding volumes e.g. for bounding dynamic objects, to avoid having to transform the rays to the coordinate system of the dynamic object.

Note that the use of Bounding Volumes for bounding more complex primitive types may not be confused with Bounding Volume Hierarchies for hierarchical scene subdivision (see below), but is a complementary (though related) technique.

3.3.2 Spatial and Hierarchical Scene Subdivision

Usually the most successful way to accelerating ray tracing is to reduce the number of ray-primitive intersection operations. This is usually achieved by building an index data structures that allows for quickly finding those primitives that are “close” to a given ray, and to skip primitives that are far away. During “traversal” of this acceleration structure, only those potential candidates thus have to be intersected, and the total number of intersection tests can be significantly reduced.

Over the last 20 years, many different kinds of acceleration structures have been developed, like e.g. uniform, non-uniform, recursive and hierarchical grids [Amanatides87, Klimaszewski97, Cohen94, Gigante88, Fujimoto86, Hsiung92, Jevans89], Octrees [Glassner84, Cohen94, Whang95, Samet89], Bounding Volume Hierarchies [Rubin80, Kay86, Haines91a, Smits98], BSP trees (or kd-trees) [Sung92, Subramanian90a, Havran01, Bittner99], and even higher-dimensional, directional techniques such as ray classification [Arvo87, Simiakakis95].

As already noted by Kay [Kay86] in 1986, in principle these techniques mainly differ in whether they hierarchically organize the scene primitives (as done by Bounding Volume Hierarchies), or whether they subdivide object space (or ray space) into a set of unique voxels (as done by BSPs, kd-trees, or Grids).

3.3.2.1 Bounding Volume Hierarchies

In the first class (Bounding Volume Hierarchies) each primitive is stored only once in the hierarchy. This usually leads to a predictable memory consumption, and guarantees that each primitive is intersected only exactly once during traversal of the hierarchy (which is usually not the case for spatial subdivision techniques, see below). On the other hand, different parts of the hierarchy may overlap the same regions in space. This often leads to inefficient traversal such as intersecting the primitives in the wrong order[10] or traversing the same space multiple times).

3.3.2.2 Spatial Subdivision (Grids, Octrees, BSPs, etc.)

In contrast to BVHs, spatial subdivision techniques subdivide three-dimensional space into a finite, non-overlapping set of volume elements (voxels), in which each voxel keeps a list of references to all the primitives overlapping that respective voxel.

Traversing a ray through the spatial acceleration structure then sequentially iterates through all the voxels encountered by a ray, thereby intersecting

[10] The correct order can be guaranteed by organizing the not-yet-traversed parts of the hierarchy in a priority queue and always traversing the closest one. This however incurs additional cost in each traversal step. Also, please keep in mind that the correct order is not important at all for shadow rays.

the primitives referenced by the voxels. If this traversal is performed in front-to-back order, spatial subdivision techniques allow for *early ray termination*: As soon as a valid intersection is found at a certain distance along the ray, all the voxels behind this distance can be immediately skipped without any further computations. On the other hand, spatial subdivision has the drawback that the number of encountered primitives is most reduced if the voxels get very small, which usually requires lots of memory. In that case, however, primitives will usually overlap many different voxels. This not only further increases the memory consumption of these techniques, but often leads to the same primitive being encountered several times during traversal of a ray.

Mailboxing: Fortunately, these multiple intersections of a primitive can be helped by *mailboxing:* In mailboxing [Amanatides87, Glassner89, Kirk91, Wald01a], each ray gets a unique ID assigned to it, and each primitive records the ID of the last ray it was intersected with. During traversal then, multiple intersections can be avoided by simply comparing the current ray ID with the ID of the last intersected ray[11].

On the other hand, mailboxing itself incurs a certain cost (e.g. for deciding whether an intersection has to be performed or not), requires a significant amount of memory (one integer or pointer per primitive), and can easily lead to costly, incoherent memory accesses and cache thrashing. Furthermore, both memory consumption and cache thrashing get worse when using multiple threads, as the mailbox cannot be shared between threads. This makes mailboxing problematic especially for multiprocessor systems and multithreaded hardware architectures. For single-threaded systems with sufficient memory however mailboxing usually still improves performance.

In order to remove memory consumption and improve the caching behavior, *hashed mailboxing* can be used (see [Wald01a]). Though this has shown to be less efficient than "standard" mailboxing in the general case, it is usually preferrable if many threads are to be used and/or if memory is scarce.

3.3.2.3 Algorithmic Variants and Optimizations

As discussed so far, the different acceleration data structured mainly differ conceptually, e.g. spatial subdivion vs. BVHs, or uniform vs. hierarchical subdivision. However, even if the data structure itself is fixed, the performance is still significantly affected by the exact way in which the respective data structure is constructed and traversed.

[11] Please keep in mind that mailboxing is not required at all for Bounding Volume Hierarchies, as these do not lead to multiple intersections with the same primitive.

Different Traversal Algorithms: Though a different traversal algorithm should still traverse exactly the same voxels (and thus intersect the same primitives), the traversal algorithm often consumes a significant fraction of compute time [12]. As such, a different traversal algorithm can significantly affect performance.

Today, there exist several different traversal algorithms for each of the different kind of acceleration structure, each of which have different advantages and disadvantages.

While most of these alternative traversal algorithms do not actually change the underlying data structure, some methods require to slightly modify the data structure itself, e.g. by adding neighbour links [Samet89, Havran98] for faster "horizontal" traversal from one voxel to its neighbour.

Variants of Building the Hierarchy: Apart from the traversal algorithm, the eventual performance of an acceleration structure is significantly affected by the algorithm for constructing the data structure. As the traversal algorithm itself usually does not change either the number nor order of voxels (and thus primitives) encountered by the ray, the number of traversal steps and primitive intersections is mostly determined by the way that the acceleration data structure has been built (i.e. by the number, size, and organization of the voxels).

Even for uniform subdivision (whose structure is determined entirely by the grid resolution, there usually is a tradeoff between reducing the number of primitive intersections and performing too many traversal steps. In practice the optimal grid resolution is hard to determine, and can only be guessed approximately using some heuristics. For hierarchical data structures, finding "the best" organization is even more complex[13].

Though there are several groundbreaking papers on the optimal construction of ray tracing hierarchies (e.g. by Goldsmith and Salmon [Goldsmith87], Subramanian [Subramanian90b, Subramanian90a, Cassen95], or MacDonald and Booth [MacDonald89, MacDonald90]; also see [Havran01] for an overview), the importance of using these techniques is often underestimated in practice. Even the RTRT/OpenRT system at its original publication only considered a naive kd-tree construction algorithm. Havind added a Surface Area Heuristic [Havran01] for building the kd-tree afterwards since then has roughly doubled performance (see Section 7.3).

[12] In practice (i.e. for realistically complex scenes and relatively simple primitives), traversal is usually several times as costly as the ray-primitive intersections.

[13] For hierarchical subdivision, the decision whether to continue subdivion or not, as well as the exact position of the split have to be performed anew in each construction step.

Shadow ray optimizations: As already noted in the previous section, traversal of shadow rays can be accelerated by immediately terminating traversal after *any* intersetion has been found along the ray. This is a simple optimization that can usually be integrated into the traversal algorithm with a single conditional, and as such is present in virtually every ray tracer.

This optimization is especially useful since in most applications of ray tracing most of the rays cast are shadow ray. As such, this optimization is implemented in most avaiable ray tracing systems. Obviously, it is also implemented in RTRT/OpenRT.

Optimizations for finding all intersections along a ray: As already noted before, certain algorithms (e.g. global illumination with “global lines” [Sbert97], or computing transparency along a shadow ray) can benefit from finding *all* intersections along a ray at once. This can significantly improve performance for these tasks, as finding N intersections in one costly traversal is usually much faster than performing N successive traversals from each hit point to the next (even though the ’find all’ traversal is more costly than a single ’find nearest’). This optimization however is only useful for a restricted set of applications. As such, it is not implemented at all in RTRT/OpenRT.

Using special hardware features: Finally, a huge potential for making ray tracers faster lies in exploiting different and more powerful hardware resources. Examples include ray tracing on highly parallel supercomputers [Keates95, Parker99b, Muuss95a], the use of SIMD-extensions present in almost all of today CPUs [Wald01a], ray tracing on current programmable GPUs [Carr02, Purcell02], exploitation of more general programmable devices (e.g. [Mai00, Anido02, Du03]), and eventually the use of dedicated hardware that has been especially designed for ray tracing [Humphreys96, Hall01, Schmittler02]. Fortunately, the use of more powerful hardware platforms often is orthogonal to the “algorithmic” acceleration techniques, and thus can often be combined with such techniques. Even so, special hardware features may be easier to employ with some algorithms than others[14]. As the algorithmic issues of ray tracing are increasingly considered to be mostly solved, such optimizations for specific new hardware features are likely to receive more and more attention.

[14] For example, a SIMD traversal is easier to implement on a kd-tree than on other acceleration structures.

3.4 Summary

In this chapter we have given a brief overview over the different methods and techniques for accelerating ray tracing that have been proposed over the last two decades. As mentioned in the beginning, this list of techniques is not complete, but should contain at least the most famous and most commonly used techniques.

Among all these techniques, hierarchical subdivision methods are the most important, as they allow for reducing the computational complexity (from $O(N)$ to $O(log\ N)$ [Havran01]), whereas all other methods, roughly speaking, only improve "the constants" of the ray tracing algorithm. As such, any non-trivial ray tracing system today uses one or another form of the previously discussed scene subdivision and traversal techniques.

While all of these acceleration methods are commonly agreed to have the same computational complexity of $O(NlogN)$, their respective performance in practive varies from case to case, depending on scene, implementation, hardware platform, and application (i.e. ray distribution). As such, it is not possible to name one single method that is always best, though practice has shown that kd-trees usually perform at least comparable to any other technique [Havran01, Havran00]. Though much of the original ray tracing literature covers Octrees, in practive today the most commonly used techniques are either uniform grids or kd-trees. Grids (especially uniform ones) are advantageous because of their simplicity. Because of this, they are also well suited for implementation on hardware architectures with restricted propgramming model (such as e.g. GPUs)[15]), whereas kd-trees usually adapt better to varying scene complexity, and often achieve superior performance if used correctly.

Except for faster intersection tests and hierarchical subdivision, most of the other previously discussed techniques (such as reducing the number of rays in one or another form) are limited to special cases and to a restricted set of applications. As such they are of limited use for a ray tracing system that is to be used for as many applications as possible, and will thus not receive any further attention in the remainder of this thesis. Even so, many of those are orthogonal to the techniques used by RTRT/OpenRT, and can still be combined with the RTRT/OpenRT system to achieve even higher performance for those applications in which they are effective.

In order to be as wide a class of applications as possible the RTRT core has concentrated mostly on those topics that influence the performance of

[15] Uniform grids do not need to maintain a recursion stack, and can be implemented with few floating point operations

any ray tracer: fast intersection, a good acceleration data structure, fast traversal algorithms, and high-quality algorithms for building the hierarchy. These four topics – with respect to the RTRT realtime ray tracing core – will be discussed in more detail lateron in Part II of this thesis.

Chapter 4
Interactive Ray Tracing

In the form just presented, ray tracing is a well-known technique, and is – in one or another variant – the dominating technique for almost all high-quality offline rendering tasks. For interactive applications however, it is used rarely. In fact, many researchers still believe that ray tracing will never be suitable for realtime applications, or that – if this were to come at all – it is a remote option for the distant future.

However, ray tracing has many advantageous features that would be highly beneficial to have, also – and especially – in the field of realtime rendering. In this chapter, we are going to summarize these advantages of ray tracing, and will argue that ray tracing also – and especially – offers many benefits for interactive rendering.

Having outlined all the advantages of ray tracing over rasterization based approaches, we are going to discuss the reasons why – even though it seems to be the technologically superior technique – ray tracing has not emerged as the standard technique for realtime rendering in the first place. Based on a brief discussion of currently ongoing trends towards larger scenes and more realism, we then argue that these reasons no longer apply today, which makes it likely that ray tracing will play a larger role for realtime graphics in the near future. By shortly summarizing the currently ongoing work in realtime ray tracing, we will finally show that the trend towards this technology is not to happen in a remote future, but that this is already happening today.

4.1 Why Interactive Ray Tracing ?

Interactive rendering today is almost exclusively the domain of rasterization-based algorithms that have been put into highly integrated and optimized graphics chips. Their performance increased dramatically over the last few years and now even exceeds that of (what used to be) graphics supercomputers. In important aspects like floating point performance, memory bandwidth, and chip complexity[1] they have even overtaken state-of-the-art CPUs. At the same time this hardware can be provided at low cost such that most of today's PCs already come equipped with state-of-the-art 3D graphics devices.

Even though the performance increase of 3D graphics hardware has been tremendous over the last few years, it is still far from sufficient for many applications. In particular computer games, visualization applications, and virtual design/virtual prototyping applications have strong demands on rendering speed, image quality, realism, and scene complexity. Furthermore many of these applications demand a large degree of reliability, both on the rendering algorithm itself (e.g. when combining different rendering effects), as well on the quality and correctness of the rendered images.

Current graphics hardware has seen a number of difficulties in these respects because it is quite limited in both image quality and efficiency: In image quality, the achievable effects are usually quite limited, especially if compared to ray tracing. Though many applications (especially games) feature stunning visual effects, these effects are often due to manual design of the scene. In games, for example, it is quite common to manually design textures and scene levels to achieve certain effects like (smooth) shadows, reflections, indirect lighting, etc.

Recently, the affects achievable by graphics devices has been significantly improved by features like multi-texturing, programmable vertex processing, programmable fragment processing, dependent texture lookups, and many more. However, it becomes increasingly difficult for applications to actually take advantage of these features as they are hard to program and even simple shading effects are difficult to express in terms of these extensions. Especially global effects like shadows and reflections – let alone plug-and-play interaction between different shaders – are hard to achieve. While all these extensions are important tools for achieving impressive effects in the above-mentioned manual design process, they are still far from providing the same level of simplicity and automaticity in achieving the same global effects as ray tracing. Finally, this process of manually enhancing the visual appearance of scenes (e.g. through hand-drawn textures) is only feasible for a restricted set of applications. For example, adding realism through manual user intervention is tolerable for computer games that require several months to a few years to be implemented anyway, but it is not an option for the average end user application, in which such effects are desired to be generated

[1] Measured in transistors per chip.

automatically after simply loading a model.

Also in scene complexity, graphics hardware is quite limited, as rasterization exhibits essentially linear cost in the number of polygons. In order to achieve sufficient rendering performance sophisticated preprocessing is required for reducing the number of rendered polygons per frame. This can be accomplished through techniques such as geometric simplification, level-of-detail, occlusion culling, and many others. Again, for game-like applications this often involves a special design and manual “optimizations” of the scene by the game designer.

The problem of limited scene complexity becomes furtherly aggravated by the affect that – even with programmable shading – many advanced effects require multiple rendering passes of the same scene. For example, shadows are still costly to compute interactively for dynamic scenes of reasonable complexity.

Especially in these two domains where graphics hardware becomes problematic – realism and scene complexity – ray tracing is widely accepted as being the “better” algorithm. Ray tracing is famous for its ability to generate high-quality images, and widely accepted as “the” primary tool for realistic image synthesis. Similarly, the fact that ray tracing is logarithmic in scene complexity – and thus very efficient for complex scenes – can be considered common knowledge in computer graphics. In fact, ray tracing actually offers a long list of features that would be beneficial for interactive, high-quality graphics as demanded by many applications. In the following, we will briefly summarize the most important of these advantages:

4.1.1 Flexibility

Ray tracing allows us to trace individual or unstructured groups of rays. This provides for efficient computation of just the required information, e.g. for sampling narrow glossy highlights, for filling holes in image-based rendering, and for importance sampling of illumination [Ward91]. This flexibility of efficiently tracing arbitrary individual rays even allows for computing completely unstructured light paths (see e.g. the Headlight example in Section 11.2). Eventually this flexibility is required if we want to achieve interactive global illumination simulations based on ray tracing.

4.1.2 Occlusion Culling and Logarithmic Complexity

Ray tracing enables efficient rendering of complex scenes through its built in occlusion culling as well as its logarithmic complexity in the number of scene primitives. Using a simple search data structure it can quickly locate the relevant geometry in a scene and stops its front to back processing as soon as visibility has been determined. This approach to *process geometry on demand* stands in strong contrast to the *"send all geometry and discard at the end"* approach taken by current triangle rasterization hardware.

The importance of occlusion culling can also be seen from the fact that even the newest generations of rasterization hardware offer extensions for some degree of occlusion culling (e.g. occlusion queries [NVidia01] and deferred shading [Olano98]). These however are usually quite limited, and not as generally applicable as in ray tracing.

4.1.3 Output-Sensitivity

Ray tracing is "output-sensitive" in the sense that it only computes what is actually needed for the final image. For example, triangle rasterization often requires to process the entire scene (e.g. for computing a shadow map or a reflection map) even though it is not yet clear whether this effect is actually needed at all (i.e. if the object with the reflection map is visible or not). In contrast to that, ray tracing only computes effects that are actually visible.

This output-sensitivity applies to many different aspects: Only those triangles are intersected that are close to a ray, only those parts of a scene are touched that are actually visible, only those reflections or shadows are computed that actually influence a pixel to be shaded, etc. Most importantly, this output-sensitivity applies to each individual pixel. For example, costly shading effects such as multiple relfection and refraction affect the compute time of only those pixels in which they are actually seen, and do not affect the cost of any other pixel.

4.1.4 Efficient Shading

This output-sensitivity also applies to shading. With ray tracing, samples are only shaded after visibility has been determined. Given the trend toward more and more realistic and complex shading, this avoids redundant computations for invisible geometry.

Note that "deferred shading" [Olano98] has also recently been proposed for graphics hardare. In ray tracing, however this concept is much more automatic. For example, deferred shading in ray tracing also affects secondary shading effects (e.g. via reflections). For hardware multipass rendering, this would require the hardware to decide automatically which rendering passes to use.

4.1.5 Simpler Shader Programming

Programming shaders that create special lighting and appearance effects has been at the core of realistic rendering. While writing shaders (e.g. for the RenderMan standard [Apodaka00]) is fairly straightforward, adopting these shaders to be used in the pipeline model of rasterization has been very difficult [Peercy00]. Though todays programmable graphics hardware supports many of the core shading operations (like texturing or procedural computations), expressing the global interaction of different shaders is quite hard to achieve. Since ray tracing is not limited to this pipeline model it can make direct use of shaders [Gritz96, Slusallek95].

4.1.6 Correctness

By default ray tracing computes physically-correct reflections, refractions, and shading[2] . In case the correct results are not required or are too costly to compute, ray tracing can still make use of the same approximations used to generate these effects for rasterization-based approaches, such as reflection or environment maps. This is in contrast to rasterization, where approximations are the only option and it is difficult to even come close to realistic effects.

4.1.7 Parallel Scalability

Ray tracing is known for being "embarrassingly parallel", and thus trivially parallelizable as long as a high enough bandwidth to the scene data is provided. Given the exponential growth of available hardware resources, ray tracing should be better able to utilize it than rasterization, which has been difficult to scale efficiently [Eldridge00].

It is due to this long list of advantages that ray tracing is an interesting alternative even in the field of interactive 3D graphics. The challenge is to improve the speed of ray tracing to the extent that it can compete with – and eventually outperform – rasterization-based algorithms.

4.2 Why not earlier, and why today ?

Having just argued that ray tracing offers so many advantages over triangle rasterization, there are two obvious questions that become apparent: First,

[2] In the sense that the visibility term is handled correctly. This does not mean that all the shading computations performed by the shaders are automatically and necessarily "physically-correct".

why it has not emerged as the leading interactive technique in the first place, and second, why we think that this situation is fundamentally different today.

The first question is rather easy to answer: When interactive computer graphics first emerged, demands on computer graphics have been fundamentally different from today. 20 years ago, scenes have been quite simple – in the order of a few dozen to a few hundred triangles – for which the logarithmic complexity of ray tracing could not pay off, and for which the high initial cost of ray tracing was too high. Similarly, occlusion culling in such simple scenes was not an issue, as overdraw was minimal. Second, shading was quite simple (not even considering per-pixel shading), for which the programmability of ray tracing could not pay off. Furthermore, rasterizing a few big triangle with simple shading allowed rasterization to efficiently exploit coherence by using incremental, cheap integer operations for rastering a triangle. Finally, these incremental integer operations – with minimal demand for on-chip memory – eventually made it possible to dramatically accelerate rasterization by dedicated hardware. For these simplified demands, and with these restricted hardware resources, ray tracing would simply have been the wrong choice.

On the other hand, with this argument in mind it is also clear why these reasons do no longer hold today: Due to increasing demand for more complex shading, triangle rasterization today performs all operations in floating point, too. Similarly, most operations today are performed per pixel (and often on triangles covering at most a few pixels), for which interpolation and incremental operations no longer work. Also memory demands for graphics hardware are quite high today, due to the need to support many textures for faking certain effects. Actually, modern graphics chips use more hardware resources than even complete CPUs. Furthermore, scene complexity increases to million-triangle scenes with significant overdraw (especially for multi-pass algorithms like shadow volumes), for which the logarithmic complexity of occlusion culling features of ray tracing are simply the better choice. Finally, there is increasing demand for advanced features like realistic shadows, reflections, caustics, or global illumination, which get increasingly costly to approximate in a convincing way on graphics hardware.

Obviously, there are still several important applications for which ray tracing does not make any sense at all, or for which it is unclear how to ray trace them at all (like wire-frame rendering, point based rendering, immediate-mode rendering, etc), or for which it is not yet efficient enough (like many games, etc). Even so, with the strong and unbroken trend towards increasingly complex scenes, increasingly complex shading, and increasing demand for advanced features, it seems only a question of time until ray tracing will eventually overtake triangle rasterization. In fact, for many kinds of applications (see some of the applications demonstrated lateron in this thesis), this is already the case today. This is even more surprising given the fact that realtime ray tracing today is still an upcoming, and software-only technology, which in these applications is compared against sophisticated hardware implementations of triangle rasterization.

However, being a de-facto standard for interactive rendering, triangle rasterization is an "entrenched" technology that is hard to challenge even with technological advantages. For example, all commonly accepted APIs, graphics libraries, scene graphs, games, and applications being used in practice today build on triangle rasterization. In order to become a ubiquitously accepted technology for the end user, many of these issues still have to be overcome.

Chapter 5

Towards Realtime Ray Tracing – Related Work

"There are many ways of going forward, but only one way of standing still."
Franklin D. Roosevelt

Today, there are three main different hardware platform on which realtime ray tracing can be realized:

CPUs that run highly optimized and parallelized software implementations of the classical ray tracing algorithm,

Programmable GPUs that can be used as massively parallel, powerful streaming processors, that essentially run a specially designed software ray tracer, and

Special-purpose hardware that is explicitly designed for realtime ray tracing.

Software-based systems essentially run fast implementations of the traditional ray tracing algorithm. However, they have specifically been optimized for speed rather than for quality and flexibility. Additionally, they often use parallel or distributed processing to achieve interactive frame rates. This parallelization can be realized on both shared-memory multiprocessor-machines [Keates95, Muuss95a, Muuss95b, Parker99b, Parker99a, Parker98a], as well as on loosely-coupled clusters of commodity PCs [Wald01a, Wald01c, Wald03a, DeMarle03].

Recently, Purcell et al.[Purcell02] have shown that ray tracing can also be realized on programmable graphics hardware. In his work, Purcell has exploited the programmability of today's GPUs by using the graphics card as a massively parallel, highly efficient streaming processor. Here, the recursive ray tracing algorithm is first reformulated as a stream processing task, by expressing the core algorithms of ray tracing – i.e. traversal, intersection, and shading – as small "kernels" that operate on a stream of pixels and textures, where each pixel corresponds to exactly one ray. The different kernels can then be implemented using pixels shaders, and can be executed by applying the respective pixel shader to the pixels of a screen-aligned quad.

Finally, the third alternative to realizing ray tracing is the design of custom hardware that is specialized for ray tracing. In that approach, the whole ray tracing algorithm is embedded in hardware. Given todays hardware resources, Schmittler et al. [Schmittler02] have recently shown that this approach is indeed feasible. In fact, it apparently can be realized using less hardware resources than used in a modern GPU, and promises to achieve full-screen ray traced images at interactive rates even on a single graphics chip.

This PhD will concentrate mostly on specific details of the Saarland RTRT/OpenRT engine. However, we first want to give an overview and short summary of the different approaches to realtime ray tracing that have been taken so far.

5.1 Realtime Ray Tracing in Software

In order to reach realtime ray traced frame rates with a software system, one has to focus on two different aspects: First, the system has to be built on a highly optimized ray tracing kernel that optimally uses the CPU. This includes *both* using the best available algorithms *and* paying careful attention on implementing them optimally on the given hardware. Second, as even the best algorithms and the fastest CPUs today cannot yet deliver the performance needed for practical applications, a software based system also requires exploiting the tremendous potential for parallel processing in ray tracing. This allows for combining the resources and compute power of multiple CPUs (or PCs) by using parallel or distributed ray tracing.

Especially the latter is common to all software approaches to realtime ray tracing so far. This is both true for shared-memory systems (e.g. by Keates and Hubbold [Keates95], Muuss et al.[Muuss95a, Muuss95b], and Parker et al.[Parker99b, Parker99a, Parker98a]), as well as for current systems based on PCs and PC clusters (e.g. [Wald01a, Wald01c, Wald03a, DeMarle03]).

5.1.1 Interactive Ray Tracing on Shared-Memory Systems

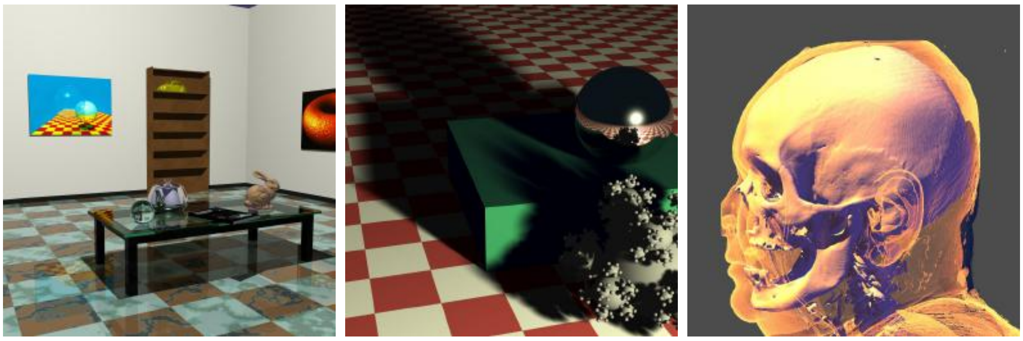


Figure 5.1: Two examples from the Utah Interactive Ray Tracing System. a) A typical ray-traced scenes with parametric patches, shadows and reflections. b) A ray tracing test scene with smooth shadows. c) Complex volume rendering. (Image courtesy Steve Parker)

Though ray tracing itself trivially lends itself to parallelization, special care has to be taken in an interactive setting where only a minimum amount of time can be spent on communication and synchronization. Generally, these issues – fast inter-processor communication and synchronization – can best be handled on shared-memory computers.

Thus, it is not surprising that interactive ray tracing has first been realized on massively parallel shared-memory supercomputers. These systems provided the required floating point power and memory bandwidth, and combined the performance of many CPUs with relatively little programming effort.

The first to achieve interactive frame rates on such platforms were Muuss et al. [Muuss95a, Muuss95b] who used interactive ray tracing to simulate radar systems in highly complex CSG (Constructive Solid Geometry) environments that would otherwise have been impossible to be rendered interactively[1].

On a similar hardware platform, Parker et al. [Parker99b, Parker99a, Parker98a, Parker98b] were the first to show a full-featured ray tracer with shadows, reflections, textures, etc. (see Figures 5.1a and 5.1b). Additionally, their system allows for high-quality volume rendering [Parker99a] and isosurface visualization [Parker98a] (see Figure 5.1c).

[1] At the same time, Keates and Hubbold [Keates95] also presented a massively parallel interactive ray tracer on the KSR-1 machine. The achieved performance however was only close to interactivity, and did not allow for the same practical applications as Muuss's approach.

5.1.2 The Saarland RTRT/OpenRT Engine

Today, a more cost-effective approach to obtain high compute power is the use of a clusters of commodity PCs. Such systems are already widely available and usually cost only a fraction of a shared-memory machine while providing equivalent performance. However, PC clusters do have many drawbacks compared to a shared-memory supercomputer, i.e. they do not offer hardware-supported inter-processor communication, and they have less memory, less communication bandwidth, and much higher latencies.

In 2001, Wald et al. [Wald01a, Wald01c] showed that interactive ray tracing can also be realized on such low-cost hardware. Their system – the Saarland University's RTRT/OpenRT ray tracing engine[2] – combines a fast ray tracing core with sophisticated parallelization on a cluster of commodity PCs. In the meantime, this system has been extended to a complete rendering engine featuring a fast ray tracing core, efficient parallelization, support for dynamic scenes, and a flexible and powerful API. Most of the following discussions on kernel issues, parallelization, dynamic scenes, and API issues will be based on this system.

5.1.3 The Utah "Star-Ray" Architecture

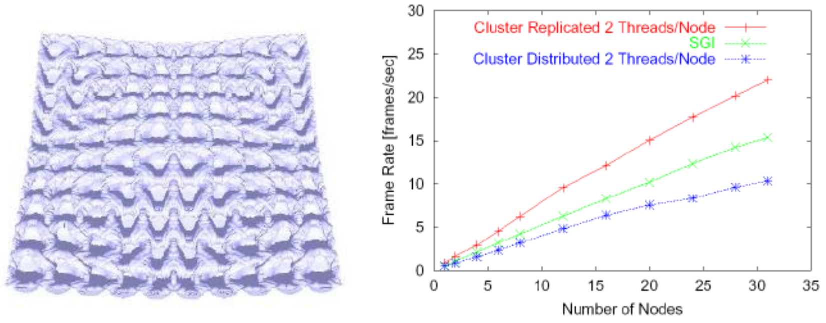


Figure 5.2: Example images from the cluster-based Utah Star-ray system. Left: One frame from an eight gigabyte volume data set rendered interactively with analytic isosurfacing. Right: Performance comparison of their new, cluster based system is comparison to the Onyx. Using the same number of nodes, their cluster based system provides roughly the same performance as the Onyx system. (Image courtesy David E. DeMarle)

[2] Note: "RTRT" refers to the "Real-Time Ray Tracing" core of the engine, while "OpenRT" refers to the API through which this engine is driven (see Section 10)

Just recently, the above-mentioned “Utah-system” [Parker99b, Parker99a, Parker98a] (now called “Star-Ray”) has also been ported to run on a PC clusters [DeMarle03]. It too consists of a sophisticated parallelization framework around a highly optimized ray tracing core. In its core, the new system uses the same algorithms as on the original system on the Onyx [Parker98a]: Highly efficient traversal of the volume data set that quickly skips uninteresting regions, efficient data layout using bricking to improve caching (reported to bring up to a tenfold performance improvement on certain architectures [Shirley02]), optimized algorithms for analytic ray-isosurface intersection computation, and efficient parallelization in the image plane.

While certain of the systems aspects – i.e. the distribution framework and optimization for memory accesses – are similar to the RTRT/OpenRT engine, the system has been optimized mainly for the interactive visualization of volumes and isosurfaces, and does not primarily target polygonal scenes and lighting simulation.

Due to the above-mentioned drawbacks of using PC clusters – less memory, less communication bandwidth, and higher latencies – the parallelization and communication layer of the PC-based Star-Ray system had to be adapted [DeMarle03]. Similar to the Saarland System, they now use a client-server approach, in which the server controls the clients via TCP/IP by sending them image tiles to be computed. Using the same number of nodes, their cluster-based system achieves roughly the same performance as the original, shared memory based system on the Onyx (see Figure 5.2).

The new Star-Ray system is also able to handle massively complex volume data sets by implementing a software layer offering a distributed shared-memory architecture: Volume data is separated into disjoint regions that are kept distributed among the different machines. If a client needs access to remote data, this software layer transparently fetches and caches the required data. Additionally, they perform several optimization to reduce the bandwidth for transferring these tiles. Their system for handling massive volume data is similar to the approach that Wald et al. have taken for rendering massive polygonal data sets [Wald01c], but uses a better, distributed scheme of storing the data. While this distributed data storage costs roughly half the performance of their system, it allows them to render an eight gigabyte dataset (of a Richtmyer-Meshkov instability) at interactive rates with high-quality analytic isosurfaces, as shown in Figure 5.2.

As can be seen, both the Saarland Engine as well as the new Utah engine have concentrated on similar issues: First, a highly optimized kernel that especially considers memory effects. Second, sophisticated parallelization with special emphasis on handling the bandwidth and latency issues of a PC cluster.

While volume rendering obviously requires different algorithms and optimizations than ray tracing in polygonal scenes, many of the concepts are still similar. In the remainder of this thesis, we will concentrate only on polygonal scenes. However, most of the discussed concepts should apply similarly to volume rendering.

5.2 Ray Tracing on Programmable GPUs

Whereas GPUs have originally been designed for triangle rasterization, it soon became popular to "abuse" the capabilities of GPUs for other tasks, e.g. by using multi-pass algorithms for higher-quality shading [Proudfoot01, Mark01]. Due to popular demand – mainly driven by the game industry – graphics hardware has recently evolved to become increasingly programmable, from a reconfigurable rasterization pipeline, over register combiners and vertex shaders, down to todays fragment programs [NVidia01].

Today, fragment programs allow for executing small programs, including multiple accesses to textures – potentially depending on each other – and very fast, SIMD-style arithmetic operations. This programmability is still somewhat limited, e.g. with respect to the number of instructions or texture accesses that can be executed, in the number of kind of write operations, loops, and branches. While it is unlikely that GPUs will ever be as flexible and programmable as CPUs, many of these limitations will probably be reduced.

With this programmability of graphics hardware, it now becomes possible to use the GPU also for other tasks that have previously been the domain of CPU based systems, e.g. for linear algebra operations, matrix solvers, and simulation tasks [Larsen01, Harris02, Krüger03, Bolz03]. Essentially, the GPU is well suited for any kind of algorithm that is highly parallel and compute intensive, which obviously also includes many algorithms in computer graphics. As such, there have already been several approaches for using the GPU for ray tracing [Purcell02, Carr02], and for photon mapping [Purcell03].

5.2.1 Hybrid CPU/GPU Ray Tracing: The Ray Engine

One of the first approaches to using GPUs for ray tracing was taken by Carr et al.'s "Ray Engine" [Carr02], which uses the GPU as a very powerful "coprocessor" for ray intersection computations. Their approach was motivated by the observation that GPUs as extremely powerful at compute-intensive tasks (such as e.g. ray-triangle intersection), but rather weak at algorithms with a complex control flow such as kd-tree traversal, which require branching, looping, and recursion.

With this observation in mind, they have chosen to build a hybrid system that attempts to combine the advantages of CPUs and GPUs by splitting the ray tracing algorithms in two separate parts – a compute-intensive part, and the parts with complex control flow – and execute those respective parts on GPU and CPU in parallel. Ray-triangle intersection is completely moved to the GPU, which was reported to be able to compute up to 114 ray-triangle intersections per second [Carr02], which was roughly 3–5 times as much as a the best published ray-triangle intersection rate on a CPU at that time[3]. To feed the GPU with ray-triangle intersection tasks, the CPU performed the traversal of rays through the acceleration data structure, and forwarded the intersection tasks to the GPU.

Inclucing scene management and traversal on the CPU, they have reported an effective rate of 218,000 rays per second in the teapot model, which already met the lower end of the fastest availabel CPU ray tracer at that time: However, the ray engine suffered from several architectural limitations: First, the approach required too much communication between CPU and GPU, which often does not pay off due to the high communication cost. For example, the cost for sending the data for a ray over a PCI bus is rather high compared to just performing the intersection on the CPU itself. Even worse, the traversal algorithm on the CPU depends on the results of the intersection computations, requiring read-back from the GPU, which is both rather slow and has very high latencies. Finally, achieving optimal performance would have required to use both CPU and GPU at full utilization by balancing the cost for traversal and intersection. This however is quite hard, as the ratio of traversal cost over intersection cost changes from scene to scene, and often from frame to frame.

Thus, while the ray engine did achieve a speedup over their own software implementation, it never actually matched the speed of the RTRT/OpenRT system running on a single CPU. Still, the ray engine was a highly interesting bit of research, as it has clearly shown two facts: First, that GPUs are considerably faster in compute intensive tasks. And second, that hybrid approaches are too complicated, and that using GPUs for ray tracing requires to map the *entire* ray tracing algorithm to the GPU as done by Purcell et al.[Purcell02].

[3] The best published results at that time resulted from [Wald01a], which reported 20–40 million ray-triangle intersections per second on a 800MHz Pentium III CPU (in 2001).

5.2.2 Ray Tracing on Programmable Graphics Hardware

In contrast to Carr's work, the goal of Purcell et al. was not to use the GPU as a coprocessor, but rather to map the whole ray tracing algorithm to the GPU. Their basic idea [Purcell02] was to view the GPUs as a stream processing hardware[4] (see Figure 5.3): Rendering a screen-aligned quad with a programmable shader can be viewed as executing a kernel (i.e. the programmable shader) on a stream of input elements (i.e. the fragments of the quad). The input data for each stream element can be read from corresponding screen-aligned textures, will be processed by the kernel-code during per-pixel shading, and will be stored in the frame buffer after processing. Finally, the loop can be closed by copying the frame buffer to a texture as an input for the next kernel. Additionally, the kernels can also access shared global data, which can also be stored in textures. As the newest generations of graphics hardware support floating point formats for both textures and frame buffer, all operations can be performed in floating point.

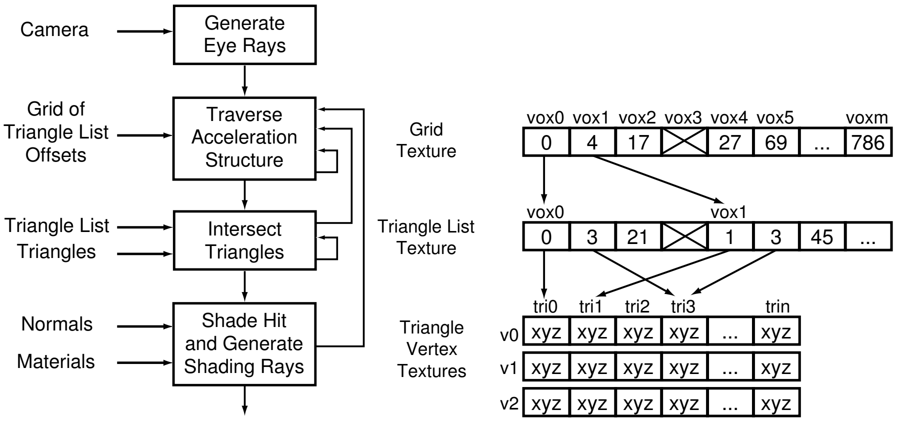

Camera
Generate Eye Rays
Grid of Triangle List Offsets
Traverse Acceleration Structure
Triangle List
Triangles
Intersect Triangles
Normals
Materials
Shade Hit and Generate Shading Rays
Grid Texture
vox0 vox1 vox2 vox3 vox4 vox5 voxm
0 4 17 27 69 ... 786
Triangle List Texture
vox0 vox1
0 3 21 1 3 45 ...
Triangle Vertex Textures
tri0 tri1 tri2 tri3 trin
v0 xyz xyz xyz xyz ... xyz
v1 xyz xyz xyz xyz ... xyz
v2 xyz xyz xyz xyz ... xyz

Figure 5.3: The GPU as a stream processing hardware. Left: Each pixel corresponds to a stream element. Rendering one screen-aligned quad with an appropriate programmable shader then executes a "kernel" over each pixel, reads the data from corresponding screen-aligned textures, and writes the processed stream elements into the frame buffer. Right: Organizing the scene data in textures. (Image courtesy Timothy J. Purcell)

In order to realize a ray tracer on such a hardware architecture, Purcell et al. have reformulated the ray tracing algorithm in a streaming manner: Each ray corresponds to a stream element, and each operation that is performed on

[4] Note that there are ongoing discussions on the actual programming model offered by GPUs: While most researchers see the GPU as a streaming processor, other researchers view it as a general parallel processor, a SIMD processor, or a vector processor.

a ray – e.g. ray-object intersection, grid traversal, or shading – is expressed as a small compute kernel, which in turn is realized in a programmable shader.

For example, one would start with a “ray generation” kernel to generate a ray for each pixel. The rays generated by this kernel are then be stored in screen-aligned textures, i.e. one 3-float texture for ray origin, and one 3-float texture for ray direction. Other kernels, e.g. a ray-triangle intersection kernel, could then read this ray data for each pixel by accessing those textures.

All global data, e.g. the triangle descriptions, the acceleration structure, etc., then has to be stored in textures, from where they can be read by the kernels. “Pointers” between different data items can be realized by using one-dimensional float textures, whose float values can be used as texture coordinates to index into a second texture using a dependent read operation[5]. Figure 5.3 shows how the scene was organized in several textures as used by Purcell et al [Purcell02].

Whereas the just discussed “ray generation” kernel can simply be executed for all pixels, control flow is gets much more complicated for other algorithms: During traversal, different rays may perform different operations: Some rays may still be traversing the grid, others are already intersecting triangles, and yet other rays might already be shaded. On a streaming architecture, each stream would always only contain rays that undergo the same operation.

As however the screen-aligned kernel is actually called for *every* pixel, the streaming control flow has to be emulated by “skipping” pixels that should not undergo the current kernel computations. This can be realized by marking each ray to be in a certain “state”, which specifies which operations it is subject to. For example, a ray in state “traversal” would only undergo traversal operations. After entering a non-zero voxel, the intersection kernel would then change the rays state to “voxel intersection”, putting the ray into the “voxel intersection” stream. As such, rendering a quad with an “traversal” kernel would perform exactly one traversal step for each ray in “traversal” state, a “intersection” kernel would compute exactly one intersection for each ray intersecting a triangle, and so forth.

Using the just described approach makes it possible to map the whole ray tracing algorithm to the GPU, without the need for costly communication or feedback to the CPU. In a simplified view, all the CPU has to do is issuing screen-aligned quads with the programmable shaders for the different kernels.

Having the full ray tracing algorithm on the GPU allowed for exploiting the full performance of the GPU (see Figure 5.4). In a prototype system

[5] Note that the “index” actually is a float value, as GPUs currently do not support integers.

running on a Radeon 9700 Pro [ATI02], interactive frame rates of up to 3 frames per second including shadows and reflections have been reported at least for resolutions of 256x256 pixels and relatively simple scenes. In a "Cornell Box", even frame rates of 15 fps at 256x256 pixels could be achieved including shadow computations. While this already corresponds to roughly 2 million rays per second, the results are likely to significantly benefit from the improvement of next-generation graphics hardware.

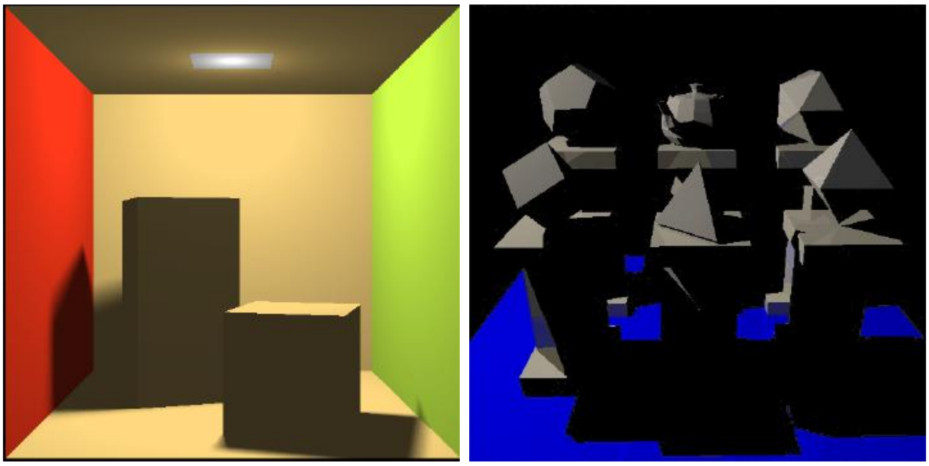


Figure 5.4: Examples from Purcell's ray tracer on a Radeon9700 Pro GPU, rendered at 256x256 pixels. Left: Cornell box scene ray traced with soft shadows, running at up to 15 fps. Right: "Teapotahedron" scene ray traced with shadows and reflections, with up to 3 fps. (Image courtesy Timothy J. Purcell)

However, the limitations of graphics hardware still poses several problems: For example, having to store the complete scene in texture memory severely limits the complexity of the scenes that can be rendered. Furthermore, recursion is hard to implement on graphics hardware (due to the lack of branching and dependent writes), making it hard to use hierarchical data structures such as kd-trees, and restricting the proposed approach to less efficient regular grids that do not require recursion. The lack of recursion also makes it hard to implement full-featured recursive ray tracing.

Furthermore, current hardware is quite limited in the number and kind of "write" operations, incurring some overhead due to the need to break up one kernel with too many writes into a larger number of smaller kernels with the correct number of write ops.

Another drawback of the approach is that it is quite bandwidth-intensive. Often, a rather small kernel has to read in data from many textures, and then only performs a simple operation on it. In order to pass any information to the next (probably just as small) kernel, it first has to write all temporary data to a texture, which in turn the next kernel has to read.

Finally, the proposed way of having rays in different states and having the kernels "skip" rays in the wrong state becomes increasingly inefficient for a larger number of computational states (that are likely to be required for practical applications). Though certain optimizations are possible, the proposed solution is still a poor substitute for the actually preferable, missing branching behavior.

In his work, Purcell has shown that it is indeed possible to do ray tracing on a GPU. However, while it is certainly true that GPUs have a significant performance advantage over CPUs in compute intensive tasks, ray tracing is actually much more control-flow intensive than compute intensive. While it is important to know that it is at least possible to realize such complex control flow *at all* on a GPU, the overhead incurred by realizing it currently negates its advantages.

This drawbacks of using GPUs for ray tracing can probably be avoided by future GPU architectures if they allow dependent writes, branching, and shader programs of higher than current complexity. At that point however, a GPU has actually evolved into a full-featured CPU.

5.3 The SaarCOR Realtime Ray Tracing Architecture

Apart from writing software ray tracers to either run on CPUs or on programmable GPUs, ray tracing can also be realized by mapping the whole algorithm to custom hardware. In the past, building hardware support for ray tracing has been pursued by several researchers: Most approaches focussed on volume ray casting, since this is considerably less complex than full-featured, recursive ray tracing for polygonal scenes. For volume ray casting, several practical solutions with realtime frame rates have been proposed, e.g. [Meissner98, Pfister99, Pfister01, Meißner02]. For full-featured ray tracing, however, most approaches supported only parts of the ray tracing process, and were thus limited to accelerating offline ray tracing that could not achieve interactive frame rates, e.g. [Humphreys96, Hall01, Pulleyblank87, Kedem84, Leray87]. Even with these severe restrictions so, some of these approaches even made it into commercial products [ART, Hall01]. Nonetheless neither of these approaches have ever been designed for, nor have they even targeted realtime ray tracing of polygonal scenes.

The reason for this lack of realtime ray tracing hardware is that it requires a considerable amount of hardware resources that until recently have been hard if not impossible to realize: For example, ray tracing hardware usually requires to store the whole scene in on-board memory. This requires many megabytes of on-board memory that would have been unaffordable only a few years ago. Quite recently, it has been shown that a virtual memory architecture can help in reducing the amount of physical memory required for a ray tracing hardware: As shown by Schmittler et al. [Schmittler03], such a virtual memory architecture needs only a small fraction of the scene data in physical on-board memory, and can page in missing data from host memory if required. This allows for interactively ray tracing even highly complex scenes with only a small amount of on-board memory[6]. Only a few years ago however even these significantly reduced memory requirements would have been impossible to realize.

Even more importantly than physical memory is chip complexity and floating point performance: Not too long ago even a single floating point unit was considered costly in terms of the required numbers of transistors. Ray tracing however does require floating point computations, and – at least in order to reach interactive performance – requires to have many of such units available in parallel and on the same chip.

In summary, the hardware resources for realtime ray tracing hardware have simply not been available a few years ago, thereby severely limiting the results of previous attempts towards ray tracing hardware. Only recently, progress in hardware technology has removed most of these limitations. Typical on-board memory sizes are sufficient for ray tracing, and many parallel floating point units can be put on a single chip. Even today, standard graphics chips are capable of performing hundreds of floating point computations in each cycle. As such, the raw hardware resources for ray tracing are becoming readily available.

5.3.1 Basic Concepts of the SaarCOR Architecture

Even if the both memory size and raw floating point power are now available, many problems remain: Though it has become reasonably easy to put many parallel FPUs on a chip, it is often quite hard to feed these functional units fast enough to keep them busy, due to limited off-chip bandwith, and dependencies in the calculations.

This is especially problematic for a bandwidth-intensive tasks, and for tasks with almost-random and unpredictable memory access patterns. Un-

[6] Though the scene still has to fit into the main memory of the host machine

fortunately, both are true for classical recursive ray tracing: Each ray often performs rather few operations on each data it accesses (e.g. one cheap traversal step on a whole BSP node), and unrelated rays usually access memory almost randomly. As such, a chip that simply traces hundreds of individual and unrelated rays in parallel is not feasible, even though its required raw floating point performance could be realized.

5.3.1.1 Packet Tracing

Last year, Schmittler et al. [Schmittler02] have shown that these bandwidth limitations can be overcome by using the same ideas used for fast software ray tracing [Wald01a] (also see Section 7): Tracing packets of coherent rays can dramatically reduce the average bandwidth required per ray – both off-chip as well as on-chip bandwidth: Coherent rays usually access the same BSP nodes and triangles. If such rays are traced together, each accessed data value can be reused for many rays, thereby reducing the average amount of data fetched per ray.

Actually, the SaarCOR architecture carries these concepts even further than the software implementation, as the packet sizes used in SaarCOR are usually much larger than used in RTRT/OpenRT (usually 8x8 or 16x16). In the software implementation, experiments with larger packet sizes have shown that more than only four rays per packet will not pay off, as the SIMD architecture cannot perform more than four floating point operations in parallel, and since the ray tracer is not bandwidth bound anyway. In fact, larger packets get costly for the software implementation due to a higher overhead.

scene	#triangles	#lights with shadows	reflection-depth	textures
Cruiser	3 637 101	0	0	–
BQD-1	2 133 537	0	0	–
BQD-2	2 133 537	0	0	–
Quake	39 424	0	0	bilinear
Conference	282 000	2	0	–
Office	33 952	3	3	–

Table 5.1: Some statistical data on the scenes used for benchmarking. The scenes have been chosen to test a wide range of parameters: from simple to complex geometry, and with and without textures, shadows and reflection. For the respective images, see Figure 5.6.

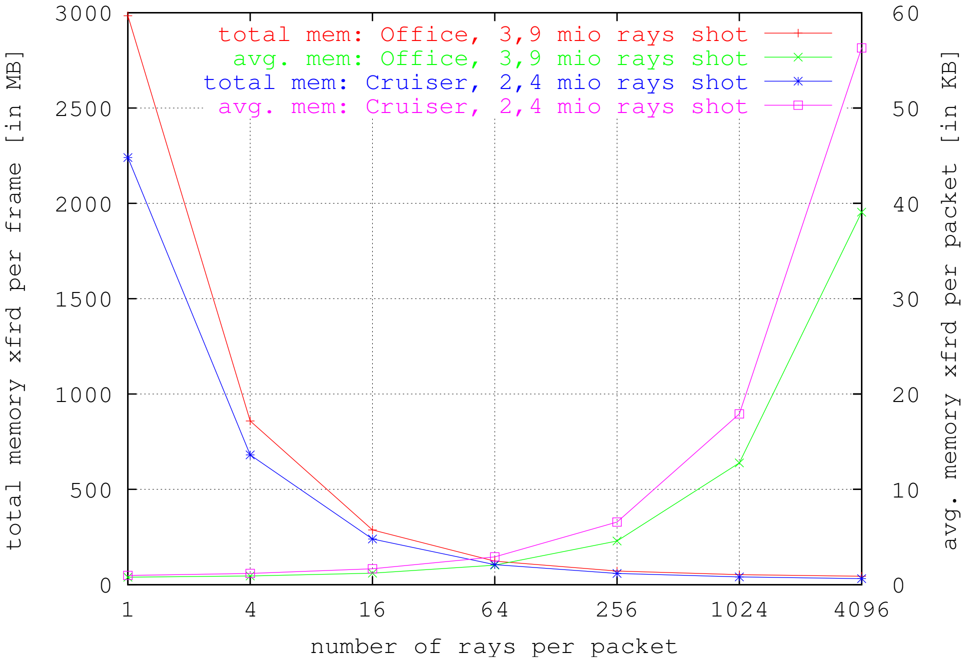


Figure 5.5: Influence of the number of rays per packet on the amount of memory transfered during rendering of one frame, and the average amount of memory requested by a packet. As can be seen, the total amount of memory transfered drops quickly and significantly with the number of rays per packet (data courtesy of Jörg Schmittler [Schmittler02]).

For the hardware architecture, however, larger packets are still beneficial (see Figure 5.5): The larger the packet, the total memory has to be transfered per frame. However, larger packets are also likely to traverse BSP regions that not all rays would have required to traverse if traced alone. While this results in considerable overhead for SIMD implementations, this does not happen for the SaarCOR architecture, as the hardware contains some simple logic to track which rays in a packet are "active" in a given subtree, and does not perform any more computations on those rays. As such, incoherent packets that traverse regions in which only part of the rays would have traversed just become smaller (resulting in less bandwidth reduction), but will not perform any excess computations.

Furthermore, at the targeted full-screen resolutions of $1k \times 1k$ pixels (probably plus antialiasing) even packets of 8×8 or 16×16 rays will be highly coherent. This allows for using rather large packet sizes, which dramatically reduces the required bandwidth, and thus enables to almost completely overcome the bandwidth issue.

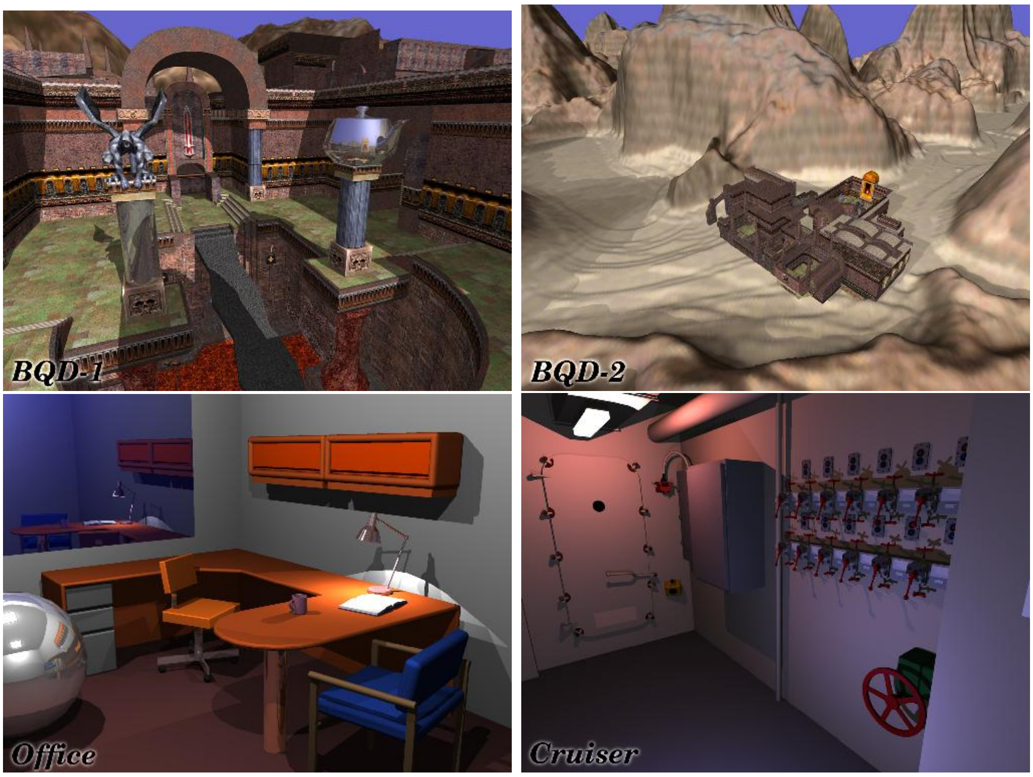


Figure 5.6: Some of the scenes used for benchmarking. Some statistical data on these scene is summarized in Table 5.1.

5.3.1.2 Simultaneous Multi-Threading

The second fundamental problem of ray tracing hardware is the high degree of data dependencies: Each ray usually has to fetch one data value (e.g. a BSP node), process it, and – depending on the outcome of this processing – fetch the next node, and so forth. As each fetch depends on the previous traversal step, prefetching the data is not possible[7]. If any such data value is not readily available, it has to be read from off-chip memory, and will arrive with considerable latency. If unhandled, this would lead to excessive pipeline stalls, idle times, and low performance of the hardware.

To avoid such idle times, SaarCOR uses a concept similar to *HyperThreading* [Intel02c], but with a significantly larger number of virtual threads, as inspired by the SB-PRAM architecture [Paul02]: To realize this concept, each pipeline in the SaarCOR architecture processes several (e.g. 16) number of packets at the same time, and switches to another packet after each traversal

[7] Though the node obviously can be prefetched, the time span between one travesal step and the next is much too small to hide a the high memory access latency.

step. Like the SB-PRAM, SaarCOR is built in such a way that the entire memory latency (even from off-chip memory) can be hidden by working on other packets during that time.

5.3.2 Hardware Architecture

The SaarCOR architecture is a very modular architecture, and can be configured in many different ways: The main ray tracing computations (traversal and intersection) are performed in a *ray tracing core* (called RTC in Figure 5.7), which consists of three stages: traversal, triangle-list-fetch, and triangle intersection. Each operation is always performed on a whole packet of rays. Together with its corresponding unit in the *ray generation and shading unit (RGS)*, each RTC realizes a full ray tracing pipeline. Each pipeline also implements the HyperThreading mechanism as described above.

To account for the difference in cost of traversal and intersection, all stages inside one pipeline can again be realized by multiple functional units, for example a pipeline could have 4 traversal units, but only one intersection unit. The exact number of units is configurable, as are packet size and number of parallel threads.

Additionally, the SaarCOR architecture supports a configurable number of parallel pipes (see Figure 5.7). The different pipes are all controlled by the RGS unit, which assigns ray packets to the individual pipes, similar to the techniques used for parallelizing the software architecture, see Section 8.

Due to the previously mentioned bandwidth reductions, multiple pipes can share the same memory controller. Even more importantly, data flow in SaarCOR is quite simple, and does not require costly data routing mechanisms between the different functional units.

5.3.3 Results

As the SaarCOR hardware architecture has not been physically built yet, all experiments can only be simulated. For this task, a special simulator has been developed, which uses a modified version of the RTRT software system to gather data on real-world rays including their memory accesses. The resulting raw data is then fed into a cycle-accurate simulator, that simulates the performance (including all effects like memory accesses, caches, latencies, bus-contention, unavailability of resources, etc) of the SaarCOR hardware. The simulator allows for simulating all the parameters of the SaarCOR hardware, like number of pipes, number of threads, cache size, and caching method, etc.

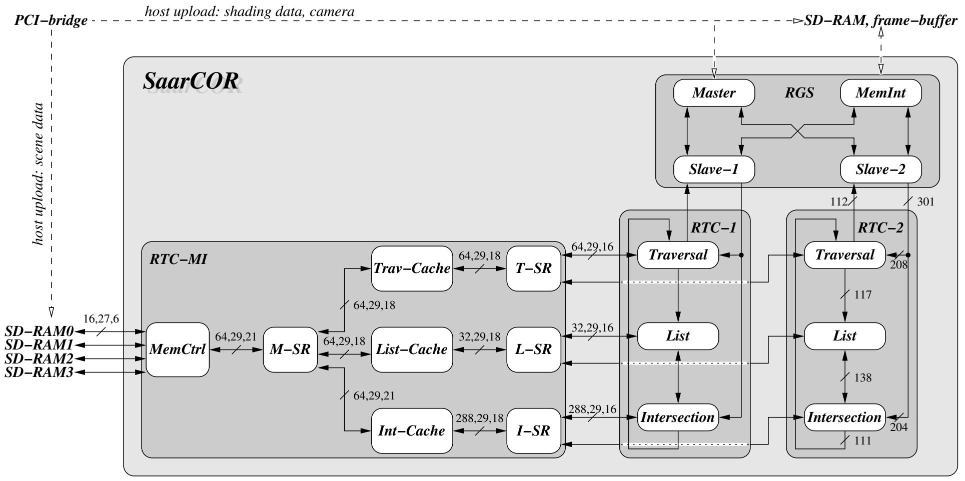


Figure 5.7: Overview of the SaarCOR hardware architecture: The SaarCOR hardware model is split into three parts: The ray tracing core (RTC), the ray-generation and shading unit (RGS), and RTC's memory manager (RTC-MI). To allow for easy scaling, several RTCs are supported. Please note the simple routing scheme used: it contains only point-to-point connections and small busses, whose width is also shown separated into data-, address-, and control-bits (courtesy of Jörg Schmittler).

Using this simulator, a large number of scenes has been tested. For a complete overview of these experiments and their results, refer to the original SaarCOR paper [Schmittler02]. Note, however, that most of the results published in the original paper are outdated, as some newer developments (like better BSP trees from the software implementation) have considerably improved these early results. Some of the newer resuls are shortly summarized in Table 5.3.3.

scene	Cruiser	BQD-1	BQD-2	Quake	Conference	Office
Frame Rate	170	137	59	129	77	44

Table 5.2: Simulated performance of a standard SaarCOR chip in different scenes. Numbers denote frames per second at 1024 × 768 *pixels. Note that the different scenes shoot a varying number of rays, due to varying reflectivity and number of light sources, see Table 5.1.*

Even in the originally published results a “Standard Configuration” of the SaarCOR architecture (which even requires less hardware resources than a state-of-the-art rasterisation chip), realtime frame rates at full-screen resolusions could be reached for OpenGL like-shading (with more than 100 fps even for highly non-trivial models, see Figure 5.6). Additionally the SaarCOR architecture allowed to easily scale the performance of the chip by adding more pipelines, resulting in an almost linear scaling in number of pipelines.

Being based on ray tracing, the SaarCOR architecture showed a logarithmic behaviour in scene size, making it possible to scale to highly complex models of up to millions of triangles while still maintaining realtime frame rates.

Finally, SaarCOR allowed to also compute effects like shadows and reflections at interactive rates. In fact, SaarCOR has shown to be roughly linear in the number of ray that have to be traced.

In summary, these results allow for drawing some interesting conclusions: First, that realtime framerates for full-screen, full-featured ray tracing can indeed be realized with todays hardware technology. Second, that packet tracing combined with simultaneous multithreading allow for efficiently removing the bandwidth bottleneck memory latency issues of ray tracing hardware. And finally, that even for OpenGL like shading at comparable frame rates, this ray tracing hardware would require less hardware resources than are used on todays graphics cards, meaning that for many tasks, ray tracing would already be more efficient than rasterization already today.

5.3.4 Remaining Problems

Even though the results obtained by the SaarCOR architecture have been quite impressive, many questions still have to be adressed: First of all, it yet remains to be seen how the integration of programmable shading into the SaarCOR architecture would influence some of the results. For example, it is not clear whether the same bandwidth reductions could also be realized for packet shading as have been achieved for packet tracing. Furthermore, costly programmable shading is likely to actually become more costly than the cost for tracing a ray. This trend can already be seen in the RTRT/OpenRT software ray tracer, and is likely to also be true in the SaarCOR architecture.

Furthermore, it is not yet clear how large packets of rays can efficiently by maintained in the context of complex programmable shaders and large packet sizes. While it is rather easy to create coherent ray packets for a fixed lighting model and for point light sources, this gets much more complicated for arbitrary shaders. In a fully programmable setting where each surface point might spawn an arbitrary number of arbitrary rays, eventually some kind of sorting and scheduling would have to be performed to sort all these rays into a small number of coherent packets. This problem has not yet been investigated.

Finally, efficient handling of dynamic scenes also has to be investigated. While some of the algorithms from the RTRT/OpenRT system (see Section 9) can probably also be employed for the SaarCOR architecture, eventually some new algorithms (and probably hardware support) will be required for handling incoherent motion at realtime frame rates.

5.4 Towards Realtime Ray Tracing – Conclusions

On currently existing technology, there exist several options for realizing realtime ray tracing, including software ray tracing on commodity CPUs, ray tracing on programmable graphics hardware, and the design of special purpose ray tracing hardware.

So far, the above systems are mostly prototype systems that are not yet widely used for practical applications. Still, they already achieve impressive performance and enable applications that have been impossible with different technologies: Walk-throughs of massive models without the need for approximations (e.g. [Parker99b, Wald01c]), interactive visualization of massively complex scientific and volumetric data sets [Parker99a, DeMarle03], interactive visualization of non-polygonal surfaces [Parker98a, Muuss95a], interactive simulation of complex reflection patterns [Benthin02], interactive global illumination [Wald02b, Benthin03], etc. (for an overview of what is possible with interactive ray tracing today also see Chapter 11). As shown for example in Section 5.3, this increased flexibility, quality, and performance can even be realized with less hardware resources than used in today's rasterization technology.

All these approaches to realtime ray tracing are fundamentally different and have different advantages and disadvantages: Software ray tracing offers the largest flexibility because commodity CPUs impose no restrictions on the program they are running. On the other hand, software systems suffer from the fact that most CPUs concentrate on accelerating single-threaded applications, and thus cannot optimally exploit the inherent parallelism in ray tracing. Also, modern CPUs are designed for a much wider range of tasks are not optimized for ray tracing like computations.

In contrast, programmable GPUs can leverage the superior performance, bandwidth, and – especially – parallelism of modern chip design for ray tracing. Additionally, GPUs have become cheap and are commonly available in most PCs, providing a cost-effective approach to fast ray tracing on the desktop. However, the programming model of GPUs still imposes severe restrictions on the algorithms that can be implemented efficiently. As a result, ray tracers on GPUs today are usually not faster than CPU ray tracers. The restricted programming model of todays GPUs usually leads to a significant overhead by forcing the use of less optimal algorithms (e.g. uniform grids vs. kd-trees, etc). This overhead then often over-compensates the performance advantage. While some of these restrictions will eventually disappear, it seems unlikely that GPUs will ever be as flexible as CPUs.

Finally, special-purpose hardware promises optimal performance given current hardware technology, but is obviously more expensive and time-consuming to realize. However, the performance of special-purpose hardware is what other implementations will have to be judged against.

The presented approaches cover a wide spectrum in terms of flexibility (software ray tracing), cost efficiency (GPUs), and performance (SaarCOR). While it is not yet clear which one will be most successful, they *all* have the potential of eventually providing realtime ray tracing performance on every desktop.

Part II

The RTRT/OpenRT Realtime Ray Tracing Engine

Chapter 6
General Design Issues

> *"Some argue that in the very long term, rendering may best be solved by some variant of ray tracing, in which huge numbers of rays sample the environment for the eye's view of each frame. And there will also be colonies on Mars, underwater cities, and personal jet packs."*
>
> *Tomas Möller and Eric Haines,*
> *"Real-Time Rendering", 1st edition (page 391)*

After the first part of this thesis has concentrated on the foundations and state of the art in ray tracing and realtime ray tracing, this second part is going to take a closer look at the RTRT/OpenRT realtime ray tracing system. In the following chapters, most of the actual algorithms and implementation issues of this system will be presented and discussed in detail. Before doing this however it is important to first discuss the higher-level design decisions that had to be taken when designing and building this system.

When setting out to build a realtime ray tracing system, there are many issues to be addressed. For example, one has to decide

- whether one wants to support only triangles vs. also supporting more general primitives,
- which kind of index structure one to use (e.g. grids vs. BSPs),
- whether to optimize for processor-specific extensions or rather targeting a portable system;
- whether to target commodity hardware (i.e. desktop PCs and clusters) vs. concentrating on shared-memory machines

- whether to design new, better algorithms or rather use a brute-force approach,
- what kind of compromises one is willing to make for the sake of performance (e.g. static-only scenes vs. dynamic scenes, recursive ray tracing vs. only ray casting, etc).

Though already quite long, this list is all but complete. Even worse, none of these questions can be answered by one final, definite answer. For example, realtime ray tracing can be realized just as well on PC clusters as on shared-memory machines[1]. Finally, many of these design decisions influence others. For example, some kind of index structures may be better suited for certain hardware platforms than others, whereas it might be vice versa on another architecture [2]). As such, none of these issues may be addressed in isolation, and have to be seen in the context of the whole system. Therefore, we will discuss some of these design questions below before describing any of the actual implementation details.

6.1 General Design Decisions

All the previously mentioned design issues are important issues to address, and heavily influence the outcome of the project. However, it is even more important to first discuss two questions that determine the overall "design philosophy" that one is going to follow: First, whether one prefers a "brute force" philosophy or a "smart" philosophy, and second, whether one uses a "top-down" approach or rather a "bottom-up" approach.

Smartness vs. brute force: In the "smart" approach, one strives to derive new, clever algorithms that achieve the same goal (i.e. producing a certain image) in a more clever way, e.g. by doing interpolation wherever possible instead of tracing certain rays. In the brute-force approach, one rather implements the same simple, well-known algorithm, but tries to achieve that as efficient as possible by optimally combining the most efficient algorithms, low-level optimizations, and available hardware resources.

[1] Compare e.g [Parker99b] and [Wald01c].

[2] For example, grids are better suited for implementation on a GPU [Purcell02] than kd-trees, whereas kd-trees lend better to implementation on processors with SIMD extensions [Wald01a]

Top-down vs. bottom-up: This determines whether one wants to redesign a completely new system from scratch (bottom-up), or whether one wants to progressively optimize an existing system (top-down).

Once this general design philosophy has been determined, the questions to many of the above design questions follow naturally.

6.1.1 Smartness vs. Brute Force

In the past, most approaches towards accelerating ray tracing have followed the "smart" approach, and have tried to avoid doing certain work for the same outcome (also see the overview in Section 3). With this approach however there is the continuous danger to invest more work for avoiding a certain operation than just performing that operation in the first place. For example, evaluating several heuristics to decide whether a single ray has to be traced or not may easily cost more than just shooting this ray, especially if the ray shooting operation is implemented efficiently[3].

In contrast to this, brute-force optimizations usually do not introduce additional cost factors or performance tradeoffs. Furthermore, brute-force approaches have proven very successful in other applications. In hardware rasterization, for example, todays most efficient graphics chips are just extremely optimized brute-force implementations of triangle rasterization as used ten years ago. Finally, a huge advantage of the brute-force approach is that such optimized algorithms still perform exactly the same operations as the original algorithms (just faster) and do not have to rely on special assumptions (e.g. on the type and order of rays shot). In contrast to this, "clever" approaches often rely on certain assumptions, fail if these assumptions and restrictions are not met, and thus are not applicable for the general case.

With all the work that over the last two decades has been invested in optimizing ray tracing, a common impression is that a brute-force approach might not carry far for ray tracing because of little potential being left for further optimizations. Though this argument sounds convincing, it is wrong nonetheless: While it is true that many people have worked on optimizing certain and isolated parts of a ray tracer (e.g. faster ray-triangle intersection [Möller97, Möller, Badouel92, Held97, Woo90, Haines91b], and

[3] This does not imply that following "clever" approaches does not make sense at all: While trying to avoid shooting single rays may not pay off, such approaches may still be very successful if they help avoiding much more costly operations. The render cache for example (see Section 3) can hardly accelerate a realtime ray tracer, but may still be highly beneficial for a full-featured global illumination system)

better acceleration structure construction and traversal [Glassner84, Kay86, Fujimoto86, Amanatides87, Arvo87, Goldsmith87, Arvo88, Jevans89, Samet89, MacDonald89, Arvo90a, Haines91a, Sung91, Kirk91, Spackman91, Hsiung92, Sung92, Cohen94, Cassen95, Whang95, Klimaszewski97, Smits98, Simiakakis95, Havran01, Subramanian90a]), most of these achievements have been proposed in isolation. For example, a system with a new, better traversal method does not automatically use the best intersection method, the best construction method, or the best implementation. Furthermore, most of the previous approaches to accelerating ray tracing have concentrated on algorithmic improvements (such as better acceleration data structures), while few effort has been spent on optimizing towards modern CPUs[4]. On the other hand, those approaches that have concentrated on efficient implementations have often neglected algorithmic issues. Finally, many optimizations to ray tracing have been derived quite some time ago, and have therefore been designed for architectures that are fundamentally different from a modern CPU with different limitations and bottlenecks. This leaves plenty of room for performance gains when building a complete ray tracing system.

With this in mind, the RTRT system was designed to follow a brute-force approach, with the general design philosophy to not re-invent the wheel, but rather to pick the best algorithms that have been developed over the recent years, to combine them in an optimal way, and to put special emphasis on an efficient implementation that is optimized towards modern CPUs.

Surprisingly enough, when starting the project it was not even clear what level of performance could be expected to be achievable at all. While the algorithmic complexity of ray shooting has been sufficiently investigated (and is commonly agreed to be $O(logN)$ on average [Szirmay-Kalos98, Havran01]), is was totally unclear what kind of practical performance would be possible to achieve with today's hardware resources, i.e. how many rays per second one would be able to shoot in a given scene an on a given CPU. As the performance of a ray tracer is influenced by many factors, this question still is not answered sufficiently. Even the fastest ray tracers today (including the RTRT kernel) still leave plenty of room for optimizations.

6.1.2 Top-Down vs. Bottom-Up

Except for optimally combining the best available algorithms, a brute force approach obviously requires very efficient implementation and extensive low-level optimizations in order to exploit the available hardware as well as pos-

[4] While much effort has been spent on such specialized optimizations, few of such work ever gets published, as such low-level optimizations are often not considered "scientific research".

sible. In order to do this, it is usually better to design a new system from scratch than trying to optimize an existing system.

Optimizing an existing system has the obvious advantage that each improvement has an immediate benefit for already existing users and applications. Furthermore, existing systems allow for working with (and optimize for) real-world applications and data. However, existing systems also tend to be quite complex. As a result, many of the bottlenecks are not immediately obvious. Furthermore, many important design issues are fixed already, and cannot be easily changed without major effort. Furthermore, optimizing and profiling complex systems is complicated and error-prone, as each single change might influence many different parts of the system.

In contrast to this, building from scratch allows for starting with a small set of core routines, which can then be thoroughly optimized and profiled before adding new functionality. Adding new functionality incrementally also has the benefit that the cost for each added feature can instantly be determined when adding it. This makes the cost of each extended feature much more transparent, which in turn simplifies follow-up design decisions. For example, just clearing the frame buffer before casting the first rays hardly has a (percentile) influence on the rendering time of an existing offline renderer, and would as such hardly be identified as being a limiting factor. In a realtime ray tracing kernel however it would have a significant (relative) performance impact, and can be avoided before many such small cost factors have accumulate to a state where none of these individual small sources of overhead can be identified any more.

Because of these reasons, we have chosen the bottom-up approach in the sense that we have implemented everything from scratch. In this process, the “golden design rule” was (and still is) to optimize for speed first – with all compromises on portability, flexibility, supported features, etc – and to look out for ways to extend and enhance the concept lateron. Though this seems obvious, it is important to note that this approach – optimize first, extend lateron – also implies that from the very beginning, everything should always be kept as simple as possible[5]. When in doubt whether a certain feature might lateron be beneficial or not, it is safer to *not* implement it in the beginning, and only implement it once it is really needed. Once some code is implemented and carefully optimized, it is also important that it gets modified as little as possible when adding new features lateron. On the other hand, code that has once been optimized has to be continuously re-checked and re-optimized, due to newly available compilers and tools,

[5] This approach is also famous under the name “The KISS Principle”, where KISS is an abbreviation of “keep it simple, stupid”.

slightly changed program structure or slightly changed parameters of newly available hardware. Because of this, the core routines of RTRT (i.e. triangle intersection and BSP traversal) are continuously re-optimized, and have been completely re-implemented several times since their original implementation. Because of these continuous improvements, the newest implementation of the RTRT code (see Section 7) is – when measured on exactly the same CPU with the same clock speed – roughly two to four times faster than the original (already quite heavily optimized) implementation as presented in [Wald01a].

6.2 Efficiency and Optimization Issues

In order to achieve optimal performance, code has to be optimized towards the underlying hardware. Modern CPUs can be extremely powerful, but often achieve only a fraction of their performance if used incorrectly.

Due to various economic reasons, todays CPUs still follow the traditional "single-threaded" approach. To achieve their level of performance, they rely primarily on three techniques: Extremely high clock rates, very long pipelines, and extensive caching. Roughly speaking, increasing the pipeline length allows for further increasing clock speed, and bigger caches allow for longer pipelines.

Though this approach has been economically very successful, it is nonetheless becoming increasingly problematic, as pipeline stalls (e.g. through dependency chains, branch mis-predictions, or cache misses) get increasingly costly. Modern processors have several hardware features such as branch prediction, speculative execution, semi-automatic prefetching, out-of-order execution, and other techniques [Hennessy96, Intel, AMDb, AMDb] in order to avoid such pipeline stalls[6]. However, the success of these hardware approaches is fairly limited and depends to a large degree on the complexity of the input program code. The same in fact is true for compiler technology: Though modern compilers [Intel02a, GNU] perform highly sophisticated optimizations (e.g. to optimally reorder and schedule operations), their success is quite limited, and depends on the input code.

[6] In fact, todays CPUs spend more die area on techniques like branch prediction or speculative execution than they spend for their floating point units.

6.2.1 Reducing Code Complexity

Pipeline stalls are especially likely for code with many conditionals. Conditionals tend to create dependency chains [7], which often make it impossible for the CPU to predict what instructions will be executed in the near future. Without being able to predict which instructions will have to be executed in the near future, the CPU can not keep the pipeline filled with soon-to-be-executed instructions, and has to stall. Though the negative effects of conditionals can be lessened by hardware features such as branch prediction and speculative execution, these features are effective only up to a certain degree of complexity, and often fail for code sections containing many conditionals, especially if these are nested or depend on each other.

Complex code also often contains many function calls. Though these help in structuring complex code, they are extremely costly due to the need for many costly stack operations. Even though "inlining" of functions may help, the compiler may decide *not* to inline a function even if the programmer expects this. Especially "virtual" functions should be avoided wherever possible. Virtual functions cannot be inlined[8], always generate a pipeline stall, and are quite costly overall[9].

Therefore, we prefer simple code that contains few conditionals, and organize it such that it can execute in tight inner loops[10]. Such code is easier to maintain and can be well optimized by the programmer as well as by the compiler. These optimizations become more and more important as processor pipelines get longer and the gap between processor speed and memory bandwidth and latency gets bigger.

[7] Some dependency chains can by avoided by "dependent moves" (which in C-code correspond to the "question mark" operator "a?b:c"). A "dependent move" (`v=c?r1:r2`) avoids having to perform a conditional, and is usually significantly faster than (`if (c) v=r1; else v=r2;`). This optimization is often performed automatically by the compiler, which however cannot always detect this case.

[8] Virtual functions *can* be inlined if the callee type is known to the caller already during compile time, which is, however, usually not the case.

[9] Inserting a *single* virtual function call during ray shooting was measured to slow the overall RTRT performance down by up to five percent!

[10] Of course, this does *not* imply that the entire RTRT/OpenRT system is written in low-level, unstructured C-code. Such low-level optimizations are only used in the most performance critical sections, which only form a small fraction of the overall system. Most other parts are written in C++, and make extensive use of the concepts of structured and object oriented programming [Gamma94].

6.2.2 Optimizing for Memory and Caches

Keeping the CPUs functional units busy also requires to continuously keep it supplied with data to work on. Given the increasing gap between compute performance and memory bandwidth (and latency) this gets increasingly complicated.

Due to the traditionally high floating point demands of ray tracing – which in the past was poorly supported by the CPUs – ray tracing even today is often considered to be *compute bound*[11]. While this may have been true in the past, todays CPUs offer very high compute performance, but suffer from an increasingly bad ratio between compute performance to memory bandwidth. As such, most ray tracers today are actually limited much more by memory bandwidth (and its high latency) than by compute power: If the ray tracer requires too much memory bandwidth, the CPU requests data faster than the memory can deliver it, and has to stall.

In order to reduce accesses to external memory, modern CPUs have extremely large caches[12]. Even so, realistic scene sizes will never completely fit into processor caches.

Even if memory bandwidth were not the limiting factor, its *latency* is a major issues. For example, each access to memory – e.g. due to a cache miss – stalls the pipeline until the data is available. Given the high clock-speed vs. memory latency ratio, each single cache miss can therefore cost dozens to hundreds of cycles[13]. Obviously, the longer the pipelines, the more costly are pipeline stalls.

As a result, code and memory layout should be structured to reduce bandwidth, and especially to use the caches as well as possible. Most importantly this requires to reduce the “memory footprint” of the algorithm, a carefully designed and cache-friendly data layout of data structures, avoiding incoherent (random) memory accesses, alignment of data, and prefetching.

6.2.2.1 Prefetching

Data that is likely to be used in the future should be *prefetched*. Modern CPUs have special operations for cache control, with which data can be prefetched, and with which one can influence how data is being cached, e.g. in

[11] An algorithm is compute bound if its performance is mostly limited by the lack of sufficient compute performance.

[12] Modern CPUs often spend more than half their die area on on-chip caches!

[13] This makes many “traditional” optimization techniques problematic. For example, avoiding a costly operation by tabulating may lead to cache misses when reading the tabulated values, which may be more costly than performing the operation itself in the first place.

which level in the cache hierarchy to prefetch data, or whether to write data *into* the cache or route it *around* it (i.e. directly to memory) [14].

Prefetching can be very effective at hiding cache miss latencies. However, prefetching requires to know in advance which data will be needed for future operations. To be most effective, this requires to predict data accesses *at least* a hundred cycles in advance. This gets especially problematic for complex data structures, which often require "pointer chasing" to traverse. Here once again, simple code is preferable over complex code, as predicting data accesses in simple code is much easier for both the compiler and the programmer.

6.2.2.2 Alignment and Data Organization

Efficient cache usage also requires data to be aligned in a cache-friendly way. For example, data should be kept aligned with cache lines wherever possible. Compound data types that are not aligned have a certain probability of overlapping a cache-line boundary, thereby require fetching two cache lines from memory where one might have been sufficient. Furthermore, such data may also generate two cache misses while loading.

Furthermore, data that is likely to be used at the same time should be stored next to each other. This increases the chance of cache hits, as after each access to the first value it is guaranteed that the second value is already in the cache. Similarly, having both values stored in the same cache line allows for prefetching both with a single instruction. On the other hand, data that will not be used at the same time should be stored in separate places, to avoid spending half of a cache line for data that is not required at that time.

For the same reason we also separate read-only data (e.g. preprocessed triangle data as described below in Section 7.1.4 from read-write data such as mailboxes [Amanatides87, Glassner89, Kirk91]. If a mailbox would be stored with the triangle data (as it is usually implemented), an entire cache line would be marked changed even though only a single integer has actually been modified. This becomes a huge problem in a multi-threaded environment, where by constantly changing mailboxes each processor keeps invalidating cache lines in all other processors[15].

[14] Not storing data in the cache when writing it to memory is useful when writing data to memory that is likely not to be needed in the near future. If such data were written to the cache (which is the default behavior), it may "overwrite" other cached data that might be more likely to be needed in the near future (an effects that is known as "cache pollution"). This is especially useful if large blocks of such data have to be written, as these are likely to otherwise overwrite large parts of the cache.

[15] Note that this can be avoided by "hashed mailboxing"[Wald01a]: In hashed mailboxing, each thread stores its mailbox information in a small hash table that easily fits into

6.2.2.3 Storage Reduction and Bandwidth Reduction

For many applications, it can also be useful to compress data, e.g. by storing boolean values in chars or single bits instead of 32-bit integers, or using 16-bit `shorts` vs. 32-bit `ints`[16]. Compressing data allows for fitting more values into the cache, and at the same time reduces memory bandwidth for loading them. However, this often has to be performed very carefully, as the overhead for extracting the information from a compressed form may easily offset the gain achieved by reduced bandwidth and improved caching. For example, an older version of the RTRT core used to store the kd-tree splitting dimension (see below) in the *upper* two bits of another value (see [Wald01a]), which required a "shift" to extract them. Storing this value in the *lower* two bits instead allowed for replacing the shift operation with a bitwise "and", which resulted in an *overall* traversal speedup of up to 5 percent[17].

Data alignment and caching issues are extremely sensitive issues that are hard to "plan" and "design" correctly. Most often, the best solution has to be found by manual trial and error. Though this is often tiresome and time-consuming work, these issues are extremely important in order to extract the maximum performance out of a modern CPU. In fact, memory issues are probably as important for the speed of the RTRT system as any other issue. Furthermore, considering memory and caching effects gets even more important for every new processor generation, as the gap between CPU speed and memory bandwidth and latency will continue to increase.

6.2.3 Exploiting Processor-Specific SIMD Extensions

Because of their good price/performance ratio and ubiquitous availability, we have chosen to optimize towards commodity PC processors based on the `ia32/x86` architecture, i.e. the Intel *PentiumPro* [Intel01] and AMD Athlon [AMD03a] families.

Whereas floating point operations have traditionally been very expensive, todays processors are extremely fast at floating point code, with float operations often as fast as integer operations[18]. However, floating point units are

the processor cache.

[16] Note that this depends very much on the application. If cache usage is not an issue, accesses to `ints` on many architectures are faster than accesses to `shorts` or `chars`. (Side note: Accessing and operating with *unsigned* `ints` is usually faster than accessing signed values)

[17] This high performance impact can easily be explained by taking into mind that 'shift's have a much higher latency than 'and's, and that this operation is performed in *each* traversal step.

[18] Typically, these processors have several parallel pipelines, some for floating point op-

often not fully utilized due to badly written code, and due to the awkward programming model of the "standard" x87 FPU. The x87 FPU is operated like a stack machine, and often requires several FPU stack operations before performing an actual computation, leading to a bad FPU utilization.

In order to provide more floating point performance, newer processors provide SIMD[19] extensions, in which multiple (usually four) floating point operations can be executed with a single instruction. Examples of such extensions are Intel's SSE [Intel02b], AMD's 3DNow! [AMDa], and IBM/Motorola's AltiVec [AltiVec]. Whereas AMD originally introduced its own SIMD extension (3DNow!), it now also supports Intel's SSE instruction set. Generally, these processors achieve their best performance *only* when using these extensions. This is especially true for the newer members of these processor families: Whereas the AMD Athlon and Pentium-III processors have still been reasonably efficient at standard non-SSE floating point code, the Pentium-IV definitely requires to use SSE in order to achieve its peak performance. As such, we have chosen to derive algorithm that allow for the efficient exploitation of these SIMD extensions (see Section 7.2.3). Because almost all processors today supports Intel's SSE instruction set, we have opted for SSE[20]. Note that the described algorithms and data layout issues are not restricted to Intel's SSE, but apply similar to 3DNow! or others.

6.3 Efficient use of SIMD Extensions

As just described, efficient use of the SIMD capabilities is a key requirement for unfolding the full performance of modern CPUs. Unfortunately using these SIMD extensions is often quite complicated. As we will lateron use these extensions quite heavily (especially in Chapters 7 and 13.3), we first have to discuss some important issues that have to be kept in mind when trying when (re-)designing algorithms to make use of these extensions.

erations, some for integer and logical operations. These can best be used by code that has a good mix of floating point and integer/logical operations. As such, "optimizing" code by replacing floating-point operations by integer operations can back-fire, as the floating point pipeline runs idle, and the integer pipeline gets overloaded by the additional work that has to be done.

[19] SIMD = Single Instruction, Multiple Data [Hennessy96]

[20] SSE today is supported on both the Intel Pentium-III and Pentium-IV/Xeon processor families, as well as on AMD's Athlon MP/XP, and even on the 64-bit AMD Opteron/Athlon64. It seems likely that the SSE instruction set will also be supported in future Intel/AMD processor generations

6.3.1 Instruction Parallelism vs. Data Parallelism

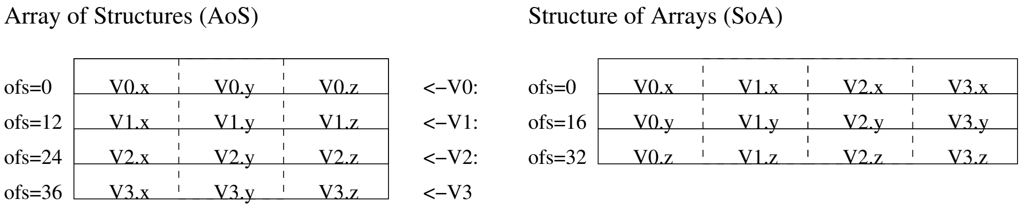
Perhaps the biggest issue with using SIMD extensions is that they can only exploit parallelism in programs if there is any, and as such often require to restructure the program to offer more parallelism. Parallelism can be exploited in two different ways: Instruction-level parallelism, and data parallelism. *Instruction-level parallelism* can be exploited by reordering existing sequential programs to combine independent operations for parallel execution. For example, a dot product of two four-dimensional vectors requires four multiplications, which can be performed in parallel with one single 4-float SIMD multiply. In contrast to this, *data parallel* approaches perform exactly the same operation on four different values in parallel. In the example of the dot product, a data parallel approach would perform four dot products at the same time.

The SSE instruction set has been optimized for data parallel approaches. For example, it cannot be efficiently used for the above example of accelerating a single dot product: Though the four multiplies can be executed in one single operation, adding the four resulting values together is a “horizontal” operation that does not fit the SIMD philosophy and as such is quite costly to implement with the SSE instruction set. If instead the code can be restructured (and data realigned) such that four such dot products can be performed in a data-parallel approach, these four dot products together can be performed faster than a single dot product on the `x87` FPU!

This also explains why data parallel approaches in general are preferably to instruction parallel ones: As mentioned above, instruction parallelism can only be exploited once there is any. Usually few algorithms (such as some matrix or vector operations) will allow for performing four independent operations in each cycle. If this is only the case every few instructions, the utilization of the SIMD units will drop accordingly. In contrast to this, many compute-intensive tasks often perform the same task (such as a dot product) several times on different data, which then can be transformed to a data-parallel form, in which a SIMD operation can be performed in almost every cycle. However, even data parallelism can be exploited only if there is any: If the algorithm only requires computing a *single* dot product (and not four dot products at the same time), data parallelism can not be used. Most often, this requires manual restructuring of the algorithm until such data parallelism is available.

6.3.2 SIMD-friendly Data Organization

Even if the algorithm itself offers enough parallelism to be exploited, efficient use of SIMD extensions imposes several restrictions on data organization: For example, SSE requires data to be aligned to 16 byte boundaries; access to non-aligned data can be several times as expensive as access to aligned data. Being able to best exploit the SIMD extensions thus often requires to restructure the program: Best performance can usually be achieved for "streaming"-like programs, i.e. programs that execute a simple sequence of SIMD operations on a large stream of data (such as adding long vectors, or multiplying large matrices). Furthermore, high SIMD performance requires data to be already in the caches (see above), few conditionals, few data dependency chains, and careful alignment of data.



Array of Structures (AoS)

ofs=0	V0.x	V0.y	V0.z	<–V0:
ofs=12	V1.x	V1.y	V1.z	<–V1:
ofs=24	V2.x	V2.y	V2.z	<–V2:
ofs=36	V3.x	V3.y	V3.z	<–V3

Structure of Arrays (SoA)

ofs=0	V0.x	V1.x	V2.x	V3.x
ofs=16	V0.y	V1.y	V2.y	V3.y
ofs=32	V0.z	V1.z	V2.z	V3.z

Figure 6.1: SIMD data organization. Left: The usual, convenient "Array of Structures (AoS)" organization for storing three four-float vectors. Right: The "Structure of Arrays (SoA)" organization as required for efficient SIMD computations. The latter typically requires manual reorganization of program and data structures, and is often awkward to use (especially for object-oriented programmers). However, it is a prerequisite for fast SIMD code.

Finally, using SIMD extensions requires to redesign all data structures: Whereas it is usually most intuitive to organize data in a "array of structures" (AoS) form (such as storing four vectors as `V0,V1,V2,V3`), efficient SIMD programming requires to reorganize such data into a more SIMD-friendly "structure of arrays" (SoA) form (in the previous example, as `V0.x, V1.x, V2.x, V3.x, V0.y, V1.y`..., see Figure 6.1). Though this SoA data organization is a prerequisite for fast SIMD code, it is often awkward and un-intuitive for the programmer, can not be generated automatically by the compiler, and as such requires manual restructuring of all data structures.

These issues related to using SSE instructions bear several implications for the data layout and algorithms used in the RTRT kernel. These will be explained below.

6.3.3 SSE Code Generation

Given proper (manual) data organization and presence of enough data or instruction parallelism, SSE code can be generated in three different ways: Automatic code generation by the compiler, manual assembly programming, and so-called “intrinsics” [Intel02a]. Modern compilers such as the GNU gcc 3.3.1 or Intel’s ICC 7.1 compiler [GNU, Intel02a] offer automatic “vectorization” of existing C-code, which can be quite efficient for streaming-like code operating on large arrays of numbers. However, initial experiments with having the compiler automatically vectorize intersection or traversal code have been quite disappointing so far, even if code and data structures have already been manually reorganized to a SIMD friendly form. On the other hand, writing SIMD code by manual assembly coding requires significant effort. In order to avoid the tiresome manual coding of assembler code, we use the “intrinsics” [Intel02a] offered by the Intel ICC and GNU gcc compilers [21]. Intrinsics allow for writing easily maintainable low-level SIMD code in a C-style manner that can then be tightly integrated with standard C/C++ code [Intel02a, GNU]. Using intrinsics also allows the compiler to perform automatic low-level optimizations such as loop unrolling, instruction scheduling, constant propagation, common-subtree-elimination, register allocation, etc. Given the extremely high complexity of modern processors – especially when considering multiple inter-operating pipelines and instruction reordering – a good compiler can usually perform such low-level optimizations much more efficiently (and often even better) than a human programmer.

6.4 Implications on the RTRT Core

In the previous sections, we have discussed some of the design issues that have to be taken into mind when setting out to build a high-performance system on modern commodity hardware. With these issues in mind, many of the design decisions mentioned at the beginning of this chapter now follow naturally. To summarize: As motivated above, overall design philosophy is to follow a brute-force approach, in which we combine the best available algorithms that are then combined and implemented as efficiently as possible. As explained above, this is best done using a bottom-up approach, in which we first implement and optimize the most performance-critical parts of the system, and add needed functionality only once it is needed[22]. In order

[21] Intrinsics are only available in gcc since version 3.3. Since that version, they are also compatible to ICC. As such, exactly the same code can be compiled with both compilers.

[22] For example, this implies that we first concentrate on static scenes and single CPUs only, and care about parallelization and dynamic scenes only lateron.

to achieve the highest possible performance, we have to explicitly design towards modern commodity CPUs. This in turn requires us to concentrate on memory- and caching-effects, to use algorithms with as simple as possible code, and to investigate means of efficiently using SIMD extensions, which we have to best use in a data-parallel manner.

So far, we have intentionally kept the discussion on a more general level, and have not yet explicitly applied them to ray tracing. In fact, all of these just mentioned issues would apply equally well to other high-performance applications on modern CPUs, and are not limited to ray tracing only.

As our eventual system nonetheless is building a realtime ray tracing system, this section discuss the implications of these design issues on the actual RTRT core.

6.4.1 BSPs, kd-Trees, BVHs, Octrees, or Grids

For traversing the scene, we use an axis-aligned BSP (binary space partitioning) tree [Sung92, Subramanian90b, Havran01]: Its traversal algorithm is shorter and much simpler compared to octrees, bounding volume hierarchies (BVH), and hierarchical grids[23]. While the traversal code for uniform grids is comparably simple, uniform grids can not adapt as well to scene complexity. Of course, we use axis-aligned BSP trees and not general BSPs with arbitrarily oriented splitting planes, as these are cheaper to traverse, and simpler to construct.

BSP tree traversal is most easily formulated recursively, and might therefore on first sight appear to contradict above mentioned goal of preferring simple code and reducing function calls. However, BSP traversal can be transformed to a compact iterative algorithm [Keller98, Havran01], with tight inner loops and few conditional.

Finally, BSPs lend naturally to SIMD implementation (see below) and thus allow for accelerating the traversal with SSE. It is unclear how SIMD extensions could be efficiently used for traversing grids[24].

[23] Quite commonly, axis-aligned BSP trees are also also called “kd-trees”, or simply “BSPs” (e.g. [Subramanian90a]). Unfortunately both terms are not ideally suited: “BSP” refers to the more general binary space partitioning tree with arbitrarily oriented splitting planes. and “kd-tree” also refers to a special form of higher-dimensional binary trees for efficiently storing points [Bentley75, Bentley79]. Such kd-trees are also used e.g. for photon mapping [Jensen01], but are fundamentally different from axis-aligned BSP trees as used in ray tracing. As such, the term “axis-aligned BSP tree” is most exact, and thus preferable. Unfortunately, it is also quite awkward. In the remainder of this thesis, we will equally use all three terms – BSP, axis-aligned BSP, and kd-tree – but will always refer to the concept of axis-aligned BSP trees.

[24] Though it seems obvious to just perform one grid traversal step each for four inde-

6.4.2 Triangles vs. High-Level Primitives

We have also chosen to exclusively support triangles as geometric primitives. As a result, the inner loop that performs intersection computations on lists of objects does not have to branch to a different function for each type of primitive (or even worse call a virtual function for each primitive to be intersected). By limiting the code to triangles we lose little flexibility as other surface types can be well approximated by triangle meshes [25]. While the number of primitives increases when tesselating more general primitives, this is more than compensated by the better performance of the ray tracing engine. Also note that similar observations hold for the shading process.

6.4.3 SIMD Ray Tracing

As described above, efficient use of the floating point units requires to use SIMD extensions. As discussed above, this requires to either exploit instruction level parallelism, or to take a data-parallel approach (see Section 6.3).

Taking a closer look at the typical algorithms in ray tracing, there is little potential for instruction level parallelism at all: The most compute-intensive task during ray tracing (and thus the most likely to offer lots of instruction parallelism for floating point operations) is triangle intersection. However, most of the floating point operations in the various triangle intersection algorithms are usually spent on dot products and vector products, which are not well suited for SSE implementation as described above. Furthermore, triangle intersection tests usually contain lots of conditionals, a complex control flow, a non-streaming data access pattern, and data many data dependencies. As a result of these reasons, ray-triangle intersection is badly suited for an instruction parallel implementation, especially given the restrictions of the SSE instruction set.

Unfortunately ray traversal offers even less potential for instruction-level parallelism than triangle intersection, as traversing a BSP node requires but a single subtraction and a single multiply, and otherwise only performs logical operations[26]. Performing multiple traversal steps at the same time is not possible either, as the input of each traversal step depends on the outcome of the previous one. Since neither triangle intersection nor traversal offer lots of instruction level parallelism, significant speedups by instruction parallel approaches can not be expected.

pendent rays, this usually won't work because the four rays will often require access to different grid cells, which doesn't easily map to implementation in SSE.

[25] Also see the discussion in Section 2.1.1.

[26] Similar arguments apply for other acceleration structures.

6.4.4 Data Parallel SIMD Ray Tracing

On the other hand, using ray tracing in a data-parallel fashion is not easy, either. For ray triangle intersection, there are basically two different approaches, either intersecting one ray with several triangles, or intersecting one triangle with several rays. Obviously, "several" in the context of SSE usually means either "four" or multiples of four. Both methods can be easily and efficiently implemented, and achieve near-perfect speedups (i.e. speedups close to four).

However, apply them in practice requires to take additional measures. For example, ray tracing typically traverses a single ray through a data structure and intersects it sequentially with each of the triangles visited during traversal. Thus, in order to take advantage of a fast data-parallel ray-triangle intersection implementation, we need to modify the ray tracing algorithm such that we always have four ray-triangle intersections available for computation each time we call the respective routine.

6.4.4.1 Alternative 1: Intersecting 1 Ray with 4 Triangles (1:4)

The most obvious of these two approaches is intersecting one ray with four triangles (in the following called the "1:4" SIMD approach). The 1:4 approach is easy to implement, and does not require any actual changes to the rest of the ray tracer except storing triangles that are likely to be intersected together in batches of four.

However, this approach in practice is quite problematic, as it is only efficient for rays that have to intersect many triangles at the same time. Unfortunately this is usually not the case in ray tracing, as the whole idea behind the use of acceleration structures is to reduce the number of triangles visited in each traversal step to an absolute minimum. For example, well-built BSP trees often have only one or two triangles in each voxel, and often intersect less than four triangles on average (also see Table 7.5). If most voxels contain less than four triangles the 1:4 approach leads to bad utilization of the SSE units, and thus can not provide any significant speedup. Though it is theoretically possible to "gather" several intersections along different traversal steps, this approach in practice is not feasible either: First, this approach requires expensive data reorganization to combine triangles from different voxels to an SSE-usable format (see 6.3.2); and second, always continuing traversal until four intersections are available leads to the traversal of nodes (and intersection of the triangles therein) that might not have been traversed in the sequential algorithm (see Figure 6.2).

Apart from the question how to efficiently feed the 1:4 ray-triangle inter-

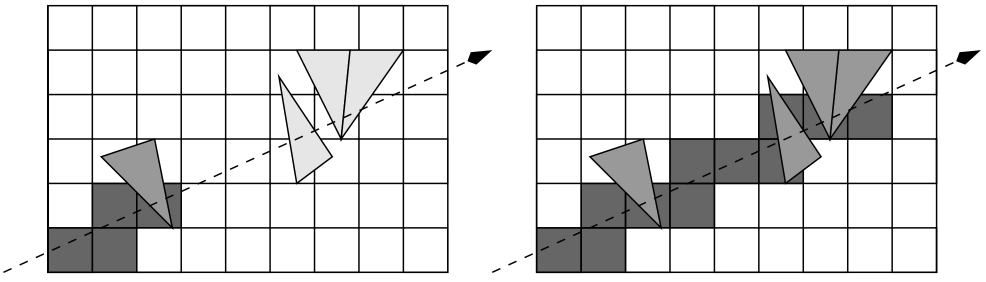


Figure 6.2: Left: Sequential traversal of a ray through a grid data structure, intersecting one triangle. In that case, a 1:4 ray-triangle traversal would only perform one out of four potential intersections, leading to low SSE utilization and bad performance. Right: "Gathering" intersections (i.e. traversing until four intersections are available before doing the intersections) leads to even more overhead: In this simple (and common) example, it leads to both additional traversal steps through the data structure, as well as to intersection with triangles that would otherwise not have to be intersected at all.

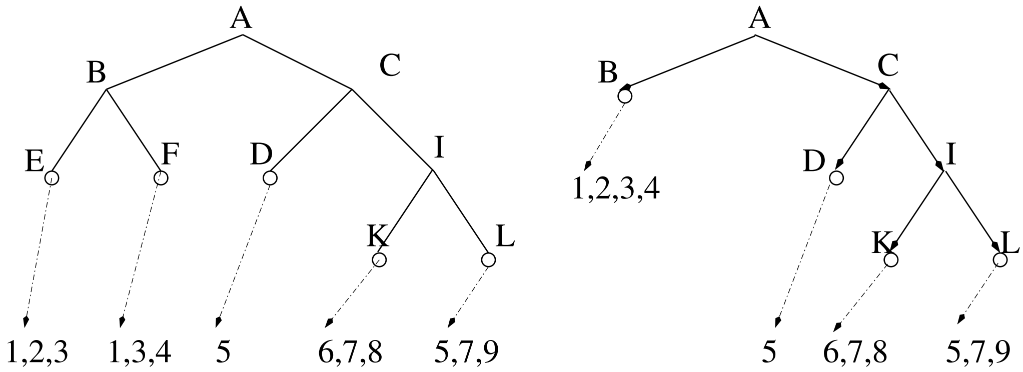


Figure 6.3: Being able to intersect four triangles at once with a fast 1:4 SIMD intersection routine allows for shallower BSP trees: If the cost for intersecting four triangles is the same as for intersecting a single triangle, the leaves "E" and "F" (left side) can be safely combined to one larger leaf "B" with the same intersection cost (right side). Note however that this argument does not allow for combining leaves "K" and "L". Also note in practive the impact of this technique is much less pronounced than in this simple example, as the relative *impact of removing some leaf nodes is much less in a deeper BSP.*

section code with enough suitable intersection operations, it is yet unclear how the 1:4 approach could be generalized to the *traversal* of either BSP trees, kd-trees, octrees, or grids: Successive traversal steps of one ray usually depend on each other, and thus can not be performed in parallel. However, as a ray tracer usually performs many more traversal steps than intersection operations (see Table 7.5), accelerating the ray-triangle intersections alone can not be expected to yield a significant speedup.

Note however that the 1:4 approach can still be beneficial when optimizing the BSP tree with the 1:4 intersection in mind (see Figure 6.3). A fast 1:4 intersection routine allows for larger leaves in the BSP tree, resulting in a shallower and less complex BSP tree, which in turn leads to fewer traversal operations and higher performance. Experiments with this approach have shown some speedups over the pure C code single ray traversal. These speedup however are usually rather small, and thus are not investigated any closer in the remainder of this thesis.

6.4.4.2 Alternative 2: Tracing Packets of Rays (4:1)

Compared to the 1:4-approach, the alternative approach of intersecting four rays with the same triangle is much more efficient. At first sight, the 4:1 approach seems infeasible due to the need to always have four rays available for intersection. This is usually not the case in ray tracing, as rays are traditionally traced recursively and independently of each other. Furthermore, different rays typically operate on different data, which conflicts with the goal of handling them together on the same data item.

Fortunately, though randomly generated rays certainly operate on different data, in practice we find surprisingly large coherence between rays: Most of the rays are actually very similar to nearby rays, and perform similar operations[27]. For example, neighboring primary rays most often traverse the same regions of space, visit the same BSP nodes, and intersect the same triangles. Though the nodes and triangles visited by two such rays may from time to time vary, the majority of operations will be the same[28]. As such, coherent rays can be easily and efficiently traversed and intersected together in packets of four. This obviously requires algorithms that have been especially designed to traverse, intersect, and shade *packets of rays* in parallel, which will be explained in the following chapter.

Efficiency: Obviously, the success of this approach depends to a large degree on the amount of coherence between rays. Fortunately however coherence is not only high for primary rays, but is also largely present for

[27] At high resolutions, many primary rays actually intersect exactly the same triangles as those in neighboring pixels.

[28] Even if rays diverge at the leaves of the BSP tree, they can still be traversed together through the upper levels of the tree

secondary and especially for shadow rays: Coherent rays usually hit similar regions in space, from which they form – when connected to the same light source – packets of coherent shadow rays [Wald01a, Benthin03, Schmittler02] (see e.g. Section 13.3). Though the coherence for shadow rays in practice is slightly less than for primary rays, coherence is still very high (see e.g. [Schmittler02] for an investigation of packet traversal overhead for varying packet sizes). Even secondary rays resulting from reflection and refraction often offer a significant amount of coherence. While it is relatively easy to construct worst-case examples, the approach so far has shown to work quite well in practice.

Interestingly, the approach benefits from the increasing trend towards higher image resolutions[29]: At higher resolution, the same view is sampled with a higher density, which translates to increased coherence of the primary rays. If sampled densely, even reflection rays off a highly curved object show significant coherence.

6.4.5 Shading

Once we have an efficient means of traversing and intersecting packets of rays using SIMD extensions, we still have to shade the resulting hit points. Obviously, a simple solution is to split the ray packet up into four individual rays, and to feed them to their respective shaders one at a time. Though this approach obviously works, it has two drawbacks: First, this is a quite costly process, as it requires data reorganization and does not allow for a data-parallel SIMD approach during shading. Second, after having split up a packet of primary rays, each shader only operates on a single ray, and cannot easily combine them with secondary rays cast by other shaders in order to form coherent ray packets also for secondary and shadow rays. Without having coherent ray packets for these secondary rays however we cannot use the fast packet traversal code for these rays, either.

Instead of splitting the packets up, we can also follow the same approach as during traversal, and just shade four rays in parallel. This is especially promising for the generation of secondary ray packets: Such packets can best be generated by processing all the four original rays in parallel, e.g. by connecting them to the same light source. This simple and efficient way of generating secondary ray packets is no longer possible once the rays have been split up.

On the other hand, this data-parallel approach to shading is problematic, too: A data-parallel SIMD approach only works efficiently if all four rays

[29] Obviously the same argument holds for pixel-supersampling

perform *exactly* the same computations (including their data accesses). For shading, however, this is often not the case, as different rays can have hit triangles with different shaders. Roughly speaking, shading only sets in at the hitpoints, where the rays are furthest apart and most incoherent. As such, shading is much less likely of performing the same operations and accessing the same data for each ray in the packet, making it much less suited for a data-parallel SIMD approach than traversal and intersection.

Note that this does not contradict our previous discussion on coherency during traversal and intersection: Even rays that have traversed exactly the same nodes, have visited exactly the same voxels, and have intersected exactly the same triangles may easily end up having different intersections, and thus different shaders (see Figure 6.4). As different shaders can perform different operations, it is not possible to execute four different shaders in a data-parallel approach.

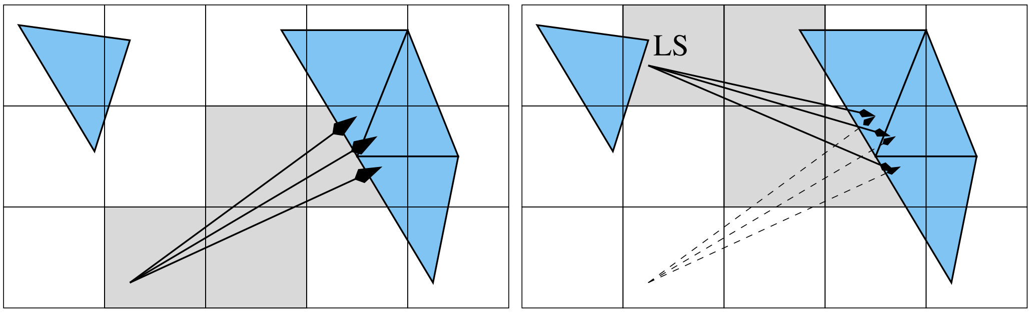


Figure 6.4: Coherence is worst during shading Left: Even rays that perform exactly the same operations during traversal and intersection may easily end up on different triangle with different shaders (though this drawing depicts a regular grid for ease of illustration, exactly the same problem happens in BSP trees). Even if the different triangles have the same shader, they still require access to different data (e.g. vertex normals or texture coordinates). Note that coherence during shading is even worse than for secondary ray (right): For example, connecting the four hit points on different triangles to a point light source (LS) can once again generate four coherent rays that traverse the same voxels!

If we can guarantee that all rays are shaded by exactly the same shader[30], however, hitpoints on different triangles require the individual rays to operate on different data. As this data is stored in different locations, it has to be

[30] Actually, it does not have to be exactly the shader: It is sufficient if the four shader instances belong to the same shader class, and thus execute the same shader program (with different parameters).

copied and rearranged to a SIMD-friendly manner (see above). This process alone is usually quite costly.

Summary

In summary, the efficient shading of packets of coherent rays in a streaming manner is still an unsolved problem and requires closer investigation. In first experiments, a restricted data-parallel packet shading approach has shown to work well for fixed lighting models [Benthin03], in which we can guarantee that the scene contains only one kind of shader. For a less restricted setting of having a mix of different, arbitrarily programmable shaders however it is yet unclear how such packet shading would work. Especially the *efficient* generation of coherent packets of secondary rays cast by different shaders is an open question[31].

As there is no single, final answer on how to best shade packets of rays, the RTRT/OpenRT engine currently supports both of the previously mentioned approaches: For those restricted cases in which the lighting model is fixed for all rays, we use specialized packet shaders that work in a data-parallel manner. Though a fixed lighting model indeed is a severe restriction, it still allows for even as complex tasks as interactive global illumination (see Section 13.3) or the fast visualization of highly complex models via ray casting.

For more sophisticated applications that require a mix of different shaders, we split the rays up and pass them to arbitrarily programmable shaders that are dynamically loaded from shared libraries. Though this obviously results in a severe performance penalty (also see Table 7.6 in the following Chapter), the flexibility of such shader plug-ins is essential for making a ray tracing engine the general tool that enables all the applications discussed in later sections, so this penalty is currently unavoidable.

[31] Obviously, it would be possible to reorder rays into coherent packets by following the same approach as outlined in [Pharr97]. This approach however would most likely be too costly for interactive use.

Chapter 7

The RTRT Core – Triangle Intersection and BSP Traversal

> *"In theory, there is no difference between theory and practice. But, in practice, there is."*
>
> *Jan L.A. van de Snepscheut*

In the previous chapter, we have discussed the overall design issues of the RTRT system. To summarize the most important points, we have chosen to only support triangles, to exploit SIMD extensions in a data-parallel way, to optimize for memory and caches, and to use BSP trees as an acceleration structure. In this chapter, we are now going to discuss the actual algorithms and implementation of these topics in more detail.

7.1 Fast Triangle Intersection in RTRT

Fast ray triangle intersection code has long been an active field of research in computer graphics and has lead to a large variety of algorithms, e.g. Moeller-Trumbore [Möller97, Möller], Glassner [Glassner89], Badouel [Badouel92], Pluecker [Erickson97, Shoemake98], and many others. The RTRT core uses a modified version of the projection method (see below), which has been specially designed to run as fast as possible with single-ray C code, while still being well suited for SSE code.

Essentially, the task of computing a ray-triangle intersection can be described as follows: Given a ray $R(t) = O + tD; t \in (0, t_{max})$[1] (going from its

[1] In practice, rays usually start at $t_{min} = \epsilon$ in order to avoid "self-intersection", see Section 2.1

origin O into direction D), and a triangle with vertices A, B and C, determine whether the ray has a valid hit-point $H = R(t_{hit})$ with the triangle, i.e. whether there exists a t_{hit} with $t_{min} \leq t_{hit} \leq t_{max}$ and $R(t_{hit})$ is inside the triangle.

In case of having found a valid hit point, many ray tracers require that the ray-triangle intersection routine also returns the barycentric coordinates (or local surface coordinates) of the hit-point for shading purposes. As these coordinates are often computed anyway in the process of determining the hit-point, we follow this pattern. Note, however, that this is not the case for shadow rays, for which only the boolean yes/no decision is important, and which can be slightly optimized by not storing these coordinates.

7.1.1 Barycentric Coordinate Tests

While there are many different methods for computing ray-triangle intersections, many of them are based on computing the barycentric coordinates of the hitpoint and using those for determining whether there is a valid intersection or not[2] (e.g. [Badouel92, Shirley03, Glassner89]). In fact, most ray-triangle intersection algorithms (including the one proposed here) follow this general pattern, and are often only variants and different implementations of the same idea.

In order to use barycentric coordinates for computing ray triangle intersections, one fist computes the signed distance t_{plane} along the ray to the plane embedding the triangle. Given the geometric normal $N = (B-A) \times (C-A)$ and a triangle vertex A, this can be computed as $t_{plane} = -\frac{(O-A).N}{D.N}$. The calculated distance t_{plane} is then tested for whether it lies in the interval in which the ray is actually looking for intersections. If not, no valid intersection can occur, and the triangle test returns "no intersection". The triangle normal N is often computed "on the fly". This minimizes storage requirements, but requires a costly vector product.

If this so-called *distance test* has been passed, one has to check whether the ray actually pierces the triangle. To do this, the actual intersection point with the plane is computed as $H = R(t_{plane}) = O + t_{plane}D$, and is then tested whether it actually lies inside the triangle. The barycentric coordinates of H can then be computed in several ways, e.g. by solving the system of equations $H = \alpha A + \beta B + \gamma C$, or geometrically by considering the relative *signed* (!) areas of the triangles ABC, HBC, AHC and ABH.

Once the barycentric coordinates α, β and γ of H are known, one can

[2] The barycentric coordinates of H are the values α, β and γ for which $\alpha A + \beta B + \gamma C = H, \alpha + \beta + \gamma = 1$. If H is inside the triangle, both α,β and γ are positive.

determine whether H is inside the triangle by and checking whether the conditions

$$0 \leq \alpha \leq 1, \quad 0 \leq \beta \leq 1, \quad 0 \leq \gamma \leq 1$$

are fulfilled. Note that it is sufficient to check whether $\beta \geq 0$, $\gamma \geq 0$ and $\beta + \gamma \leq 1$, which follows from the properties of barycentric coordinates ($\alpha + \beta + \gamma = 1$).

7.1.2 Projection Method

The projection method is an optimization of the barycentric coordinate test. It exploits the fact that projecting both triangle ABC and hit-point H into any other plane (except for the planes that are orthogonal to the plane ABC) does not change the barycentric coordinates of H. The computations for calculating the barycentric coordinates can then be optimized by projecting both triangle and hit-point H into one of the 2D coordinate planes (XY-, XZ- or YZ-plane), in which all further computations can be performed in 2D. For reasons of numerical stability, one should project into the plane in which the triangle has maximum projected area. This so-called "projection dimension" corresponds to the dimension in which the normal N has its maximum absolute component.

After projection, all computations can be performed more efficiently in 2D. For example, projecting into the XY plane (i.e. projection dimension is 'Z') yields

$$H' = \alpha A' + \beta B' + \gamma C',$$

where A', B', C' and H' are the projected points of A, B, C, and H, respectively. Substituting $\alpha = 1 - \beta - \gamma$ and rearranging the terms yields

$$\beta(B' - A') + \gamma(C' - A') = H' - A'.$$

This can be solved (e.g. using the Horner scheme), yielding $\beta = \frac{\det|bh|}{\det|bc|}, \gamma = \frac{\det|hc|}{\det|bc|}$, (where $b = C' - A'$, $c = B' - A'$ and $h = H' - A'$). In 2D, this can be expressed quite efficiently as

$$\beta = \frac{b_x h_y - b_y h_x}{b_x c_y - b_y c_x}, \gamma = \frac{h_x c_y - h_y c_x}{b_x c_y - b_y c_x}. \tag{7.1}$$

In pseudo-code, the projection method usually looks like the following:

```
// calc edges and normal
b = C-A; c = B-A; N = Cross(c,b);
```

```
// distance test
t_plane = - Dot((O-A),N) / Dot(D,N);
if (t_plane < Epsilon || t_plane > t_max) return NO_HIT;

// determine projection dimensiondimensions
if (|N.x| > |N.y|)
  if (|N.x| > |N.z|) k = 0; /* X */ else k=2; /* Z */
else
  if (|N.y| > |N.z|) k = 1; /* Y */ else k=2; /* Z */
u = (k+1) mod 3; v = (k+2) mod 3;

// calc hitpoint
H[u] = O[u] + t_plane * D[u];
H[v] = O[v] + t_plane * D[v];

beta  = (b[u] * H[v] - b[v] * H[u]) / (b[u] * c[v] - b[v] * c[u]);
if (beta < 0) return NO_HIT;

gamma = (c[v] * H[u] - c[u] * H[v]) / (b[u] * c[v] - b[v] * c[u]);
if (gamma < 0) return NO_HIT;

if (beta+gamma > 1) return NO_HIT;

return HIT(t_plane,beta,gamma);
```

7.1.3 Optimizing the Projection Method

Taking a closer look at the execution pattern of the above mentioned projection method, it becomes obvious that for different executions on the same triangle many values will be recomputed every time: For example, the edges and normal of a triangle will be recomputed for every intersection test with this triangle, and also the result of determining the projection case will always remain the same. These - and other - computations are thus redundant, and can be saved by precomputing and storing them. This saves the costly computations for the normal, and enables to avoid the branches for determining the projection case. Once the normal is known, the two secondary dimensions ($u = (k+1) mod\, 3$ and $v = (k+2) mod\, 3$) can then be determined by a simple table lookup ($intmodulo[5] = \{0, 1, 2, 0, 1\}$), without having to perform the two expensive modulo operations.

Note that we do not have to store the full normal: If k is the projection dimension, $N.k$ can never be zero. As such, we can divide the normal N by $N.k$, yielding $N' = \frac{N}{N.k}$. Then $t = \frac{(A-O).N'}{D.N'} = \frac{A.N' - O_u.N'_u - O_v.N'_v - O_k.N'_k}{D_u.N'_u + D_v.N'_v - D_k.N'_k}$.

Obviously the values $d = A.N'$, $N'_u = \frac{N_u}{N_k}$ and $N'_v = \frac{N_v}{N_k}$ are constant for each triangle and thus can be precomputed. By definition, N'_k is equal to one, and thus doesn't have to be stored. Furthermore, knowing that $N'_k = 1$ saves two additional multiplications.

The same idea – simplifying the computations and precomputing as many of the terms as possible – can also be applied to the edges: Rearranging the terms for computing β and γ yields

$$\begin{aligned}\beta &= \frac{1}{b_x c_y - b_y c_x}(b_x H_y - b_x A_y - b_y H_x + b_y A_x) \\ &= \frac{b_x}{b_x c_y - b_y c_x} H_y + \frac{-b_y}{b_x c_y - b_y c_x} H_x + \frac{b_y A_x - b_x A_y}{b_x c_y - b_y c_x} \\ &= K_{\beta y} H_y + K_{\beta x} H_x + K_{\beta d}.\end{aligned}$$

This equation now depends only on the projected coordinates H_x and H_y of the hit-point H (which can be calculated entirely from N', O and D). After precomputing and storing the constants $K_{\beta y}$, $K_{\beta x}$, and $K_{\beta,d}$, $\beta = K_{b,nu} H_x + K_{b,nv} H_y + K_{b,d}$[3] can be computed quite efficiently. Note that no other values have to be stored for computing β. Obviously, the same procedure works for the second barycentric coordinate, γ. The last one, α then does not require any further storage space, as $\alpha = 1 - \beta - \gamma$.

With these simplifications and precomputations, only very few operations have to be performed during runtime. In the worst case[4], only 10 multiplies, 1 division, and 11 additions are needed for an intersection. If the ray fails already at the distance test, only 4 muls, 5 adds, and 1 division are needed. Neither geometric normal nor the edge vectors have to be stored or computed during intersection.

7.1.4 Cache-optimized Data Layout

Obviously, preprocessing can save quite some amount of computations. However, as mentioned above this has to be done quite carefully: Due to the high cost of a cache miss, using additional memory for storing precomputed values carries the chance of actually costing more than the operation itself. On

[3] It is interesting to note that the same three values can also be derived and explained geometrically. In that case, $K_{b,nu}, K_{b,nv}$ and $K_{b,d}$ correspond to the line equation $L_b(u, v) = K_{b,nu}.u + K_{b,nv}.v + K_{b,d} = 1$ of side $b = C' - A'$ (hence the name of the constants), properly scaled such that inserting the third vertex B' into the line equation yields $L_b(B'_x, B'_y) = 0$.

[4] Note that with a good BSP tree, this worst-case cost (a valid intersection) happens quite frequently, as a good BSP tree already avoids most unsuccessful intersection operations, see Table 7.5.

the other hand, careful data layout can even simplify the memory access patterns, and can help in prefetching and in reducing cache misses. Using the just mentioned simplifications, all data needed for a triangle intersection can be expressed in only 10 values: 3 floats (d, N'_u, N'_v) for the scaled plane equation, 3 floats each for the two 2D line equations in the u/v plane, and one int (actually only 2 bits) for storing the projection case k.

Note that these 10 values comprise *all* the data required for the triangle test. In fact, with these precomputed values it is not even necessary any more to know the actual vertex positions of the triangle. Though these are still stored somewhere for potential shading purposes (resulting in an actual *increase* in total memory consumption), they do not have to be accessed at all during traversal and intersection.

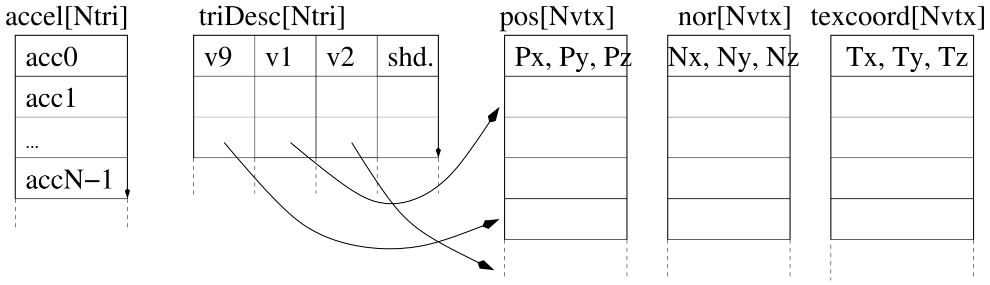


Figure 7.1: The RTRT core organizes its geometry in the typical "Vertex Array" organization (also called "Indexed Face Sets" in VRML97 terms [Carey97]): Vertices are stored in arrays from where they are referenced by triangles. Each triangle is described by pointers (or IDs in our case) to its three vertices, plus an ID for specifying its shader (also see Figure 10.1 to see how this fits into the full OpenRT system.). The different vertex attributes (e.g. position, normal, texture coordinates etc) are stored in separate lists, thereby allowing to store only those data that have actually be specified by the application. Additionally to this typical data layout, the RTRT core keeps a separate acceleration record for each triangle that stores all data required for an intersection in a preprocessed form. Thus, neither *ID record nor vertex data is ever touched during traversal and intersection. Whereas typical intersection algorithm require to fetch data from four different, non-cache-aligned memory locations (thereby having to chase the pointers in the ID record), RTRT fetches only this single acceleration data, which lends well to caching and prefetching.*

Since we know the access pattern of the intersection algorithm, we can even store the 10 values in the order in which they are accessed by the CPU to enable even better data access for the CPU. This leads to a very simple data layout for our triangle acceleration structure:

```
struct TriAccel
{
  // first 16 byte half cache line
  // plane:
  float n_u; //!< == normal.u / normal.k
  float n_v; //!< == normal.v / normal.k
  float n_d; //!< constant of plane equation
  int k;    // projection dimension

  // second 16 byte half cache line
  // line equation for line ac
  float b_nu;
  float b_nv;
  float b_d;
  int pad; // pad to next cache line

  // third 16 byte half cache line
  // line equation for line ab
  float c_nu;
  float c_nv;
  float c_d;
  int pad; // pad to 48 bytes for cache alignment purposes
};
```

Though this data layout actually uses *more* memory than other intersection algorithms operating directly on the vertices (like e.g. Moeller-Trumbore [Möller97]), it is likely to use the cache better (see Figure 7.1): Operating directly on the vertices requires to first access a record that contains the vertex IDs, which require to access at least one cache line. Then accessing the vertices themselves again requires to touch three cache lines, except if the vertices are incidentally stored next to each other. If the index record and/or the vertices straddle cache line boundaries, another four cache lines might be required. In contrast to these up to 8 cache lines, the above structure can be guaranteed to use exactly two cache lines on 32 byte caches, and often only one cache access for 64 byte or 128 byte caches[5].

Furthermore, having all data for the intersection test in one contiguous block also allows for efficient prefetching. Having reached a leaf, prefetching the next triangle before intersecting the current one can guarantee that the next triangle is already in the cache until needed. Finally, having all required

[5] Intel Pentium-III processors have 32 byte cache lines, whereas AMD Athlon-MPs have 64 bytes, and Intel Pentium IV Xeons have 128 bytes per cache line.

data values stored sequentially one after another ideally lends to a streaming-like SIMD implementation.

However, the additional memory overhead can be problematic for extremely complex scenes for which both main memory and address space become quite a limiting factor. For these special cases, the RTRT kernel also contains an efficient implementation of the Moeller-Trumbore algorithm [Möller97] (in both a single-ray C-code as well as in an SSE implementation), which can be used for these cases.

7.1.5 C Code Implementation for Single-Ray/Triangle Intersection

Writing the code for the just derived intersection algorithm is straightforward, and can be expressed in only a few lines of code:

```
// lookup table for the modulo operation
ALIGN(ALIGN_CACHELINE) static const
unsigned int modulo[] = {0,1,2,0,1};

inline void Intersect(TriAccel &acc,Ray &ray, Hit &hit)
{
#define ku modulo[acc.k+1]
#define ku modulo[acc.k+2]
  // don’t prefetch here, assume data has already been prefetched

  // start high-latency division as early as possible
  const float nd = 1./(ray.dir[acc.k]
     + acc.n_u * ray.dir[ku] + acc.n_v * ray.dir[kv]);
  const float f = (acc.n_d - ray.org[acc.k]
     - acc.n_u * ray.org[ku] - acc.n_v * ray.org[kv]) * nd;

  // check for valid distance.
  if (!(hit.dist > f && f >  EPSILON  )) return;

  // compute hitpoint positions on uv plane
  const float hu = (ray.org[ku] + f * ray.dir[ku]);
  const float hv = (ray.org[kv] + f * ray.dir[kv]);

  // check first barycentric coordinate
  const float lambda = (hu * acc.b_nu + hv * acc.b_nv + acc.b_d);
  if (lambda < 0.0f) return;

  // check second barycentric coordinate
  const float mue = (hu * acc.c_nu + hv * acc.c_nv + acc.c_d);
  if (mue < 0.0f) return;

  // check third barycentric coordinate
  if (lambda+mue > 1.0f) return;

  // have a valid hitpoint here. store it.
  hit.dist = f;
  hit.tri = triNum;
  hit.u = lambda;
  hit.v = mue;
}
```

Note that the costly “modulo 3” operation has been replaced with a precomputed lookup table. The most costly operation in this triangle test is the division at the beginning, which in SSE code can be replaced by a faster reciprocal operation with Newton-Raphson iteration (see e.g. [Intel, AMDb]).

Also note that the actual implementation uses a C code “macro” for the intersection code, which (surprisingly) is even faster than an “inline” function as shown above. Instead of the many memory indirections into the origin and direction vectors it is also possible to do a switch-case statement based on *acc.k* at the beginning, and then use hard-coded offset values. The speed difference between these two implementations is small. Depending on the actual CPU used (i.e. Athlon vs. Pentium-III vs. Pentium-IV), sometimes one versions is faster, and sometimes the other.

7.1.5.1 Single-Ray Intersection Performance

The performance of this optimized implementation is given in Table 7.1, in which the single-ray C Code version of this triangle test is compared to a fairly optimized implementation of the standard Moeller-Trumbore triangle test [Möller97]. As can be seen, our proposed triangle test in practice is roughly twice as fast as the Moeller-Trumbore code.

7.1.6 SSE Implementation

By design, the chosen algorithm and data layout naturally lend to SSE implementation. In fact, for our SSE triangle intersection we use exactly the same code and data structures as described above in the previous Section. The only major change is that instead of a single ray, we use a structure that stores four rays together in a SIMD-friendly way (see Figure 7.2): Opposed

CPU Cycles	MT	OP	speedup
primary rays	144–172	69–74	2.1–2.3
shadow rays	127–144	68–73	1.9–2.0

Table 7.1: Performance for the RTRT optimized projection (OP) triangle test algorithm as compared to the Moeller-Trumbore algorithm (MT) [Möller97], measured in CPU cycles on a single 2.5 GHz Pentium-IV notebook. The RTRT code is measured with the single ray C code implementation, not *with the fast SIMD code described in Section 7.1.6. Note that these measurements have not been taken with synthetical ray distributions, but correspond to average case performance in typical scenes. The actual cost depends on the probability with which a ray exits at a certain test (e.g. distance test, any of the barycentric coordinate tests, or successful intersection) and as such varies from one scene to another, and also differs for shadow and 'standard' rays (i.e. primary and secondary rays). For the RTRT OP triangle test, these numbers correspond to more than 35 million ray-triangle intersections. Also note that a 2.5GHz notebook CPU is not state of the art any more.*

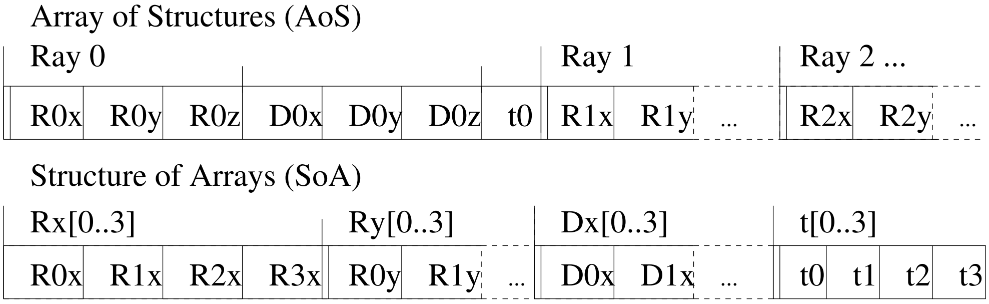


Figure 7.2: Array-of-structures (AoS) vs. structure-of-arrays (SoA) layout for our ray packets. Each ray consists of origin (R) and direction (D) vectors, as well as its maximum length (t). The same data layout has to be used for the hit point information. While the AoS layout is more natural, efficient SIMD code requires the reorganization to the SIMD-friendly SoA layout. In order to achieve sufficient performance, this layout has to be used during all computations, i.e. already during ray generation.

to the standard way of storing such four rays as an array of four ray structures (the "AoS" organization), accessing such values efficiently with SSE requires to reorganize such data into a "SoA" (structure of arrays) organization, i.e. first storing the four origin.x values, then the four origin.y values, etc.

Using SSE intrinsics, implementing the above algorithm in SSE is almost straightforward. For example, the line

```
  const float hu = (ray.org[ku] + f * ray.dir[ku]);
```

can easily be expressed as

```
  const sse_t hu = _mm_add_ps(ray4.org[ku],
                              _mm_mul_ps(f,ray4.dir[ku])).
```

Though converting the whole algorithm in that way is quite simple, the actual code is quite lengthy due to the low-level nature of the SSE operations, and as such is omitted here.

7.1.6.1 Overhead

A potential source of overhead is that even though some rays may have terminated early, all four rays have to be intersected with a triangle. For coherent rays however this is unlikely. However, not all rays may have found a valid hit, so the hit information may only be updated for rays that actually found an intersection. To achieve this, information on which of the four rays is still active is kept in a bit-field, which can be used to mask out invalid rays in a conditional move instruction when storing the hit point information.

Though this is simple to implement, it results in a considerable overhead, see Table 7.2: Whereas both shadow rays and 'standard' rays undergo *exactly* the same floating point computations until the hit/no hit information has been determined, standard rays require several masking operations in order to update the hit information only for those rays that have actually had a valid intersection. Shadow rays have to perform significantly less of these masking operations, as only a single flag has to be stored per ray, in contrast to triangle and instance ID, distance, and barycentric coordinates for normal rays.

7.1.6.2 Performance Results

The overall results of our fast ray-triangle intersection code can be seen in Table 7.2: Whereas the C Code is already much faster than the Moeller-Trumbore Test (see Table 7.1), the SSE code achieves an additional, significant speedup: On a 2.5 GHz Pentium-IV CPU, the SSE code for intersecting four rays with a single triangle requires 101–107 CPU cycles, depending on where the code exits. Amortizing this cost over all four rays results in only 25–27 cycles per intersection. Compared to the C code implementation of the RTRT OP algorithm, this results in a speedup of 2.7–2.8. Compared

CPU Cycles	C Code	SSE 4:1	SSE 4:1	speedup	rays per
	single ray	per packet	per ray		second
primary rays	69–74	101–107	25–27	2.70–2.76	92M–100M
shadow rays	68–73	80–93	20–23	3.17–3.4	108M–125M

Table 7.2: Cost (in CPU cycles) for our optimized ray-triangle test in a single ray C code implementation and in its data parallel 4:1 SSE implementation. As in Table 7.1, these numbers correspond to average-case performance in typical scenes. On a 2.5 GHz Pentium IV CPU, 20–27 cycles correspond to 108–125 million ray-triangle intersections per second. Note that the "speedup" is only calculated with respect to the single-ray C code implementation. Comparison to a C code Moeller-Trumbore implementation (see Table 7.1 would yield a speedup of more than 6.

to the C code Moeller-Trumbore implementation in Table 7.1, a speedup of more than six can be observed.

As discussed above, shadow rays have significantly less overhead for storing the hit information, and as such are much faster: A shadow ray intersection costs only 80–93 cycles per packet, respectively 20–23 per ray. This once again shows that SSE is extremely efficient for speeding up computations (the actual computations for shadow rays and primary rays are the same), but quickly suffers from any non-computation overhead.

All these measurements have been performed on a 2.5GHz Pentium-IV notebook CPU, on which these numbers correspond to 92–100 million ray triangle intersections for standard rays, and even 108–125 million intersections per second for shadow rays. The overall speedup compared to the single ray C code implementation is around 2.7 for primary and secondary rays, and 3.1–3.4 for shadow rays. This difference clearly shows the impact of the above-discussed overhead for updating the hit information for non shadow rays.

Note that this masking overhead for storing the results might be partially hidden if more than four rays would be intersected in parallel. Generally, operating on larger packet sizes would allow for a more streaming-like approach, in which the latencies of certain operations could be hidden much better. Also note that the application of this data-parallel intersection algorithm is not limited to the RTRT core, but could also be used to accelerate other ray tracing-based rendering algorithms such as memory coherent ray tracing [Pharr97].

7.2 Fast kd-Tree Traversal

Even before accelerating the triangle test, traversal of the acceleration structure was typically 2-3 times as costly as ray-triangle intersection, as a ray tracer typically performs many more traversal steps than triangle intersections (see Table 7.5 for statistical traversal data in different scenes[6]). Once the SSE triangle intersection code reduces the intersection cost by more than a factor of three, traversal is the limiting factor in our ray tracing engine. Furthermore, the SSE intersection procedure requires us to always have four rays available anyway. Therefore, we need an algorithm for efficiently traversing four rays through an acceleration structure in parallel.

As already discussed in Section 3.3.2 a wide variety of ray tracing acceleration schemes have been developed over the last two decades. For example, there are octrees, general BSP-trees, axis-aligned BSP-trees, uniform, non-uniform and hierarchical grids, ray classification, bounding volume hierarchies, and several hybrids of several of these methods[7]. As already discussed in Section 6.4.1, we have chosen to use axis-aligned BSP trees (kd-trees) for the RTRT core. Their traversal code is quite simple, and can very well be implemented in a highly optimized form. Furthermore, BSP trees usually perform at least comparable to other techniques [Havran00, Havran01], and are well-known for their robustness and applicability for a wide range of scenes. However, our main reason for using a BSP tree in the RTRT core is the simplicity of the traversal code, which allows for efficiently traversing packets of rays in parallel: Traversing a node is based on only two binary decisions, one for each child, which can efficiently be done for several rays in parallel using SSE. If any ray needs traversal of a child, all rays will traverse it in parallel.

This is in contrast to algorithms like octrees or hierarchical grids, where each of the rays might take a different decision of which voxel to traverse next. Keeping track of these states is non-trivial and was judged to be too complicated to be implemented efficiently. Bounding Volume Hierarchies have a traversal algorithm that comes close in simplicity to BSP trees, and could also be adapted to a SIMD-traversal method. However, BVHs do not partition *space*, but rather organize the hierarchy. This leads to different parts of the hierarchy overlapping themselves, does not allow for efficiently traversing the voxels in front-to-back order[8], and thus in practice makes

[6] Similar data hold for different acceleration structures, see [Havran01].

[7] See Section 3.3.2 for an overview and further references.

[8] It is possible to traverse BVHs in front-to-back order by keeping the yet-to-be-traversed parts of the hierarchy organized in a priority queue [Haines91a]. This however makes each traversal step considerably more costly than a BSP traversal step

BVHs inefficient for complex scenes (for extensive statistical experiments, see [Havran00, Havran01]). Furthermore, algorithms for *building* BVHs that are well-suited for fast traversal are less well investigated than similar algorithms for BSP trees.

7.2.1 Data Layout of a kd-Tree Node

As mentioned above, the ratio of computation to the amount of accessed memory is very low for scene traversal. This requires us to carefully design the data structure for efficient caching and prefetching.

For a typical BSP node, one has to store

- A flag specifying whether it is a leaf node or an inner node.
- For leaf nodes, an "item list", i.e. a list of integer IDs that specify the triangles in this leaf; consists of a pointer (or index) to the first item in the list, and of the number of items in the list.
- For inner nodes, the addresses of the two children, the dimension of the splitting axis (i.e. x, y, or z), and the location of the split plane.

All these values can be stored in a very compact, unified node layout of only 8 bytes: Obviously, a node can either be a leaf node or an inner node, so they can be stored in the same memory location (a `union` in C code) as long as there is at least one bit reserved for determining the kind of the node.

For inner nodes, we need half the node for storing the float value that specifies the split plane. Addressing the two children can be performed with a single pointer if children of a node are always stored next to each other. Furthermore, if all BSP nodes are stored in one contiguous array (with child nodes always stored after their parent nodes), this single pointer can be expressed as an offset relative to the current node. As this offset is positive, we can use its sign bit for storing the flag that specifies the type of node. Finally, having the nodes stored in an array guarantees that the offset is a multiple of 8 (the node size), so its lower two bits can be safely used for storing the splitting axis.

Leaf nodes can be expressed in quite the same way: The flag that specifies the node type has to remain in place, and the pointer to the start of the item list is stored just like the children pointer, as a relative offset stored in bits 2..30.

This leads to the following simple, compact structure:

```
struct BSPLeaf {
  unsigned int flagDimAndOffset;
  // bits 0..1      : splitting dimension
  // bits 2..30     : offset bits
  // bit  31 (sign) : flag whether node is a leaf
  float splitCoordinate;
};
struct BSPInner {
  unsigned int flagAndOffset;
  // bits 0..30     : offset to first son
  // bit  31 (sign) : flat whether node is a leaf
}
typedef union {
  BSPLeaf  leaf;
  BSPInner inner;
} BSPNode;
```

Note that the exact order and arrangement of the bits has been *very* carefully designed: Each value can be extracted by *exactly* one "bitwise and" operation to mask out the other bits, and does *not* require any costly shift operations for shifting bits to their correct positions.

```
#define ABSP_ISLEAF(n)    (n->flag_k_ofs & (unsigned int)(1<<31))
#define ABSP_DIMENSION(n) (n->flag_k_ofs & 0x3)
#define ABSP_OFFSET(n)    (n->flag_k_ofs & (0x7FFFFFFC))
```

As traversing a BSP node is by far the most common operation in a ray tracer, it has to be implemented with extreme care. For example, an older version of the RTRT kernel originally stored the dimension bits in the *upper* bits of the flag word, from where they could be retrieved by a single shift operation. While this seems comparably cheap, due to this single shift operation (which is quite more costly than a "bitwise and") the old version was roughly 5 percent slower than the current version.

The presented data layout allows for squeezing the whole BSP node description into 8 bytes per node, or 4–16 nodes per cache line[9]. As we always store both children of a node next to each other, both nodes are stored in the same cache line[10], and are thus always and automatically fetched together.

[9] Assuming 32 bytes per cache line on a PentiumPro Architecture (Pentium-III), 64 bytes on an AMD Athlon MP, and 128 bytes on an Intel Pentium-IV Xeon. Note that the larger cache sizes on a Xeon CPU might benefit from an improved node packing inside a cache line as discussed in [Havran97, Havran99, Havran01]

[10] In RTRT, all BSP node pairs are aligned to cache line boundaries: All nodes are stored in one consecutive, cache-aligned array, and the cache line size is a multiple of the node size.

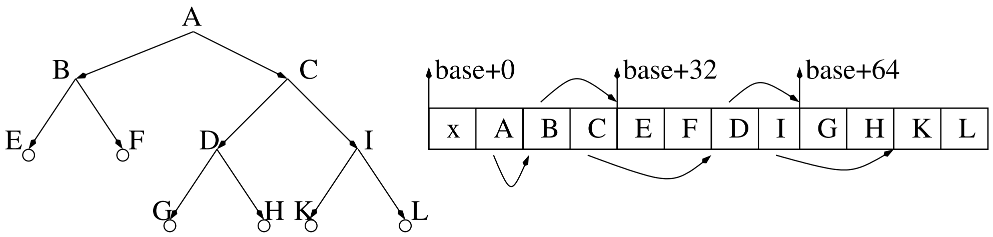


Figure 7.3: All BSP nodes (inner nodes as well as leaf nodes) in RTRT are stored in one contiguous, cache-aligned array. Depending on cache line size, either 4, 8, or 16 nodes form one cache line. Both children of the same node are always stored next to each other, and thus land in the same cache line. As cache line size is a multiple of node size, node pairs will never overlap a cache line boundary. Both children can be addressed by the same pointer, which is stored as an offset. As this offset is always positive and divisible by four, we can squeeze both node type flag (leaf or inner node) and split dimension (X, Y, or Z) in the sign bit and in the lower two bits, respectively. For leaves, pointers to the item lists (not shown) are stored exactly like pointers to nodes.

Using the same pointer for both node types allows for reducing memory latencies and pipeline stalls by prefetching, as the next data (either a node or the list of triangles) can be prefetched before even processing the current node. Note that though prefetching requires SSE cache control operations, prefetching is also possible for the single-ray, non-SIMD traversal code. Similarly, the benefits of using this optimized node layout, i.e. reduced bandwidth and improved cache utilization, positively affect both the C-code as well as the SSE implementation.

7.2.2 Fast Single-Ray kd-Tree Traversal

Before describing our algorithm for traversal of four rays in parallel, we first take a look at the traversal of a single ray: In each traversal step, we maintain the *current ray segment* $[t_{near}, t_{far}]$, which is the parameter interval of the ray that actually intersects the current voxel. This ray segment is first initialized to $[0, \infty)$[11], then clipped to the bounding box of the scene, and is updated incrementally during traversal[12]. For each traversed node, we calculate the distance d to the splitting plane defined by that node, and compare that

[11] In practice, rather to $[\epsilon, t_{max}]$

[12] Instead of clipping to the scene bounding box, it is also possible to not clip at all and rather use six additional BSP planes that represent the bounding box sides. This is typically slower in a software implementation, but can be useful for hardware implementations such as in the SaarCOR architecture

distance to the current ray segment.

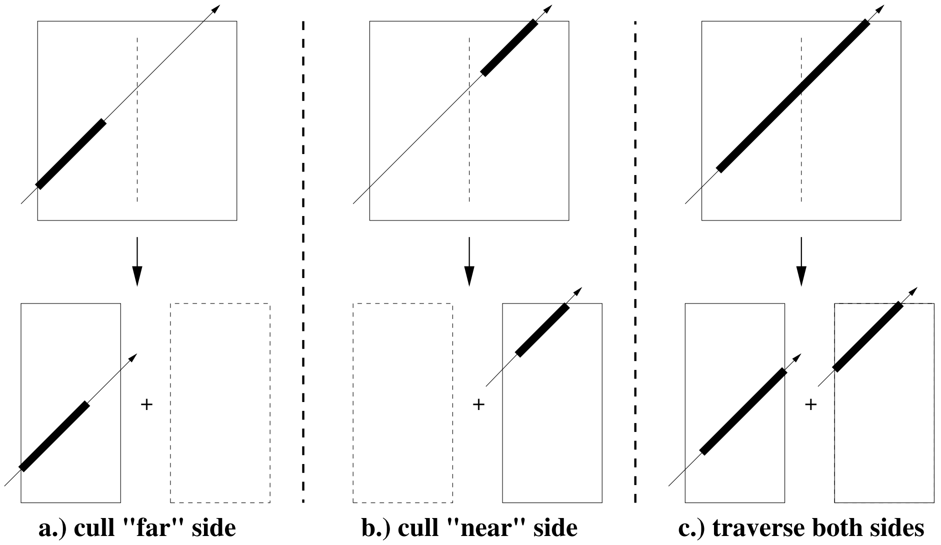


Figure 7.4: The three traversal cases in a BSP tree: A ray segment is completely in front of the splitting plane (a), completely behind it (b), or intersects both sides (c).

If the ray segment lies completely on one side of the splitting plane (i.e. $d >= t_{far}$ or $d <= t_{near}$), we can "cull" the subtree on the other side and immediately proceed to the corresponding child voxel[13]. If neither side can be culled, one computes the ray parameter at which the plane intersects, and traverses both sides in turn – the first side with ray segment $[t_{near}, d]$, and the second one with $[d, t_{far}]$. This actually leads to three different traversal cases, as depicted in Figure 7.4.

Basing the traversal entirely on the current ray segments allows for performing all computations in 1D: Only the actual ray parameters for start and end of the segment, as well as distance to the split plane have to be known. Neither the 3D coordinates of the actual entry, exit, or intersection points are required, nor is it necessary to track the current voxel's actual extent[14].

[13] Note that using "<=" and ">=" instead of "<" and ">" requires careful programming to correctly handle triangles that lie on the splitting plane. Also not that the exact implementation is quite sensitive to issues such as having rays parallel to the split plane, or rays actually lying inside the split plane. These special cases generate "Infinity"s and "NaN"s during traversal, which need special attention to handle correctly.

[14] This implies that the actual size of the voxel is not known at any time during traversal. Only the current ray segment – i.e. the overlap between the ray and the voxel – is known.

Early ray termination: In the just described implementation, voxels are traversed in front-to-back order, which allows for "early ray termination": If a valid hit point is found *inside* one voxel (i.e. $t_{hit} <= t_{far}$), traversal can be immediately terminated, as all further potential primitive intersections can only be be behind the already found hit point. This early ray termination is actually responsible for the "occlusion culling" feature of ray tracing, and can greatly enhance performance. Combined with a high-quality BSP tree (see Section 7.3), early ray termination can in many scenes lead to an average of *less than two* ray-triangle intersections per ray (see Table 7.5).

7.2.2.1 Recursive kd-Tree Traversal

In its most common recursive form, the whole traversal algorithm can be expressed quite simply:

```
void Traverse()
{
  ( t_near,t_far ) = ( Epsilon, ray.t_max );
  ( t_near,t_far ) = Clip(t_near,t_far);
  if (t_near > t_far)
     // ray misses bounding box of object
     return;
  RecTraverse( bspRoot, t_near, t_far );
}

float RecTraverse(node,t_near,t_far)
// returns distance to closest hit point
{
  if (IsLeaf(node)) {
    IntersectAllTrianglesInLeaf(node);
    return ray.t_closest_hit;
      // t_closest_hit initialized to t_max before traversal
  }
  d = (node.split - ray.org[node.dim] / ray.dir[node.dim];
  if (d <= t_near) {
    // case one,  d <= t_near <= t_far -> cull front side
    return RecTraverse(BackSideSon(node),t_near,t_far);
  } else if (d >= t_far) {
    // case two,  t_near <= t_far <= d -> cull back side
    return RecTraverse(FrontSideSon(node),t_near,t_far);
  } else {
    // case three: traverse both sides in turn
    t_hit = RecTraverse(FrontSideSon(node),t_near,d);
    if (t_hit <= d) return t_hit; // early ray termination
    return  RecTraverse(BackSideSon(node),d,t_far);
  }
}
```

7.2.2.2 Iterative kd-Tree Traversal

Due to the reasons discussed in the previous chapter, a recursive solution is not the best choice for high performance. However, the algorithm can be easily reformulated in an iterative way (see e.g. [Keller98, Havran01]), which in pseudo-code can be written up in only a few lines of code:

```
void Traverse() {
  ( t_near, t_far ) = ( Epsilon, ray.t_max );
  ( t_near, t_far )
      = scene.boundingBox.ClipRaySegment(t_near, t_far);
  node = rootNode;
  if (t_near > t_far)
     // ray misses bounding box of object
     return;
  while (1) {
     while (!node.IsLeaf()) {
        // traverse 'til next leaf
        d = (node.split - ray.org[node.dim]) / ray.dir[node.dim];
        if (d <= t_near) {
           // case one,  d <= t_near <= t_far -> cull front side
           node = BackSideSon(node);
        } else if (d >= t_far) {
           // case two,  t_near <= t_far <= d -> cull back side
           node = FrontSideSon(node);
        } else {
           // case three: traverse both sides in turn
           stack.push(BackSideSon(node),d,t_far);
           ( node, t_far ) = ( FrontSideSon(node), d );
        }
     }
     // have a leaf now
     IntersectAllTrianglesInLeaf(node);
     if (t_far <= ray.t_closesthit)
        return; // early ray termination
     if (stack is empty)
```

```
        return; // noting else to traverse any more...
      ( node, t_near, t_far ) = stack.pop();
  }
}
```

Obviously, a realtime kernel requires a very high-performance implementation of this traversal code with many low-level optimizations. For example, this includes precomputation of the "1/ray.dir[dim]" terms, an efficient stack handling, efficient calculation of "FrontSideSon" and "NearSideSon", careful data layout, and especially efficient handling, organization and ordering of the conditionals. Special emphasis has to be paid on handling all "special cases" – like for example division by zero ray direction (leading to +/- Infinity and NaN values), numerical issues (especially during the comparisons), triangles lying *in* the splitting plane, "flat voxels" leading to zero-length ray segments, etc – in an efficient though nevertheless correct manner. As the discussion of all these implementation details is quite involved, the actual low-level source code is omitted here.

7.2.3 SIMD Packet Traversal for kd-Trees

As discussed before, efficient use of the SSE instruction set during ray tracing requires to trace packets of several rays in parallel. The algorithm for tracing four different rays is essentially the same as traversing a single one: All four rays are first initialized to $(0, t_{max})$ and clipped to the scene bounding box using fast SSE code. In each traversal step then, SSE operations are used to compute the four distances to the splitting plane and to compare these to the four respective ray segments, all in parallel. If all rays require traversal of the same child, traversal immediately proceeds with this child, without having to change any of the ray segments. Otherwise, we traverse both children, with the ray segments updated accordingly.

As discussed in the previous section, efficient ray tracing requires to traverse the voxel visited by a ray in front-to-back order. However, when tracing several rays at the same time in parallel, the correct traversal order for the packet might be ambiguous, as different rays might demand a different traversal order. In order to get a consistent traversal order for the whole packet, we only allow such rays into the same packet for which the traversal order can be guaranteed to match. This however is easy to guarantee for two common cases, as discussed in more detail below: First, rays starting at the same origin can be shown to never disagree on traversal order, whatever their direction is. Second, rays with the same direction signs in all dimensions will also have the same traversal order at any splitting plane.

7.2.3.1 Resolving Traversal Order Ambiguities: Same Origin vs. Same Principle Direction

The first case already supports most of the rays in ray tracing, as all primary rays from a pinhole camera, as well as all shadow rays from point light sources fall under this category. However, the computations for determining the traversal order depend on the relation between actual origins of the rays and position of the splitting plane. As such, they have to be performed during each traversal step in the inner loop of the packet traversal code, and as such are quite costly. Furthermore, the operations for computing the respective updated ray segments get relatively complex for this alternative.

The second alternative of only combining rays with matching direction signs on first sight appears more costly: First, each packet of rays has to be checked for matching signs, and rays with non-matching signs either have to split up or require special handling. However, these special cases happen only rarely for coherent rays, which typically have similar directions. Once it is clear that the rays have matching direction signs, the computations in the inner loop get very simple, and can be expressed quite efficiently. In fact, all that is required in the inner loop of the traversal code is a simple XOR with the respective direction sign bit of the first ray. Similar arguments hold for the code computing the respective t_{near}/t_{far} values, which can be expressed quite a bit more efficiently than for the case with common ray origin. As such, the RTRT kernel only supports packets with matching directions signs. Packets are automatically and quickly tested for complying to this rule, and non-complying rays are traced with the fast single-ray traversal code.

7.2.3.2 Implementation Issues

After restricting the traversal code to packets with matching direction signs, the respective computations get quite simple. The plane distances for all four rays are computed with only one SSE “mult” and one SSE “add”, and compared to the four respective t_{near} and t_{far} values with SSE compare instructions[15]. If either *all* ray segments lie in front of the plane, or are all behind the splitting plane (corresponding to cases ’a’ and ’b’ in Figure 7.4), the other side is culled, and no special operations have to be performed for the near/far values, nor for the traversal stack. In the case that both sides have to be traversed [16], the respective ray segments get updated to

[15] Note that SSE comparisons are actually not conditionals, but rather generate bit masks that can be used for dependent moves

[16] Note that this case can also happen if *neither* ray wants to traverse both sides, as one ray might want to only traverse the left side, while an other one demands traversal of only

$[t_{min}, min(d, t_{far})]$ for the near side, respectively $[max(d, t_{near}), t_{far}]$ for the far side. The min and max operations are required as not all ray segments may actually have overlapped the splitting plane. These ray segments may obviously not get longer than they have been before.

Note that the "near" and "far" sides of a voxel (with respect to a given ray R) are determined by the order in which a directed infinite line with the same direction as R would cross this line[17]. As such, near and far side are independent of both ray origin and actual BSP plane position, and can be determined once at the start of traversal by the direction signs alone.

Deactivating invalid rays: Rays that get "forced" to traverse a subtree that they would not have traversed had they been traversed alone should obviously not influence any decisions in that subtree. This however can be achieved quite efficiently: Using the SSE min/max for updating the respective ray segments operations as just described, it can be shown easily that rays entering an "invalid" subtree automatically get their ray segments updated to negative length (i.e. $t_{near} > t_{far}$), which can be used to determine which of the rays are still "active" in a subtree. In SSE, this generates hardly any overhead at all: A single SSE compare of t_{near} and t_{far} automatically generates a bit-mask that can be used to mask out any of the latter decision flags in a single operation.

This leads to the following pseudocode for SIMD packet-traversal:

```
void IterativePacketTraverse(ray[4],hit[4]) {
  ( t_near[i], t_far[i] ) = ( Epsilon, ray.t_max );
  // i=0..3 in parallel
  // t_near[i], t_far[i] are the near/far values for the i'th ray
  ( t_near[i], t_far[i] )
      = scene.boundingBox.ClipRaySegment(t_near[i], t_far[i]);
  node = rootNode;
  while (1) {
    while (!node.IsLeaf()) {
      // traverse 'til next leaf
      d[i] = (node.split - ray[i].org[node.dim])
           / ray[i].dir[node.dim];
      active[i] = (t_near[i] < t_far[i]);
      if for all i=0..3 (d[i] <= t_near[i] || !active[i]) {
        // case one,  d <= t_near <= t_far for all active rays
        //            -> cull front side
```

the right side.

[17] The "near" side may not be confused with the "first" voxel visited by a ray, as the origin may actually lie on the "far" side.

```
        node = BackSideSon(node);
      } else if for all i=0..3 (d[i] >= t_far[i] || !active[i]) {
        // case two,  t_near <= t_far <= d for all active rays
        //             -> cull back side
        node = FrontSideSon(node);
      } else {
        // case three: traverse both sides in turn
        // correctly update all near/far values
        // push all near/far values for entire packet
        stack.push(BackSideSon(node),
                   max(d[i],t_near[i]),t_far[i]);
        ( node, t_far[i] )
           = ( FrontSideSon(node), min(d[i],t_near[i]) );
      }
     }
     // have a leaf now
     IntersectAllTrianglesInLeaf(node);
     if for all i=0..3 (t_far[i] <= ray[i].t_closesthit)
        return; // early ray termination
     if (stack is empty)
        return; // noting else to traverse any more...
     // restore all near/far values for entire packet
     ( node, t_near[i], t_far[i] ) = stack.pop();
  }
}
```

Note that all "x[i]" statements are always executed for all four rays in parallel using a SIMD instruction. While this algorithm only operates on packets of 4 rays, the extension to larger packet sizes is straightforward.

Note that the respective computations for properly computing the near/far values (including marking invalid ray segments) get quite a bit more involved for the alternative case in which the origin coincides but the directions differ.

The actual SSE implementation of this algorithm can be performed quite efficiently. Obviously, the same iterative algorithm as in the single ray code can be used, and many of the single-ray optimizations (such as changing the divisions to multiplies with the precomputed inverse) can be performed as well. All mathematical computations in the inner loop consist of only one SSE multiply and one SSE add. As SSE does not easily work together with non-SSE conditionals, many of the conditionals can be expressed more efficiently by SSE "conditional moves" (realized via SSE bit operations). Furthermore, all of the min/max operations for traversal case 3 can be expressed with a single SEE instruction each.

	2×2	4×4	8×8	256^2	1024^2
ERW6	1.4%	4.4%	11.8%	5.8%	1.4%
Office	2.6%	8.2%	21.6%	10.4%	2.6%
Conference.	3.2%	10.6%	28.2%	12.2%	3.2%

Table 7.3: Overhead (measured in number of additional node traversals) of tracing entire packets of rays at an image resolution of 1024^2 *in the first three columns: As expected, overhead increases with scene complexity (800, 34k, and 280k triangles, respectively) and packet size, but is tolerable for small packet sizes. The two columns on the right show the overhead for* 2×2 *packets at different screen resolutions.*

7.2.4 Traversal Overhead

Obviously, traversing packets of rays through the acceleration structure generates some overhead: Even if only a single ray requires traversal of a subtree or intersection with a triangle, the operation is always performed on all four rays. Our experiments have shown that this overhead is relatively small as long as the rays are coherent. Table 7.3 shows the overhead in additional BSP node traversals for different packet sizes.

As can be seen from this experiment, overhead is in the order of a few percent for 2×2 packets of rays, but goes up for larger packets. On the other hand, increasing screen resolution also increases coherence between primary rays.

Most important is the fact that the effective memory bandwidth has been reduced essentially by a factor of four through the new SIMD traversal and intersection algorithms as triangles and BSP nodes need not be loaded separately for each ray. This effect is particularly important for ray traversal as the computation to bandwidth ratio in relatively low.

Of course one could operate on even larger packets of rays to enhance the effect. However, our results show that we are running almost completely within the processor caches even with only four rays. We have therefore chosen not to use more rays per ray packet, as it would additionally increase the overhead due to redundant traversal and intersection computations, and would make the basic algorithm more complicated again[18]. For the SaarCOR architecture however (see Section 5.3), the same packet traversal principle is used with a significantly larger number of rays per packet.

[18] Larger packets especially suffer from the limited number of registers in the `ia32` architectures. Whereas most values for the single ray code can be kept in registers, larger packets require frequent load/store operations to save and restore certain values into the registers

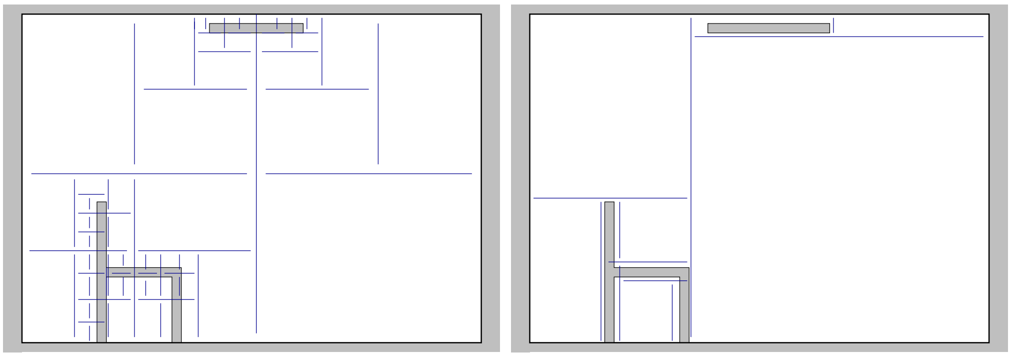


Figure 7.5: Naive kd-tree vs. high-quality kd-tree in a simple scene consisting of a room with one chair and one light source. Center: The scene with a BSP tree as it would result from a typical naive BSP construction code that always splits the biggest dimension in the middle, until a maximum depth or a minimum number of triangles is reached. Right: The same scene with a high-quality BSP as it results if the planes are placed based a good cost prediction function. Obviously, the BSP with the cost function would be significantly faster to traverse than the BSP with the naive plane placement. The effect of a good BSP tree can be even more pronounced in practical, more complex scenes.

7.3 High-Quality BSP Construction

Except for efficient traversal and intersection code as just described in Sections 7.1 and 7.2, the performance of a ray tracer using a kd-tree to a large degree depends on the algorithms with which the BSP tree has been built. Therefore, it is important to briefly discuss how good BSP trees can be built (for a more in-depth discussion of this topic, see e.g. [Havran01]).

Once the kd-tree has been built – i.e. the location and orientation of the BSP planes, and the decision when to stop subdivision have been fixed – the number of traversal steps and triangle intersections for a given ray and traversal algorithm is predetermined. As such, building a BSP tree that better adapts to the scene complexity directly influences these two critical performance parameters. This can have a *significant* impact on overall performance: For example, since its original publication in [Wald01a], the RTRT core has been enhanced with a better BSP construction code which has roughly *doubled* its performance – on top of the already very high performance as originally published. This speedup of two is entirely due to the improved BSP tree, and did not require any other changes to the core[19].

[19] Note that similar speedups apply for the SaarCOR architecture as described in Sec-

When building BSP trees, the most common approach is to always split each voxel in the middle. In the most naive approach, the splitting dimension is chosen in a round-robin fashion, and subdivision proceeds until either a maximum depth has been reached, or voxel contains less than a specified number of triangles[20]. However, it is common knowledge that the BSP tree for non-cube-like scenes can be improved by always splitting the box in the dimension where it has maximum extent[21]. This can be explained by the fact that this approach produces the most cube-like voxels[22]. However, it is also long known that putting the plane into the middle might not be a perfect position, either [Havran01].

Many people assume that placing the split plane towards the object median (i.e. placing it such that both halves contain an equal number of triangles) would be a better choice. Though this appeals to intuition, it is actually a very bad choice. Splitting at the object median aims at building a balanced tree with equal depth of all leaves. Though this is optimal for binary search trees with equal access probabilities to each leaf node, it is not optimal for ray tracing with a kd-tree: First, the probability of accessing different voxels is certainly not equally distributed, as larger voxels are more likely to be hit than small ones. Furthermore, traversing a kd-tree is actually not the same as a search in a binary search tree (in which traversal always proceeds from the root to the leaf in one straight line), but rather a range searching process in which several leaves have to be accessed, and in which traversal frequently goes up and down in the tree. As such, BSP trees should not be optimized towards having an equal number of traversal steps towards each leaf (i.e. balancing it), but should rather minimize the number of traversal steps for traversing a ray from one location to another. For this kind of traversal, BSP trees behave best if they have large voxels of empty space as close to the root node as possible, as large "empty space" allows for traversing a ray over a large distance at small cost. Splitting at the object median results in empty space being pushed far down the tree into many small voxels, and thus leads to many traversal steps and bad performance.

tion 5.3: As the SaarCOR architecture uses exactly the same data structures as the RTRT kernel (and in fact uses RTRT to generate the binary scene dumps it runs on), any speedups due to better BSPs translate similarly to better SaarCOR performance!

[20] In practice, 20–25 for maximum depth, and 2–3 for the "triangles per leaf" threshold are usually close to optimal values.

[21] Interestingly, though this is "common knowledge", it is actually a misconception except for extremely "non-cubic" voxels, as can be seen in Table 7.4 (columns 'RR' vs. 'ME'): For most scenes, splitting in the middle is actually slightly faster.

[22] For cube-like voxels, the ratio of voxel surface to voxel volume reaches its minimum. As the voxel surface influences the probability of a voxel to be hit by a ray[MacDonald89], a voxel of a given volume has the least chance of being traversed.

Scene	#triangles	absolute performance				speedup	
		RR	ME	PS	SAH	PS	ME/RR
ERW6	804	4.33	4.16	4.53	8.18	80 %	89 %
ERW10	83,600	1.30	2.74	3.03	5.51	81 %	101 %
Office	34,000	2.50	2.32	2.85	4.31	51 %	72 %
Theater	112,306	1.30	1.12	1.47	2.43	65 %	87 %
Conference (sta)	282,801	2.18	1.89	2.47	4.17	69 %	91 %
SodaHall (in)	2,247,879	2.50	2.13	2.87	3.46	20 %	38 %
SodaHall (out)	2,247,879	2.62	2.78	3.63	4.08	12 %	47 %
Cruiser	3,637,101	1.67	1.56	2.03	3.01	48 %	80 %
PowerPlant (in)	12,748,510	0.51	0.50	0.81	1.26	56 %	147 %
PowerPlant (out)	12,748,510	0.72	0.78	0.97	1.44	48 %	84 %

Table 7.4: Relative performance of rendering with BSPs built by different construction algorithms: Kaplan-BSP with round-robin subdivision (RR), Splitting the voxel in the dimension of maximum extent (ME), "PlaneShifter", i.e. ME with shifting the plane to maximize empty voxels (PS), and a surface area heuristic (SAH). Numbers correspond to million primary rays per second with SSE code on a 2.2GHz Pentium-IV Xeon. Right two columns show the relative SAH speedup as compared to PS, ME and RR. As expected the SAH performs best. Except for Soda Hall, SAH usually performs 50–80 percent faster than the best other method. Note that the effect in practice is even more pronounced: Whereas RR, ME and PS require extensive parameter tuning to achieve the result given in this table, the SAH performs reasonably well already with its default parameters.
The respective scenes can be seen in Figure 7.7, some statistical data on the generated BSPs is given in Table 7.5.

Some other intuitive improvements to the split plane position lead to more successful heuristics. For example, if one of the half-voxels produced by a split is empty, the argument of empty space being beneficial suggests that the split plane should be "shifted" as far into the non-empty half as possible. This reduces the probability of the ray having to traverse the non-empty leaf, significantly improves the BSP quality, and is easy to implement. This heuristic can also be furtherly refined to yield even more improvements. Though the results of such intuitive approaches are quite limited – in the range of 30–50 percent over the naive construction method (see Table 7.4) – they are relatively easy to implement, and thus should always be preferred over the naive approach. However, these "simple" heuristics by far cannot match the BSP quality that can be generated with a well-designed cost function (see below).

Scene	BSP	num.	number of			Triangle-Isecs	
(view)	generation	trav.	traversed leaves			mailboxing	
	strategy	steps	(total)	(empty)	(full)	yes	no
ERW6	Kaplan	32.22	8.05	1.60	6.46	15.51	6.35
	PS	33.45	7.76	4.31	3.45	9.78	5.83
	SAH	20.97	4.32	3.25	1.07	1.46	1.45
ERW10	Kaplan	51.14	9.88	1.66	8.22	17.31	8.39
	PS	54.15	9.70	6.65	3.05	7.50	6.41
	SAH	32.35	5.35	4.27	1.07	2.65	2.65
Office	Kaplan	58.80	12.76	7.47	5.29	11.63	6.03
	PS	60.04	12.10	10.64	1.46	3.39	2.73
	SAH	35.09	6.53	5.37	1.15	3.46	3.36
Theater	Kaplan	98.22	18.21	15.03	3.19	12.52	7.96
	PS	88.48	15.19	13.44	1.74	5.21	4.07
	SAH	64.86	10.40	9.13	1.28	3.79	3.68
Conference	Kaplan	68.10	14.25	9.48	4.78	9.91	5.63
	PS	68.91	13.61	12.31	1.29	2.82	2.38
	SAH	38.32	6.87	5.63	1.24	2.53	2.30
Soda Hall	Kaplan	61.96	8.70	5.45	3.25	9.58	6.20
(inside)	PS	60.06	8.24	6.81	1.43	3.73	2.98
	SAH	50.12	5.34	4.22	1.12	2.64	2.62
Soda Hall	Kaplan	99.92	17.10	14.29	2.81	8.04	5.52
(outside)	PS	73.16	11.56	10.38	1.17	2.89	2.67
	SAH	62.70	9.136	8.09	1.04	1.78	1.78
Cruiser	Kaplan	74.95	11.05	6.77	4.28	14.84	11.15
	PS	78.40	11.2	9.52	1.68	5.31	4.08
	SAH	52.34	7.019	5.74	1.28	2.73	2.57
PowerPlant	Kaplan	108.7	15.62	11.30	4.33	105.22	81.73
(inside)	PS	90.65	12.52	10.73	1.79	41.25	35.12
	SAH	72.79	9.18	7.93	1.25	5.82	5.69
PowerPlant	Kaplan	189.1	32.45	28.52	3.93	40.06	28.26
(outside 2)	PS	132.7	22.02	19.82	2.20	15.75	12.13
("overview")	SAH	109.7	19.61	17.99	1.62	10.12	9.79

Table 7.5: Impact of the different BSP generation strategies on traversal parameters: This table shows (for different scenes and views) the average number of BSP traversal steps per ray, average number of leaves encountered during traversal (empty vs. non-empty leaves), and number of ray-triangle intersections with and without mailboxing, respectively[24]*. Generation strategies measured include "Kaplan", "PlaneShifting", and Surface Area Heuristic see Table 7.4). For both Kaplan and PS, several parameter sets have been tested, the number given here corresponds to the parameter set that achieved best performance. Note that the exceptionally high number of triangles visited for the Kaplan BSP in the PowerPlant model results from the high memory consumption of the Kaplan BSP, which did not allow for "deeper" BSP trees in a 32-bit address space.*

This table clearly shows the improvements due to a better BSP tree, especially for the number of non-empty leaves, and the number of triangles visited by each ray. For images and performance of these scenes, also see Table 7.6 and Figure 7.7.

7.3.1 Surface Area Heuristic (SAH)

A more successful – though unfortunately also quite more complicated – approach is to optimize the positioning of the splitting plane via cost prediction functions in the spirit of Goldman and Salmon [Goldsmith87], MacDonald and Booth [MacDonald89, MacDonald90], and Subramanian [Subramanian90a]. Such a cost prediction function uses certain assumptions for estimating how costly a split would be. This estimate can then be used to place the plane at the position of minimal cost. Furthermore, the cost function provides a much more effective termination criteria for the subdivision than the above-mentioned "maximum depth and triangle threshold": Using a cost-estimate function, subdivision is simply terminated as soon as the estimated traversal cost for a leaf node is less than the cost for the split with minimum estimated cost.

The most famous of these cost prediction functions is the "surface area heuristic" (SAH) as introduced by MacDonald and Booth [MacDonald89, MacDonald90]: The surface area heuristic assumes that rays are equally distributed in space, and are not blocked by objects. Under these (somewhat unrealistic) assumptions, it is possible to calculate the probability with which a ray hitting a voxel also hits any of its sub-voxels. More specifically, having a voxel V that is partitioned into two voxels V_L and V_R, the probability of a ray traversing these two sub-voxels can be calculated as

$$P(V_L|V) = \frac{SA(V_L)}{SA(V)} \text{ and } P(V_R|V) = \frac{SA(V_R)}{SA(V)}$$

where $SA(V) = 2(V_w V_d + V_w V_h + V_d V_h)$ is the surface area of voxel V (with V_w,V_h, and V_d being width, depth and height of the voxel, respectively).

Once these respective probabilities are know, one can estimate the cost of a split: Assuming that a traversal step and a ray triangle intersection have an average cost of C_{trav} and C_{isec} respectively, the average cost of splitting voxel V into V_L and V_R can be estimated as

$$Cost_{split}(V_L, N_L, V_R, N_R) = C_{trav} + C_{isec}(P(V_L|V)N_L + P(V_R|V)N_R)$$

where N_L and N_R are the number of triangles in V_L and V_R, respectively.

7.3.1.1 Finding the best split positions

This function is continuous except for the split plane positions at which the numbers N_L and N_R change (also see [Havran01]). These are exactly the positions where either a triangle side ends (i.e. at a vertex), or where a triangle side pierces the side of a voxel [Havran01, Hurley02]). These locations form the "potential split positions", from which the position with the minimum cost is chosen. Unfortunately, checking all potential splits can be quite expensive, and requires a carefully designed algorithm to avoid quadratic complexity during each splitting step. Furthermore, finding all potential splits can be quite costly and numerically unstable, especially for those potential splits that are computed by intersecting a triangle side with the voxel surface.

Instead of performing these side-voxel intersections it is also possible to only consider each triangle's bounding box sides as potential split planes. This is much easier to implement, and still performs better than not using the SAH at all. However, "perfect" split positions usually achieve superior performance than only considering the bounding box sides. As such, the RTRT core uses perfect split positions, and uses a carefully designed implementation to avoid all potential numerical inaccuracies without sacrificing performance.

7.3.1.2 Automatic termination criterion

Using the above assumptions, one can estimate the minimum cost of traversing the split object. Similarly, one can estimate the cost of not splitting a voxel at all, as $Cost_{leaf}(V) = N_V \times C_{isec}$. Simply comparing these two values provides a very simple and efficient termination criterion. Of course, it is still possible to combine the surface area heuristic with other heuristics. For example, it may make sense to still specify a maximum tree depth[25], or to add heuristics for encouraging splits that produce empty space (see e.g. [Havran01]).

7.3.2 Post-Process Memory Optimizations

The BSP construction process in the RTRT core actually is a two-stage process. While the optimized data layout described in the previous section is quite easy to use during traversal, it would be quite awkward to use while building the BSP. As such, we first build the BSP tree with a more easy-to-use node layout that uses twice as much memory and lots of pointers. Once the build-tree process is finished, RTRT performs several optimizations on the BSP tree (see Figure 7.6): First, for some build-tree algorithms RTRT first iterates over the whole tree a second time, thereby undoing any splits that have not produced useful results (e.g. a node with two leaves containing the

[25] Compared to Kaplan-BSPs, a maximum tree depth with the surface area heuristic is more likely to be in the range of 50 or more

same item lists)[26]. Then, this memory-unfriendly data layout is re-arranged to the more cache-friendly form as described above. Though this data reorganization is quite costly, it is much more convenient than having to program the whole BSP construction code directly on the optimized data layout.

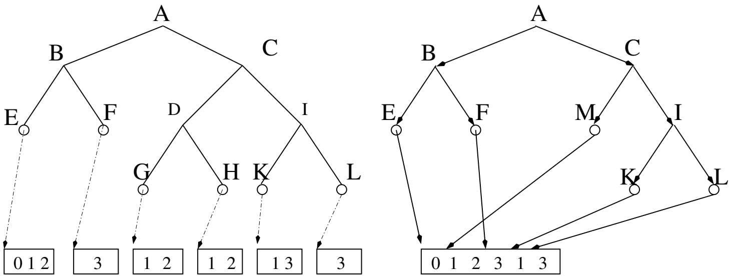


Figure 7.6: Post-process memory optimizations: After construction, splits that did not produce sensible results get collapsed (e.g. nodes G and H), and the item lists are stored in a compressed form by checking whether the same node list can already be found in the list array. Different item lists can overlap the same memory regions without any problems, as the length of the list is stored in the BSP node anyway. After these collapse operations, the BSP is reformatted to the memory-compressed form as shown in Figure 7.3.

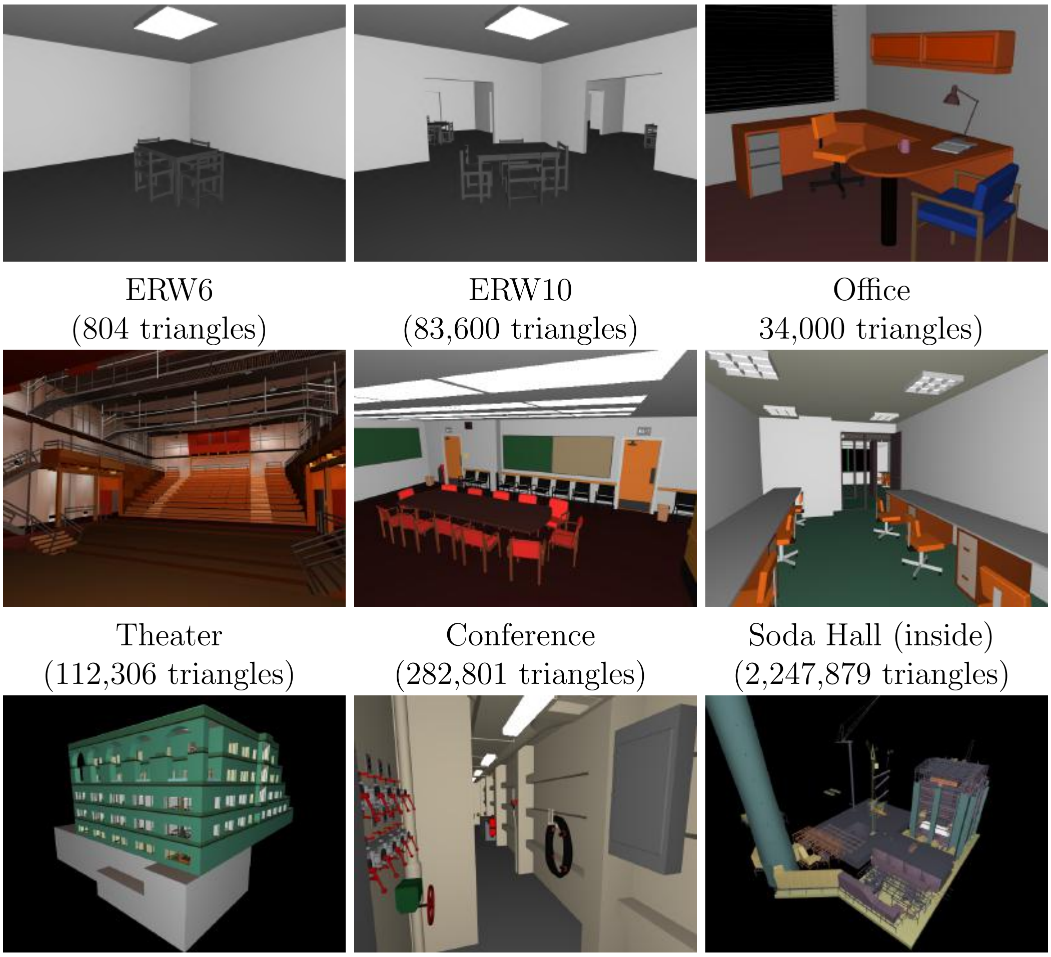
Finally, it is possible to perform some minor optimizations during the data rearrangement, such as having similar item lists use the same memory space. For example, the item lists “12,13,17” and “13,17” can be stored in the same memory region if the pointer for the second lists points “into” the first list (see Figure 7.6). Though this can save some memory especially for deeply subdivided BSPs, the performance impact of these final optimizations is quite limited.

7.3.3 Results of different BSP Construction Strategies

In its current implementation, the surface area heuristic in typical scenes is roughly 50–100 percent faster than a typical Kaplan-type BSP (see Table 7.4), and is still up to 50 percent faster than the best non-SAH as implemented in RTRT by 2001 (as used in the original 2001 “Coherent Ray Tracing” paper [Wald01a]).

Though these results are impressive, the surface area heuristic also has several problems. First of all, it can be quite costly to generate, especially for complex scenes. Second, the SAH – though being already very good –

[26] Obviously, this could also be done already during BSP construction.



ERW6
(804 triangles)

ERW10
(83,600 triangles)

Office
34,000 triangles)

Theater
(112,306 triangles)

Conference
(282,801 triangles)

Soda Hall (inside)
(2,247,879 triangles)

Soda Hall (outside)
(2,247,879 triangles)

Cruiser
(3,637,101 triangles)

Power Plant (outside)
(12,748,510 triangles)

Figure 7.7: The scenes used for the RTRT benchmarks in Table 7.6. Including simple SSE shading, these scenes run at 1.3–5.4 frames per second at full-screen (1024×1024*) resolutions on a single 2.5GHz Pentium-IV notebook CPU (see Table 7.6).*

is still not optimal[27]. Following a greedy strategy for picking the split plane can lead to getting stuck in local minima. The same is actually true for the termination criterion: Very easily, it may happen that no split can be found with a cost less than the cost of making a leaf – in which case a leaf will be generated – even though a better configuration might be found if another level of splits were considered (see e.g. Figure 7.8). This could be fixed by

[27] Computing the best BSP tree is known to be NP-complete [Havran01].

using a global optimization method, which however would probably be far too costly to generate. More importantly, the SAH is quite complicated to implement correctly, and is error-prone both to programming bugs as well as to numerical inaccuracies.

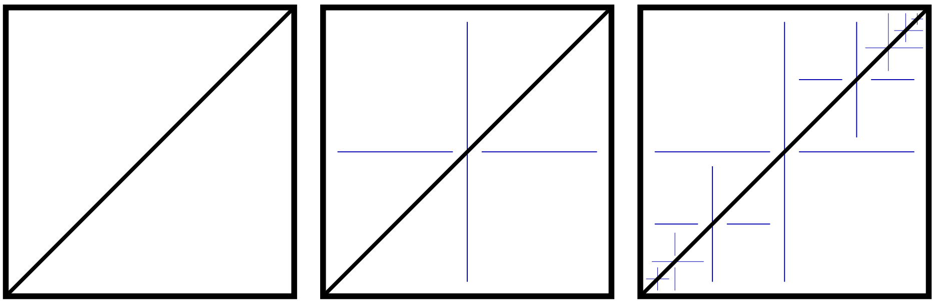


Figure 7.8: With a greedy method for choosing the split plane, the surface area heuristic can get stuck in local minima. For example, no single split plane can be found that subdivides the left voxel in a way that would have a better cost function than creating a leaf (as each side would have as many triangles as the node itself). If however a "non-optimal" split were allowed in the center, the following split would find a configuration that has less cost than the left one (center image). Right: The same argument can be repeated infinitely, making automatic termination problematic if such splits are allowed. Note that this is a very common configuration for practical scenes, as for example all walls of a room match this setting.

Finally, the SAH requires the ray tracer to work exactly: For example, working on perfect split positions often leads to the generation of "flat" cells with zero width: All triangles that are orthogonal to a coordinate axis (such as walls) will eventually end up in a cell that exactly encloses them, and which thus will be flat[28]. This can easily lead to numerical problems during traversal, as a ray traversing an empty cell actually has a zero-length overlap with this voxel, which may easily be "over-seen" by the traverser. Though this is not exactly a problem of the SAH, it may still lead to problems when using it. Obviously, the RTRT traversal code correctly handles this case.

[28] This case also has to be handled correctly during BSP construction: For example, when further subdividing a flat cell, the construction code has to take care when computing the side-voxel intersections.

CPU / scene	#tris	absolute performance (fps@1024x1024, 1CPU)			
ray tracing		SSE	SSE	SSE	C
shading		none	SSE	C	C
ERW6 (static)	804	8.95	5.38	3.80	2.09
ERW6 (dynamic)	804	4.00	3.05	2.57	1.33
Office (static)	34,000	4.68	3.45	2.86	1.39
Office (dynamic)	34,000	2.61	2.17	1.87	0.88
ERW10	83,600	5.82	3.88	3.27	1.65
Theater	112,306	2.68	2.18	1.95	1.05
Conference (dynamic)	282,801	3.17	2.50	1.98	1.01
Conference (static)	282,801	4.40	3.26	2.61	1.44
Soda Hall (in)	2,247,870	3.68	2.85	2.46	1.19
Soda Hall (out)	2,247,870	4.47	3.28	3.19	1.78
Cruiser	3,637510	3.38	2.65	2.31	1.17
Power Plant (in)	12,748,510	1.43	1.27	1.19	0.53
Power Plant (out)	12,748,510	1.59	1.39	1.40	1.17

Table 7.6: RTRT core performance in million rays per second on a single *2.5GHz Pentium-IV notebook CPU at a resolution of* 1024×1024 *pixels, in different shading configurations: SSE/none corresponds to pure ray traversal and intersection performance without shading at all; SSE/SSE means SSE packet tracing with a hard-coded simple SSE shading model; SSE/C means SSE ray tracing with C-code shading (including SoA-to-AoS data re-packing overhead); and C/C means pure C-code single ray traversal and shading. Though ray tracing scales nicely with scene complexity, even simple shading can already cost more than a factor of two given current ray tracing performance! The above numbers directly correspond to the achievable frame rate on a single 2.5GHz Pentium-IV notebook CPU at full-screen resolution (*1024×1024 *pixels). The respective benchmarking scenes can be found in Figure 7.7.*

7.4 Current RTRT Performance

As described in the previous section, the RTRT software ray tracing kernel builds the combination of highly optimized traversal and intersection routines, tracing packets of rays for efficient SIMD support, and a special emphasis on caching and memory optimizations. Though the newest version of the RTRT core still uses the same ideas as discussed in its original publication [Wald01a], the RTRT kernel since then has been significantly improved and completely re-implemented to achieve significantly higher performance [Wald03e]. This increase in performance is due to a combination of several factors:

Faster CPUs: Obviously, CPUs have become significantly faster since 2001 (from around 800MHz Pentium-III's to 3GHz Pentium-IV's today). While many other applications cannot fully benefit from this increase in clock rate, the RTRT core has been designed to fully exploit the available CPU performance (e.g. by minimizing cache misses, pipeline stalls and branch mis-predictions), and as such benefits linearly from improved CPU performance. Though the performance increase of modern CPUs is obviously not an achievement of the RTRT core itself, it is due to its special design – especially its emphasis on SIMD support and caching optimizations – that have enabled the RTRT kernel to benefit linearly from any increase in CPU performance.

Better BSP Trees: The "Coherent Ray Tracing" paper cared mostly about the fast traversal of an existing BSP tree, and neglected the algorithms for building these BSPs. The new RTRT core uses an improved "surface area heuristic" (SAH) cost prediction function for generating optimized BSP tree (see Section 7.3), which result in up to *twice* the performance than with the BSP construction code as used in the original Coherent Ray Tracing system.

Better Compilers: Modern compilers offer increasingly powerful tools for writing better and faster code. For example, RTRT achieves roughly *twice* the performance when compiling its single-ray code (which is written in plain "C/C++") with Intel's ICC (Version 7.1) compiler as compared to compiling it with the 2001 version of the GNU gcc compiler as used in the original system[29]. Comparing to most up-to-date code written in ICC intrinsics with the performance of the original 2001 SSE code written in hand-coded assembler yields similar speedups.

Better Implementations: The RTRT core algorithms cover only a few hundred lines of code, and are continuously being optimized. Since its original publication in 2001 [Wald01a], the core code has been re-implemented several times, having resulted in a significant increase in performance.

[29] The new gcc versions 3 and higher are supposed to offer similarly increased performance over pre-3.0 gcc's. Preliminary tests with gcc 3.3.1 have been positive, but a thorough evaluation has not yet been performed.

Taken together, these methods allow the current core to significantly outperform the old system even when running the old code on an up-to-date CPU. Even when traversing single, incoherent rays (i.e. *without* using the SSE instruction set) the new kernel is slightly faster than the originally published SSE code tracing packets of rays.

Exploiting the full performance of the newest SIMD code then achieves an additional performance improvement of 2–3 when shooting coherent rays (see Table 7.6). It is important to note that the RTRT kernel does not use any approximations to achieve this speedup. It still performs at least the operations of a traditional ray tracer. Considering only the pure traversal and intersection cost – i.e. without shading and without support for dynamic scenes – the RTRT kernel achieves up to $\sim$ 9 million rays per second on simple scenes, and still 1.4–4.4 million rays per second on as complex scenes as the soda hall and power plant scenes (with 1.5 and 12.5 million triangles, respectively).

Casting only primary rays with relatively simple shading, this performance allows for computing several (1.3–5.4) full screen frames per second even on a single notebook with a typical 2.5GHz Pentium-IV CPU (see Table 7.6 and Figure 7.7). Using a state of the art dual-CPU PC, this level of ray tracing performance allows generate impressive frame-rates even on a single desktop machine.

7.5 Future Work

As can be seen by the results mentioned in Table 7.6, it is clear that the biggest individual bottleneck – and thus the biggest remaining problem to be solved – is the cost for shading. As the cost for shading has traditionally been cheap compared to the cost for tracing a ray, this problem so far has not received much attention. With the current increase in ray tracing performance however even simple shading incurs a severe performance impact. As such, the biggest potential for future performance gains lies in finding ways for faster shading. However, as already discussed in Section 6.4.5 it is still unclear how this can be achieved.

Apart from faster shading, we expect that even higher ray tracing performance can be achieved by exploiting even more coherence by using larger packets. Larger packets should allow for optimizations in which not all individual rays in a packet have to be considered in each traversal step. For example, two out of the three traversal cases could be accelerated by only looking at the “corner rays” of a packet[30]. Similarly, the efficiency of the

[30] For primary rays, it is obvious to define the corner rays for a packet. For secondary

SSE code could probably be increased by larger packets, as any setup cost (such as fetching triangle data) could be amortized over more rays. Though larger packets obviously suffer from decreased coherence, this may be offset by the continuing trend towards higher image resolutions.

Furthermore, it has to be investigated how the ideas that have proven so successful in accelerating ray tracing for polygonal scenes could also be employed for other kind of ray tracing primitives, such as volumetric objects, isosurfaces, or parameteric patches.

Finally, it has to be investigated how much it is possible to further improve the quality of the BSP trees. While the average number of triangles hit by a ray is close to the optimum (see Table 7.5), it may still be possible to further reduce the number of traversal steps.

rays, the “corner” rays could be defined by the corners of an imaginary shaft bounding the rays.

Chapter 8

The RTRT Parallelization Framework

"If it won't yield to brute force, use a bigger hammer"
anonymous

Even though the ray tracing performance of the RTRT kernel allows for some limited amount of interactivity already on a single processor, one CPU alone still cannot (yet) deliver the performance required for practical applications that require complex scenes and shading, plus many rays for effects like shadows or reflections[1]. Achieving sufficient performance on todays hardware thus requires to combine the computational resources of multiple CPUs.

As current PC systems are limited, using multiple CPUs currently implies the use of clusters of single- or dual-processor PCs. Note however that this does not imply dedicated cluster systems, but can just as well be realized by loosely coupling several off-the-shelf desktop PC systems. In the medium term, it is likely that small-scale multiprocessor shared-memory systems will also be available for the PC market[2]. Given the ongoing trend towards even more powerful CPUs, it is even likely that in the longer term realtime ray tracing will eventually be possible on a single CPU. Until then however the most cost-effective approach to compute power is to use a distributed

[1] Note that scene complexity can usually be handled relatively well even on a single PC, at least as long as that single client can store the complete scene. Thus the main reason for parallelizing is the number of rays that has to be computed for images of reasonable (shading) complexity.

[2] For example, some eight-node shared-memory Opteron [AMD03b] systems have recently become available. These however are still significantly more costly than a respective 4-node dual-Opteron cluster.

memory PC cluster. Thus, we are mainly concentration on such commodity hardware[3].

This however does not mean that using shared-memory systems does not make any sense at all, nor that RTRT/OpenRT will *only* work on such cluster systems. As cluster systems tend to be more complicated to use than shared-memory systems, any software that efficiently runs in parallel on a distributed memory cluster is likely to run at least as good on a shared-memory system[4].

8.1 General System Design

While it is reasonably simple to parallelize an offline ray tracer, achieving interactive performance and good scalability at realtime frame rates requires special care when designing and implementing the parallelization framework. Especially the low bandwith and high latency of our commodity networking technology become problematic at realtime frame rates, as network round-trip times are typically much larger than tracing a large number of rays[5].

8.1.1 Screen Space Task Subdivision

Effective parallel processing requires breaking the task of ray tracing into a set of preferably independent subtasks. For predefined animations (e.g. in the movie industry), the usual way of parallelizing ray tracing is to assign different frames to different clients in huge render farms. Though this approach successfully maximizes throughput, it is not applicable to a realtime setting, in which only a single frame is to be computed at any given time.

For realtime ray tracing, there are basically two approaches: *object space* subdivision, and *screen space* subdivision [Reinhard95, Chalmers02, Simiakakis95]. Object space approaches store the scene database distributed across a number of machines, usually based on an initial spatial partitioning scheme. Rays are then forwarded between clients depending on the next spatial partition pierced by the ray. However, the resulting network bandwidth would be too large for our commodity environment. At today's ray tracing performance individual rays can often be traced much faster than they can be transferred

[3] Typical configurations include up to 24 dual-Athlon MP 1800+, or 4–8 dual Pentium IV/Xeon nodes, usually connected with 100Mbit FastEthernet or Gigabit Ethernet.

[4] Currently, RTRT/OpenRT has also been ported to various other hardware platforms, such as 64bit AMD Opteron, SGI/Mips, or SUN/Sparc, though not all of these support the SIMD packet tracing mode.

[5] At a core performance of 1–10 million rays per second (see Chapter 7), a round-trip time of 1ms, corresponds to 1,000 to 10,000 rays (even twice as much for a dual-CPU node).

across a network[6]. Finally, this approach often tends to create *hot-spots*, which would require dynamic redistribution of scene data. This in turn in quite problematic when using commodity networking technology, especially when targeting interactive response times.

Instead, we will follow the screen-based approach by having the clients compute disjunct regions of the same image. The main disadvantage of screen-based parallelization is that it usually requires a local copy of the whole scene to reside on each client, whereas splitting the model over several machines allows for rendering models that are larger than the individual clients' memories.

Usually, we do not consider this special problem, and rather assume that all clients can store the whole scene. In a previous publicatoin [Wald01c, Wald01b], we have shown how this problem can be solved efficiently by caching parts of the model on the clients and loading data on demand. Using this approach, models larger than the individual client's memories could be rendered, as long as the combined memories of all clients have been large enough to hold the working set of the entire model[7]. This respective implementation (see [Wald01c] for more details) showed the feasibility of the general approach, but was mainly a prototype implementation that left many opportunities for improvements[8].

8.1.2 Load Balancing

In screen space parallelization, one common approach is to have each client compute every n-th pixel (so-called pixel-interleaving), or every n-th row or scanline. This usually results in good load balancing, as all clients get roughly the same amount of work. However, it also leads to a severe loss of *ray coherence*, which is a key factor for fast ray tracing. Furthermore, each client essentially computes a downscaled image of the same view, and as such touches all the data visible in that view. Similarly, it translates to bad cache performance resulting from equally reduced *memory coherence.*

[6] Assuming 50 bytes to describe a ray, a core performance of 1–10 million rays per second would require a bandwidth of 50–500 megabytes per second just for being able to *send* the rays faster than they can be traced (twice as much for a dual-CPU node). This communication bottleneck also complicates any efforts of building any PCI boards as ray tracing 'coprocessors', except if that PCI board handles *all* ray tracing operations without any communication with the CPU (e.g. [Schmittler02].

[7] The same approach also works well for a hardware implementation in the SaarCOR architecture, as described in [Schmittler03].

[8] Note that the current RTRT/OpenRT system does *not* use this distributed data management, but currently replicates the entire scene on all clients as described below!

An alternative approach – that we have adopted – is to subdivide the image into rectangular "tiles" and assign those to the clients. Thus, clients work on neighboring pixels that expose a high degree of coherence. The drawback is that the cost for computing different tiles can significantly vary if a highly complex object (such as a complete power plant as shown in Section 11.3) projects onto only a few tiles, while other tiles are empty. For *static task assignments* – where all tiles are distributed among the clients *before* any actual computations – this variation in task cost would lead to bad client utilization and therefore result in low scalability.

Therefore, RTRT combines the tile-based approach with a dynamic load balancing scheme: Instead of assigning all tiles in advance, the clients follow a demand-driven strategy and will themselves ask for work: As soon as a client has finished a tile, it sends its results back to the master, and thereby automatically requests the next unassigned tile.

8.1.2.1 Optimal Tile Size

Given the ray tracing performance shown in Section 7, efficient load balancing requires having enough tasks with a high enough cost available (on each node) in order to offset the high network communication latency. For simple scenes with simple shading, it becomes a problem to have enough tiles available to keep all clients busy. However, using more and smaller tiles increases the network load and decreases the available coherence within each task. Furthermore, smaller tiles carry less "payload" of pixels to be computed while still requiring the same communication cost, and as such result in a significantly reduced ratio of computation to communication cost. For a given number of clients and compute-to-latency ratio there is a tile size that optimizes the achievable frame rate. While this optimal tile size depends on the actual settings, tiles sizes of 8 × 8 pixels (for very costly per-pixel computations) to 32 × 32 pixels (for extremely simple computations) seem to be sensible[9]. If not manually overridden, RTRT uses a default tile size of 16 × 16, which usually achieves good results.

8.2 Optimizations

Screen space parallelization and dynamic load balancing are both well-known and are applied in similar form in many different parallel ray tracing systems

[9] It should be possible to automatically and dynamically determine the optimal tile size during runtime. This however has not yet been implemented.

(for an overview, see e.g. [Chalmers02]). However, the need for communication with the different client machines – together with the high network latencies of commodity PC hardware – require very careful optimizations and several additional techniques to achieve realtime performance and good scalability.

8.2.1 Efficient Communication

Most standardized libraries such as MPI [Foruma] or PVM [Geist94] cannot provide the required level of flexibility and performance that we are faced with in an interactive environment. For example, TCP/IP has a built-in optimization (the so-called "Nagle" algorithm) that combines several small network packages to larger packets. This optimization can significantly increase network throughput, as EtherNet requires relatively large packet sizes to achieve its optimal performance [Stevens98]. Unfortunately, this algorithm can also result in significant latencies, as small packets can be significantly deferred in order to determine whether there are any following packets with which it can be combined. This is extremely disturbing for small "control packets" in which a client is instructed to perform some work. Deferring such packets can significantly reduce the overall system performance. Using a manually coded TCP/IP implementation allows for selectively turning such optimizations on for certain data, and turn it off for other. For example, sending scene updates to the clients requires maximum bandwidth, and as such has the Nagle algorithm turned on. For data that is sensitive to delays however (such as command packets) we use a separate socket for which this optimization is turned off. Such low-level optimization are much more complicated (though perhaps not impossible) to implement in standardized libraries such as MPI [Foruma] or PVM [Geist94][10], as these usually do not allow for as fine a level of control over how exactly the communication is implemented.

Therefore, all communication in the RTRT/OpenRT engine has been implemented from scratch with standard UNIX TCP/IP calls [Stevens98]. This ensures a minimum of communication latency, and extracts the maximum performance out of the network.

[10] However, switching to PVM/MPI may still be beneficial when considering different networking hardware such as MyriNet [Forumb] or InfiniBand [Futral01], for which the relative overhead of the TCP/IP protocol stack is even higher high.

8.2.2 Frame Interleaving

Another source of latency is the interval between two successive frames, in which the application usually changes the scene settings before starting the next frame. During this time, all clients would run idle. To avoid this problem, rendering is performed asynchronously to the application: While the application specifies frame N, the clients are still rendering frame $N-1$. Note, that this is similar to usual *double buffering* [Schachter83], but with one additional frame of latency.

8.2.3 Differential Updates

For realistic scene sizes, network bandwidth obviously is not high enough for sending the entire scene to each client for every frame. Thus, we only send differential updates from each frame to the next: Only those settings that have actually changed from the previous frame (e.g. the camera position, or a transformation of an object) will be sent to the clients. These updates are sent to the clients asynchronously: The server already streams partial updates of frame N while the application still has not finished specifying it, and while the clients are still working on frame $N-1$. Of course, this requires careful synchronization via multiple threads on both clients and server.

8.2.4 Task Prefetching

Upon completion of a task, a client sends its results to the server, and – in dynamic load balancing – has to wait for a new task to arrive. This delay (the network round-trip time) is usually the worst problem in dynamic load balancing, as it may result in the clients running idle waiting for work.

To cope with this problem, we have each client "prefetch" several tiles in advance. Thus, several tiles are 'in flight' towards each client at any time. As such, these tiles are being transferred across the network while the client is still busy with a different tile. Ideally, a new tile is just arriving every time a previous one is ready to be sent back to the server, as this would completely remove all tile communication latency while at the same tile maximizing the pool of yet-unassigned-tiles at the server. In practice, each client is currently prefetching about 4 tiles[11]. The optimal number of tiles in flight however depends on the ratio of tile compute cost to network latency, and might differ for other configurations.

[11] The exact number can currently be specified in a config file. Obviously, it should also be possible to dynamically modify and optimize this value during runtime. This however has not yet been investigated.

8.2.5 Multithreading

Due to a better cost/performance ratio, we usually use dual-processor machine. Using multithreading on each client then allows for sharing most data between these threads. This amortizes the communication cost for scene updates over two CPUs, as each received data item is immediately available to both CPUs.

In fact, each client uses several threads: As discussed above, all communication with the server is implemented asynchronously to the actual rendering. All of this communication is encapsulated in separate threads, which avoids network buffer overflows and at the same time minimizes response times.

As the RTRT core itself is thread-safe, each client can additionally run a varying number of ray tracing threads, which is currently either one thread for a single-CPU system, or two threads for a dual-CPU system[12].

8.3 Results

Touch RTRT/OpenRT has been successfully tested on many different hardware configurations, our standard configuration currently consists of a cluster of up to 24 dual processor AMD AthlonMP 1800+ PCs with 512 MB RAM each (48 CPUs total). The nodes are interconnected by a fully switched 100 Mbit Ethernet using a single Gigabit uplink to the master display and application server to handle the large amounts of pixel data generated in every frame. Note that this hardware setup is no longer state-of-the-art, as much faster processors and networks are already available[13].

The master machine is responsible for communicating with the application (see Section 10) and centrally manages the cluster nodes as described above. As the master machine runs *both* the application *and* the parallelization server, it should be a sufficiently powerful dual-CPU machine. To ensure optimal response times to client request, this machine should not be heavily loaded by application programs: If the application blocks the server, the server cannot react fast enough to client requents, resulting in bad load

[12] As a dual-Xeon system with hyperthreading actually features 4 virtual CPUs (2 virtual CPUs for each physical one), it might make sense to run four threads on these systems. So far however we are usually turning Hyperthreading off, as preliminary experiments have not shown any speedups, and the Linux kernels (version 2.4) currently still perform sub-optimal scheduling when this technology is turned on.

[13] The system is also frequently being used on a cluster of 12 dual 2.2GHz Pentium-IV Xeon PCs, and has also been successfully tested on newer 3GHz dual Pentium-IV PCs with gigabit interconnection, reaching more than 40 frames per second (at 640x480 pixels) on this platform.

balancing of the clients[14]. For example, having the server perform costly tone mapping computations every frame may significantly reduce the overall system performance. For similar reasons, the application should *also* avoid saturating the network with user data.

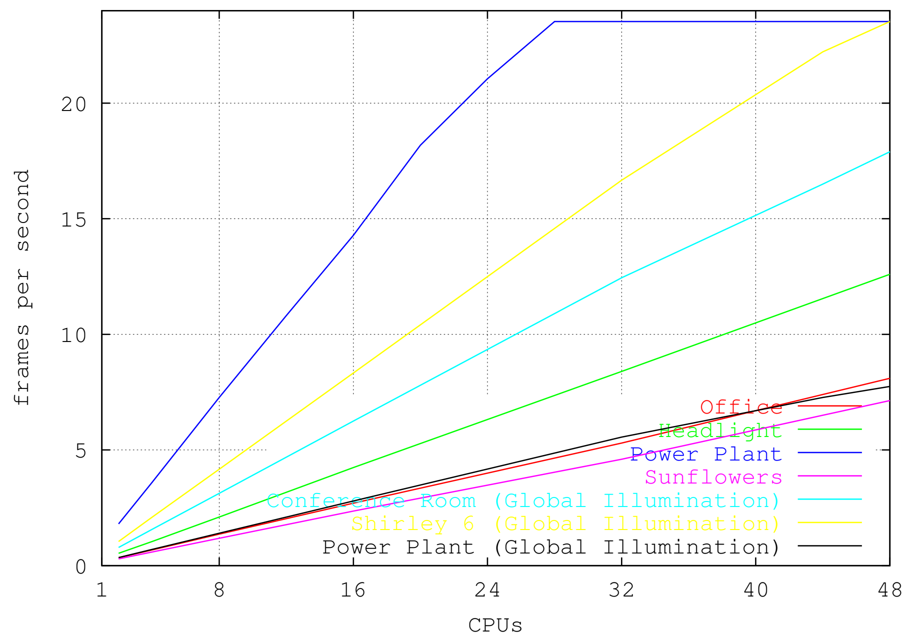


Figure 8.1: Scalability of the RTRT/OpenRT distributed ray tracing engine for different scenes and numbers of CPUs (at 640x480 pixels): PP/S: "Power Plant" scene with simple shading, PP/IGI: power plant with instant global illumination [Wald02b], SF: Oliver Deussens "Sunflowers". As can be seen, scalability is virtually linear up to a maximum framerate of ∼25 frames per second, at which the network connection to the server is saturated. On a newer GigaBit network installation, even more than 40 frames per second have been achieved. For the respective scenes see Figures 9.5, 11.3, and 13.12.

As can be seen in Figure 8.1 load balancing works fairly well for reasonably complex scenes and a good computation to latency ratio. Fortunately, many interesting applications — such as global illumination — require costly computations per pixel and thus scale well to 48 processors and more (see Figure 8.1 and Part III).

Though the system in practice scales linearly in the number of clients,

[14] Obviously the server itself runs multiple threads too. Even so a high server load negatively affects system performance.

this is only true up to an upper limit of roughly 20-25 frames per second at 640×480 pixels[15]. At this upper limit, the network is saturated by the number of pixels transferred to the server for display. Adding even more clients then results in reduced utilization of the clients, as these simply can't transfer their pixels fast enough to the server. However, this maximum framerate only depends on the servers network bandwidth, and on frame resolution.

Even at this maximum frame rate, the system still scales in quality: If the cost per pixel increases – e.g. by using more complex shading or by pixel-supersampling – more clients can be kept busy without increasing the network load. In Figure 8.1, this can for example be seen in the power plant scene: For simpler shading we start to see load balancing problems at 24 CPUs because at a resolution of 640x480 we no longer have enough jobs to keep all clients busy. For complex computations like global illumination – with a much higher cost per pixel – this problem occurs later.

8.4 Potential Improvements

While the system as descibed above already allows for many practical applications (see [Wald03a, Wald03e] and Section 11), there are still several issues that can be improved:

8.4.1 Maximum Frame Rate and the Server Bottleneck

Obviously, the biggest bottleck of the RTRT/OpenRT system is the network uplink to the server, which in its current architecture can easily become saturated by the computed pixel data. Obviously, this could be resolved by using better networking technology, like MyriNet [Forumb] or InfiniBand [Futral01], which however are still too costly for our "commodity hardware" approach. Fortunately, most applications focus on higher quality than higher frame rate, thus the limited frame rate currently is a minor problem.

However, even on the current networking technology, many improvements are still possible: First, higher effective bandwidth for gigabit ethernet can be achieved with newer GigaBit cards and their respective specially optimized

[15] On newer hardware (though still GigaBit ethernet) even 40 frames per second have been measured.

drivers[16]. Furthermore, the server can use trunking[17] of two GigaBit cards to transparently combine the bandwidth of two separate cards. On the software side, going to higher resolutions should allow for much larger packet sizes – and correspondingly achieve higher network throughput. Taken together, these measures should allow for coming much closer to the limit of PCI/33 bus bandwidth (rated at 133 megabytes per second). In theory, this should allow for frame rates of up to 30[18] frames per second over a PCI bus, and even much higher frame rates over already available busses like 64-bit/66MHz PCI, PCI-X [PCI-X], or PCI-Express [PCI Express]. At the time of this writing, the highest bandwidth that can be realized with commodity PC hardware is 10 gigabits/second. This would correspond to more than 300 frames per second, and would totally remove the server bandwith bottleneck.

8.4.2 Frame Assembly and Tone Mapping on the GPU

Obviously, the just mentioned frame rates of several dozen frames per second in full-screen resolutions would create a severe computational load just for copying the received pixels to their final image location. This is currently done by first copying the pixels to an intermediate buffer, from where they are later (once all the pixels of the current frame have been received) again copied to the frame buffer specified by the application (which may need to copy these pixels again in order to display them). This multiple buffering is neccessary due to the highly asynchronous system design and the interleaving of different frames, which results if tiles from different frames arriving "out of order".

For the frame rates mentioned above, this frame assembly load might easily reduce the servers response times for load balancing, and thus reduce scalability. This problem could be resolved by displaying the readily computed pixels by exploiting the high pixel-bandwidth of hardware-accelerated OpenGL. Similarly, bandwidth-intensive algorithms like tone mapping could be realized on the servers graphics card (also see e.g. [Artusi03, Goodnight03]).

[16] On our current (somewhat outdated) Gigabit network cards, we usually achieve only 250–400 Mbit/s. Newer Gigabit cards (with optimized drivers) have been measured to achieve much higher performance.

[17] Trunking is a process in which to physical network connections from the same network switch to the same PC (with two network cards, obviously) are virtually combined to one single network connection, i.e. the PC – though having actually two cards – gets only one single IP address. This enables to double the switch-PC bandwidth, without having to change any application programs. Most switches support trunking of at least some of their outgoing connections.

[18] $1280 \times 1024 pixels \times 3\frac{bytes}{pixel} =\sim 4\frac{MB}{frame}$, equalling $\sim 30fps$ at PCI bandwidth.

8.4.3 Server Bandwidth Reduction via Broadcasting

Currently, scene updates are replicated to every client. However, as all communication is performed via TCP/IP (which does not support broadcasting), no actual "hardware" broadcast is available. As a result, all scene data is currently sent *sequentially* to all clients one after another. This means that the network bottleneck at the server increases linearly with the number of clients, and – already for moderately costly scene updates – quickly saturates the server bandwidth as many clients are to be used. This bandwidth problem could probably be fixed by using network broadcasting (UDP) [Stevens98]. UDP communication however is not reliable, and would thus require a completely different communication architecture.

Note that the situation might be completely different when completely abandoning Ethernet technology for the favor or e.g. MyriNet [Forumb] or Infiniband [Futral01].

8.4.4 Geometry Shaders

The problem of costly network communication for scene updates could also be alleviated by employing *geometry shaders*, i.e. small programs that procedurally generate the updated geometry. This would be similar to OpenGL vertex shaders, but somewhat more flexible. Running the geometry shaders directly on the clients would remove the need to communicate these scene updates over the network. The potential and restrictions of this approach however have not yet been sufficiently evaluated.

8.4.5 Support for Virtually Unlimited Model Sizes

In its current form, the parallelization framework stores a complete replica of the whole scene data base on each client. This not only limits the size of models that can be rendered efficiently, but also unnecessarily increases the network communication by sending data to clients that might not actually need this data for its current view.

As has been shown in a prototype implementation [Wald01c], this problem can be solved by having the clients load model data "on demand" from a distributed (i.e. decentralized) model storage, which allows for supporting virtually unlimited model sizes. Though this experiment is already somewhat outdated[19], it seems beneficial to re-introduce several of the ideas outlined

[19] Due to a higher efficiency of the ray tracing core, the power plant scene as used in this experiment can now be rendered *directly*, as it now fits completely into the memory of a single PC or notebook.

therein. This, however, would require severe changes to the current parallelization framework, and would also bear several implications on the API used to drive the rendering engine.

Most importantly, the original prototype implementation has shown that reordering of rays can – and has to – be used for hiding the latencies that result from demand-loading the data (which essentially are nothing but extremely costly "cache" misses). This requires means of suspending and resuming rays which unfortunately is not trivial in a fully general ray tracing framework: For example, a ray scheduled to be suspended might actually be somewhere down a complete ray tree[20]. A framework for suspending and resuming all kinds of rays will probably require a slightly limited (separable) lighting model as proposed by Pharr et al. [Pharr97].

Furthermore, the prototype system has shown that a centralized data storage is likely to become a network bottleneck and thus is problematic for a scalabile system. This is especially problematic because a centralized data storage is the most natural for a system in which the actual application is only running on one single PC, and in which all other machines are but "dumb" rendering clients.

8.4.6 Automatic Choice of Performance Parameters

In the current system, the optimal frame rate requires the correct choice of several parameters, like the amount of prefetched tiles, threads per client, number of clients, and tile size. This currently works only semi-automatically, and may (in rare cases) even result in *less* performance when adding more clients (e.g. if the system is bound by scene update cost). Thus, it would be highly beneficial if an optimal set of parameters could be deduced automatically during runtime.

8.5 Conclusions

The parallelization framework inside the RTRT/OpenRT engine allows for efficiently scaling the RTRT kernel performance to several PCs combined with commodity network technology, thereby achieving interactive performance even for massively complex models and highly complex lighting situations that require costly shading and/or many secondary rays per pixel.

[20] This was not a problem in the prototype implementation, which only supported a hard-coded lighting model allowing for primary rays and one single shadow and reflection ray only.

The distribution process is completely transparent to both application and shaders. The application runs only on the master machine and interacts with the rendering engine only through the OpenRT API (see Section 10). The shaders are loaded dynamically on the clients and compute their pixel values independently of the application. While there is still room for improvement, the RTRT/OpenRT parallelization framework is already highly efficient, and allows for a wide range of applications [Wald03a, Wald03e]. For a brief overview of these applications, see Chapter 11.

Chapter 9
Handling Dynamic Scenes

"The time spent constructing the hierarchy tree should more than pay for itself in time saved rendering the image"

Timothy L. Kay, James T. Kajiya "Ray Tracing Complex Scenes" [Kay86] (in 1986!)

Even though ray tracing is a relatively old and well-understood technique, its use for interactive applications is still in its infancy. Several issues of interactive applications are all but fully solved. Especially the handling of dynamic scenes in an interactive context so far has received few attention by ray tracing researchers. Ray tracing research so far almost exclusively concentrated on accelerating the process of creating a *single* image, which could take from minutes to hours. Most of these approaches relied on doing extensive preprocessing by building up complex data structures to accelerate the process of tracing a ray.

Before realtime ray tracing, the time used for building an index structure such as kd-trees was insignificant compared to the long rendering times, as this preprocessing was then amortized over the remainder of a frame. Thus preprocessing times of several seconds to a few minutes could easily be tolerated in order to build a high-quality acceleration structure for an offline renderer. As long as the scene remains static, the same trick also worked for "interactive" ray tracing systems as described before – the acceleration structure was built once in an offline preprocessing step, and was then reused for all the remaining frames[1].

[1] Even though the scene itself has to remain static in this approach, it is still possible to arbitrarily change camera, material properties, shaders, and light source configurations in a scene.

In dynamic scenes, however, this trick no longer works, as each new frame requires a new acceleration structure. Building this data structure for every frame then becomes a bottleneck, as this "preprocessing" alone would often exceed the total time available per frame in an interactive setting.

Even worse, this preprocessing phase cannot easily be parallelized: Though tracing the rays can be parallelized trivially once each client has access to scene and acceleration structure, the operations for building the acceleration structure have to be performed on each client, thereby incurring a non-parallelizable cost factor. As a result, any time spent on dynamic updates becomes extremely costly especially for parallel (distributed) interactive ray tracing systems[2]. This poses a major problem for ray tracing dynamic scenes, as virtually all of todays interactive ray tracing (e.g. [Wald01c, Parker99b, DeMarle03, Wald03e]) systems have to rely on massive parallelization to achieve interactive frame rates.

Therefore, it is not surprising that all of those systems have in common that they mainly concentrate on the actual ray tracing phase and do not target dynamic scenes. Without methods for interactively modifying the scene, however, interactive ray tracing will forever be limited to simple walkthroughs of static environments, and can therefore hardly be termed truly interactive, as long as real *interaction* between the user and the environment is not impossible. In order to be truly interactive, ray tracing *must* be able to efficiently support dynamic environments. As such, efficient handling of dynamic scenes is probably one of the biggest challenges for realtime ray tracing.

9.1 Previous Work

Some methods have been proposed for the case where predefined animation paths are known in advance (e.g. [Glassner88, Gröller91]). These however are not applicable to our target setting of totally dynamic, unpredictable changes to the scene in which the motion of objects is not known in advance. Little research is available for such truly interactive settings. This research will be reviewed below.

First of all, excellent work on ray tracing in dynamic environments has recently been performed by Lext et al. with the BART project [Lext00], in which they provide an excellent analysis and classification of the problems

[2] As an example, consider a system that spends only 5% of its time on dynamic scene updates. Parallelizing this system on 19 CPUs ($\frac{1}{5\%}-1$) results in each node spending half its time on scene updates, and in a speedup of only 10 for 19 CPUs (i.e. a utlization of only $\frac{10}{19} \sim 50\%$)!

arising in dynamic scenes. Based on this analysis, they proposed a representative set of test scenes (see Figure 9.1) that have been designed to stress the different aspects of ray tracing dynamic scenes. Thus, the BART benchmark provides an excellent tool for evaluating and analyzing a dynamic ray tracing engine. For future research on dynamic ray tracing, the BART benchmark suite might well play the same role that Eric Haines' "SPD Database" [Haines87] played for offline rendering.

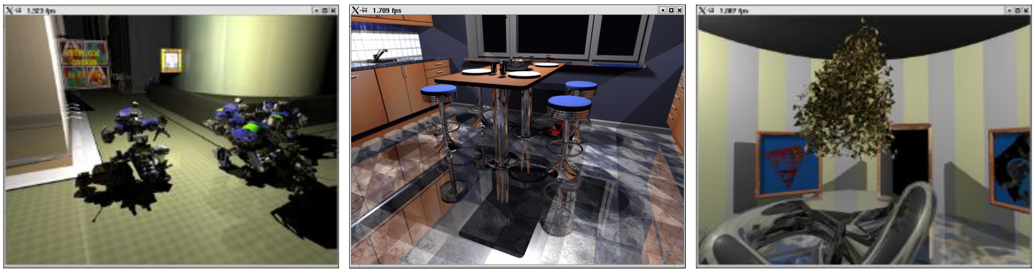


Figure 9.1: Some example screen-shots from the BART benchmark: (a) "robots", where 10 robots (each consisting of 16 independently moving body parts) are walking through a city model; (b) "kitchen", in which a small toy car is driving through a highly specular kitchen scene; and (c) "museum", where a certain amount of reflective triangles is animated incoherently to form several different shapes. The number of triangles in the museum scene can be configured from a few hundred to some hundred thousand triangles.

In their analysis, Lext et al. have classified the behavior of dynamic scenes into two inherently different classes: Hierarchical motion, and unstructured motion. In *hierarchical motion*, the animation is described by having the primitives in a scene organized into several groups that are transformed hierarchically. While different groups may move independently of all other groups, all primitives in the same group are always subject to the same, usually affine, transformation[3]. The other class is *unstructured motion*, where each triangle moves without relation to all others. For example, the robots scene in Figure 9.1a is a good example of hierarchical motion, as there is no dynamic behavior except for hierarchical translation and rotation of the different robots' body parts. In contrast to this, the museum scene (Figure 9.1c) features many incoherently moving triangles, and as such is a good example for unstructured motion. For a closer explanation of the different kinds of motion, see the BART paper [Lext00].

Though the BART paper provides an excellent analysis of dynamic ray

[3] Affine transformation are not limited to translation and rotation only, but also include e.g. shearing or scaling.

tracing, it did not attempt to propose any practical algorithms or solutions to the problems. So far, few people have worked on this topic. In a first step, Parker et al. [Parker99b] kept moving primitives out of the acceleration structure and checked them individually for every ray. This of course is only feasible for a small number of moving primitives.

Another approach would be to efficiently update the acceleration structure whenever objects move. Because objects can occupy a large number of cells in an acceleration structure this may require costly updates to large parts of the structure for each moving primitive (especially for large primitives, which tend to overlap many cells). To overcome this problem, Reinhard et al. [Reinhard00] proposed a dynamic acceleration structure based on a hierarchical grid. In order to quickly insert and delete objects independently of their size, larger objects are kept in coarser levels of the hierarchy. With this approach, objects always cover approximately a constant number of cells, thus updating the acceleration structure can be performed in constant time. However, their method resulted in a rather high overhead, and also required their data structure to be rebuilt once in a while to avoid degeneration. Furthermore, their method mainly concentrated on unstructured motion, and is not well suited for hierarchical animation.

Recently, Lext et al. [Lext01] proposed a way for quickly reconstructing an acceleration structure in a hierarchically animated scene by transforming the rays to the local object coordinate systems instead of transforming the objects and rebuilding their acceleration structures. Though their basic idea is similar to the way that our method handles hierarchical animation, to our knowledge their method so far has never been applied in an interactive context.

9.2 A Hierarchical Approach to Handling Dynamic Scenes

Essentially, our approach to handling dynamic scenes is motivated by the same observations as Lext et al. [Lext01] of how dynamic scenes typically behave: Usually large parts of a scene remain static over long periods of time. Other parts of the same scene undergo well-structured transformations such as rigid body motion or affine transformations. Yet other parts are changed in a totally unstructured way.

All these kinds of motion are fundamentally different. Even worse, many scenes actually contain a mix of all these different kinds of motion. It is unlikely that a single method can handle all these kinds of motion equally well. Because of this, we prefer an approach in which the different kinds of motion are handled with different, specialized algorithms that are then combined into a common architecture. To do this, geometric primitives are organized in separate *objects* according to their dynamic properties. Each of the three kinds of objects – static, hierarchically animated, and those with unstructured motion – is thus treated differently: Static objects will be handled as before, hierarchically animated objects are handled by transforming rays rather than the object, and objects with unstructured motion are handled by specially optimized algorithms for quickly rebuilding the affected parts of the data structure. Each object has its own acceleration structure and can be updated independently of the rest of the scene. These independent objects are then combined in a hierarchical way by organizing them in an additional top-level acceleration structure that accelerates ray traversal between the objects in a scene (see Figure 9.2).

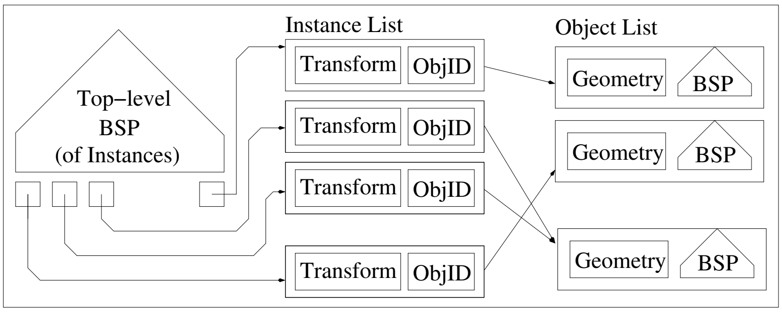


Figure 9.2: Two-level hierarchy as used in our approach: A top-level BSP contains references to instances, which contain a transformation and a reference to an object. Objects in turn consist of geometry and a local BSP tree. Multiple instances can refer to the same object with different transformations.

9.2.1 Building the Hierarchy

To enable this scheme, all triangles that are subject to the same set of transformations (e.g. all the triangles forming the head of the animated robot in Figure 9.3) must be grouped by the application into the same object.

Note that we explicitly do *not* attempt to perform this grouping automatically. Instead, this grouping has to be performed by the application that drives the ray tracer. Though this somewhat shifts the problem to the application, the application itself has the information about the motion of every part of the scene in its internal scene-graph, and can typically perform this classification without major effort. In fact, most applications already do

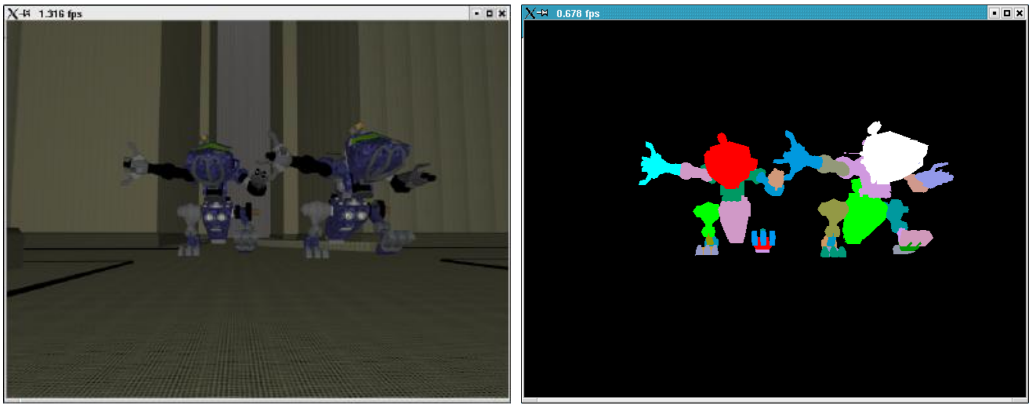


Figure 9.3: Grouping of triangles into objects for hierarchical animation. Triangles subject to the same hierarchical transformations are grouped into the same object. (a) Snapshot from the BART robots scene, (b) Same snapshot, with color-coded objects. Triangles with the same color belong to the same object.

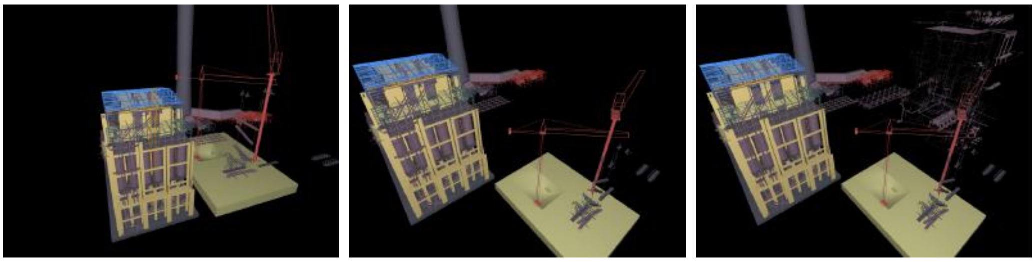


Figure 9.4: Snapshots of an interactive session in which complex parts of the 12.5 million triangle "UNC power plant" model are being moved interactively with our method: a) the original powerplant, b) moving the powerplant and the construction side apart, and c) moving part of the internal structure (the cool and warm water pipes, totalling a few million triangles!) out of the main structure).

this for OpenGL rendering, as the same grouping of objects is required for efficiently using OpenGL display lists (also see the discussion of the OpenRT API in Section 10). However, the actual grouping of objects into objects has a higher influence on rendering performance than for OpenGL display lists. As such, it is important to perform this grouping with extreme care in order to achieve good performance.

In the following, we will shortly describe how these different kinds of objects are treated, and how the top-level index structure is built and traversed.

9.3 Static and Hierarchical Motion

For *static objects*, ray traversal works as before by just traversing the ray with our usual, fast traversal algorithm.

For *hierarchically transformed objects*, we do not actually transform the geometry of the object itself, but rather store this transformation with the object, and inversely transform the rays that require intersection with this object[4].

For both static objects and for those with hierarchical motion, the local BSP tree must only be built once directly after object definition. Thus, the time for building these objects is not an issue, thereby allowing for the use of sophisticated and slow algorithms for building high-quality acceleration structures for these objects (see Section 7.3).

Obviously the transformation slightly increases the per-ray cost. However, this transformation has to be performed only once for each dynamic object visited by a ray, and is as such tolerable. This increased per-ray cost then totally removes the reconstruction cost for hierarchically animated objects, as all that is required for transforming the object is to update its transformation matrix. This is especially important as this kind of motion is usually the most common form in practical scenes. Furthermore, not having to rebuild any BSP trees make the update-cost for hierarchically transformed objects independent of object size. As such, this way of handling hierarchical animation can be used very efficiently even in extremely complex scenes. For example, Figure 9.4 shows how a complex part of the 12.5 million triangle "UNC power plant" is being moved in an interactive session.

9.3.1 Instantiation

Being able to handle objects that are subject to a transformation, the presented approach as a side effect also allows for "instantiation", i.e. using multiple instances of the same object: Parts of a scene (e.g. one of the sunflowers in Figure 9.5) can re-used several times in the same scene by creating several *instances* of this object. An instance then consists only of two properties: a reference to the original model, and a transformation matrix that the instanced object is subject to. Thus, even highly complex scenes can be stored with a small memory footprint, which in turn allows for efficiently rendering even massively complex scenes at interactive rates. As an example, Figure 9.5 shows a slight modification of Oliver Deussen's "Sunflowers" scene, which consists of several large trees with millions of triangles each,

[4] Note that this way of handling is similar to the approach of Lext et al. [Lext01].

plus 28,000 instances of 10 different sunflower models with roughly 36,000 triangles each. While only one kind of tree and 10 kinds of sunflowers have to actually be stored, in total roughly one billion triangles are potentially visible. By changing the transformation matrices of the instances, each object can be manipulated interactively while the scene renders at about 7 fps on 24 dual processor PCs at video resolution (see Table 9.4). Note that this performance can be achieved even tough a large number of rays needs to be cast in this scene: The leaves of both sunflowers and trees are modeled with transparency textures, which results in many rays for computing transparency and semi-transparent shadows.

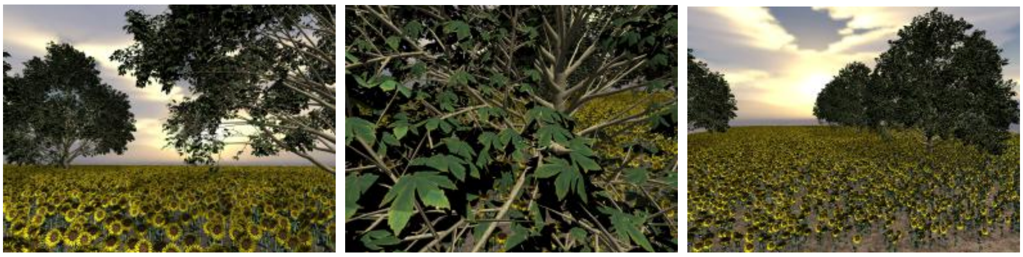


Figure 9.5: Instantiation: The "Sunflowers" scene consists of roughly 28,000 instances of 10 different kinds of sunflowers with 36,000 triangles each together with several multi-million-triangle trees. The whole scene consists of roughly one billion triangles. The center image shows a closeup of the highly detailed shadows cast by the sun onto the leaves. All leaves contain textures with transparency which increase the number of rays needed for rendering a frame. The whole scene renders at roughly 7 fps on 24 dual PCs at video resolution. All objects including the sun can be manipulated interactively.

9.4 Fast Handling of Unstructured Motion

While this simple trick of transforming rays instead of triangles elegantly avoids any reconstruction cost for hierarchical motion, it does not work for *unstructured motion*, as there the acceleration structure potentially has to be rebuilt for every frame. Even so, if triangles under unstructured motion are kept in a separate object, the BSP reconstruction cost can be localized to only those triangles that have actually been transformed. The local acceleration structures of such objects are discarded and rebuilt from the transformed triangles whenever necessary. Even though this process is costly, it is only required for objects with unstructured motion and does not affect any of the other objects. Obviously, only those objects have to be rebuilt whose primitives have actually be modified in the respective frame. Furthermore, it is possible to perform this reconstruction of dynamic objects lazily *on demand*, i.e. only once a ray actually demands intersection with that updated object. As such, the occlusion-culling feature of ray tracing also applies to the reconstruction of dynamic objects, as occluded objects do not have to be rebuilt.

The algorithms for creating highly optimized BSP trees as discussed in Section 7.3 may require several seconds even for moderately complex objects. Thus, they are not applicable to unstructured motion, for which the object BSP has to be rebuilt every frame (and thus in fractions of a second). For these cases we trade traversal performance for construction speed by using less expensive, simple heuristics for BSP plane placement, which allows for a high-performance implementation of the construction process.

9.4.1 Using less costly BSP Construction Parameters

Furthermore, we use less expensive quality parameters for the BSP plane placement heuristics. For example, a particularly important cost factor for BSP tree construction is the subdivision criterion of the BSP. As described in Section 7.3, this criterion typically consist of a maximum tree depth and a target number of triangles per leaf cell. Subdivision continues on cells with more than the target number of triangles up to the maximum depth. Typical criteria specify 2 or 3 triangles per cell and usually result in fast traversal times – but also in deeper BSPs, which are more costly to create. Particularly costly are degenerate cases, in which subdivision can not reduce the number of triangles per cell, for example if too many primitives occupy the same point in space, e.g. at vertices with a valence higher than the maximum numbers of triangles.

In order to avoid such excessive subdivisions in degenerate regions, we modified the subdivision criterion (for unstructured object BSPs): The deeper the subdivision, the more triangles will be tolerated per cell. We currently increase the tolerance threshold by a constant factor for each level of subdivision. Thus, we generally obtain significantly lower BSP trees and larger leaf cells than for static objects. Though this of course slows down the traversal of rays hitting such objects, this slowdown is usually more than made up by the significantly shorter construction time. Furthermore often only few rays hit such objects with unstructured motion and are affected by this slowdown, so using a slower BSP tree for those rays is tolerable. With the described compromises on BSP construction, unstructured motion for moderate-sized objects can be supported by rebuilding the respective object BSP every frame.

9.5 Fast Top-Level BSP Construction

As mentioned before, all kinds of objects – static, hierarchically animated, and those with with unstructured motion – are hierarchically combined in an additional top-level acceleration structure. For this top-level structure, we also use a kd-tree, and as such can use exactly the same algorithms for traversing this top-level tree than for the object BSPs, except that visiting a voxel now requires to intersect objects rather than triangles. While traversing this top-level BSP thus requires only minor changes to the original implementation, this is not the case for the construction algorithm. A scene can easily contain hundreds or thousands of instances (see Figures 9.6 and 9.5), and a straight-forward approach would be too costly for interactive use. On the other hand, the top-level BSP is traversed by *every* ray, and thus few compromises on BSP quality can be made for the top-level BSP.

Fortunately, the task of building the top-level BSP is simpler than for object BSPs: Object BSPs require costly triangle-in-cell computations, careful placement of the splitting plane, and handling of degenerate cases. The top-level BSP however only contains instances represented by an axis-aligned bounding box (AABB) of its transformed object[5].

Considering only the AABBs, optimized placement of the splitting plane becomes much easier, and any degenerate cases can be avoided.

For splitting a cell, we follow several observations:

1. It is usually beneficial to subdivide a cell in the dimension of its maximum extent, as this usually yields the most well-formed cells [Havran01].

2. Placement of the BSP plane only makes sense at the boundary of objects contained within the current cell. This is due to the fact that the cost-function can be maximal only at such boundaries [Havran01].

3. It can been shown that the optimal position for the splitting plane lies between the cells geometric center and the object median [Havran01]

Following these observations, the BSP tree can be built such that it is both suited for fast traversal by optimized plane placement, and can still be built quickly and efficiently: For each subdivision step, we try to find a splitting plane in the dimension of maximum extent (observation 1). As potential

[5] While the object itself already has an axis-aligned bounding box, this AABB is not necessarily axis-aligned any more when subject to a transformation. As such, we conservatively build the AABB of the instance by building a new instance AABB out of the transformed vertices of the objects AABB. While this somewhat overestimates the actual instance bounds, it is much less costly than computing the correct AABB by transforming all vertices.

splitting planes, only the AABB borders will be considered (observation 2). To find a good splitting plane, we first split the cell in the middle, and decide which side contains more objects, i.e. which one contains the object median. From this side, we choose the object boundary closest to the center of the cell. Thus, the splitting plane lies in-between cell center and object median, which is generally a good choice (observation 3).

As each subdivision step removes at least one potential splitting plane, termination of the subdivision can be guaranteed without further termination criteria. Degenerate cases for overlapping objects cannot happen, as only AABB boundaries are considered, and not the overlapping space itself. Choosing the splitting plane in the described way also yields relatively small and well-balanced BSP trees. Thus, we get a top-level BSP that can be traversed reasonable quickly, while still offering a fast and efficient construction algorithm.

```
BuildTree(instances,voxel)
for d = x,y,z in order of maximum extent
    P = $\{i.min_d, i.max_d | i \in instances\}$
    if ($\|P\|$ = 0) continue;
    c = center of voxel
    if (more instances on left side of c than on right)
        p = max($\{p \in P | p < c\}$)
    else
        p = min($\{p \in P | p >= c\}$)
    Split Cell (instances,cell) in d at p into
            (leftvox,leftinst),(rightvox,rightinst)
    l = BuildTree(leftinst,leftvox);
    r = BuildTree(rightinst,rightvox);
    return InnerNode(d,p,l,r);
end for
# no valid splitting plane found
return Leaf(instances)
```

9.5.1 High-Quality Top-level BSPs

Instead of this simplified BSP construction algorithm, it is also possible to use a surface area heuristic (see Section 7.3) for the top-level BSP tree. The main problem with this approach is the question how to best estimate the cost for intersecting the object. As the respective object BSPs contain only triangles, the cost for each half voxel created by a split can be safely estimated to be mostly linear in the number of triangles on each side. For the top-level

BSP however, each side contains objects of different size, for which the cost is hard to estimate.

However, the bigger problem with a surface area heuristic for the top-level BSP tree is its relatively high construction cost. While this may be neglectable for a few dozen objects, it currently becomes too expensive for a few hundred instances. As such, using an SAH would only make sense for a small number of instances as long as no fast implementations of an SAH tree builder are available. Though RTRT/OpenRT has an implementation of both SAH and the above mentioned algorithm, by default we use the simple and fast-to-build version as described above.

9.6 Fast Traversal

Once both the top-level BSP tree and all the object BSPs are built, each ray first starts traversing this top level structure. As soon as a voxel is found, the ray is intersected with the objects in the leaf by simply traversing the respective objects local acceleration structures. Once all BSPs are built, within both top-level BSP and within each object traversal is identical to traditional ray tracing in a static environment. Consequently, we use exactly the same algorithms and data structures for building and traversing that acceleration structure as we already described previously for the static case (see Section 7). For the top-level BSP, the only difference is that each leaf cell of the top-level BSP tree contains a list of instance IDs instead of triangle IDs. Only minor changes have been required to implement this modified traversal code.

As with the original implementation, a ray is first clipped to the scene bounding box and is then traversed iteratively through the top-level BSP tree. As soon as it encounters a leaf cell, it sequentially intersects the ray with all instances in this cell: For each instance, the ray is first transformed to the local coordinate system of the object, then clipped to the correct AABB of the object, and finally traversed through its acceleration structure.

9.6.1 Mailboxing

Typically, the bounding volumes of different instances will overlap. In order to avoid having to intersect a ray with the same object multiple times, mailboxing [Amanatides87, Kirk91, Havran01] is very important to use for the top-level BSP tree. While the benefit of mailboxing for the triangle for an object BSP is rather small, the high cost of intersecting the same object several times clearly justifies the use of mailboxing for the top-level BSP.

9.6.2 SSE Traversal

As our traversal and intersection algorithms do not require normalized ray directions, transforming a ray is relatively cheap, as no costly re-normalization of the transformed rays is necessary. The ray-matrix multiplications themselves can very efficiently be done using SSE [Intel02b].

Of course, our method also works with our fast SSE packet traversal code. The only caveat is that this packet traversal code requires that all rays directions in a packet have the same signs, as was described in Section 7.2.3. As a rotation can change these signs, an additional check has to be done after each transformation, and such a packet might sometimes have to be split up before intersecting an object. However, this special case is trivial and cheap to detect and work around, and happens rarely[6]. As such, its cost in practice is negligible.

9.7 Experiments and Results

The described framework is rather straightforward to implement and use as long as a shared-memory system (e.g. a dual-CPU PC) is available. However, the situation of dynamic ray tracing gets much more problematic for non-shared memory systems, as such systems often contain many non-scalable cost factors, such as communicating scene updates to the client, or having to rebuild parts of the hierarchy on every client.

As the described framework has been especially designed for performing well on loosely coupled (i.e. non-shared memory) clusters of workstations, it is of major importance to investigate the scalability of our method. To allow for representative results, we have chosen to use a wide range of experiments and test scenes. Therefore, we have chosen to use the BART benchmark scenes [Lext00], which represent a wide variety of stress factors for ray tracing of dynamic scenes. Additionally, we use several of the scenes that we encountered in practical applications [Wald02b], and a few custom-made scenes for stress-testing. Snapshots of these test scenes can be found in Figure 9.6.

All of the following experiments have been performed on a cluster of dual AMD AthlonMP 1800+ machines with a FastEthernet (100Mbit) network connection. The network is fully switched with a single GigaBit uplink to a dual AthlonMP 1700+ server. The application is running on the server and is totally unaware of the distributed rendering happening inside the render-

[6] For primary rays, there is only one row and one column of pixels in which that can happen at all.

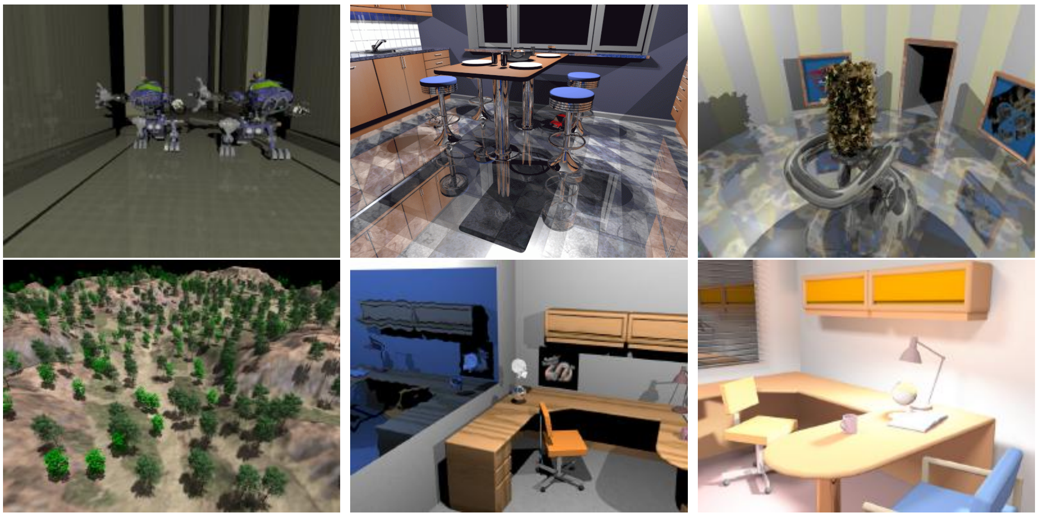


Figure 9.6: Several example frames from some of our dynamic scenes. a.) "BART robots" contains roughly 100,000 triangles in 161 moving objects, b.) "BART kitchen", c.) "BART museum" with unstructured motion of several thousand triangles. Note how the entire museum reflects in these triangles. d.) The "terrain" scene uses up to 661 instances of 2 trees, would contain several million triangles without instantiation, and also calculates detailed shadows. e.) The "office" scene in a typical ray tracing configuration, demonstrating that the method works fully automatically and completely transparently to the shader. f.) Office with interactive global illumination.

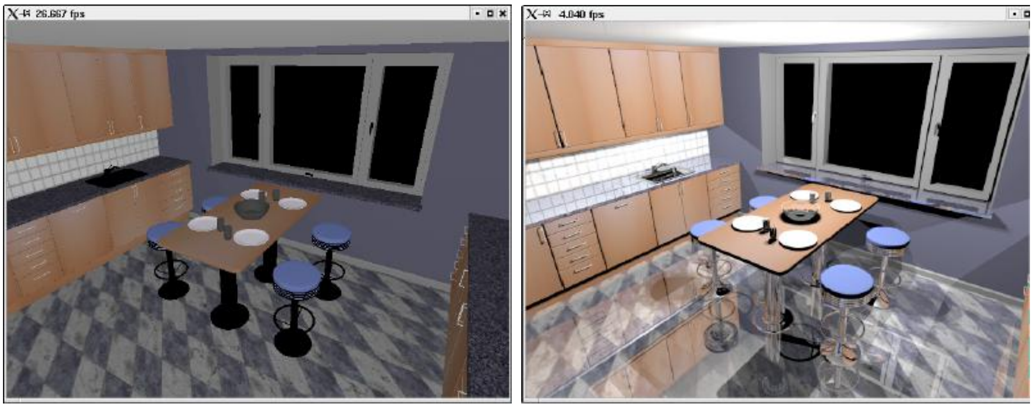


Figure 9.7: Two snapshots from the BART kitchen. a.) OpenGL-like ray casting at > 26 *fps on 32 CPUs. b.) full-featured ray tracing with shadows and 3 levels of reflections, at* > 7 *fps on 32 CPUs.*

ing engine. It manages the geometry in a scene graph, and transparently controls rendering via calls to the OpenRT API (see Chapter 10). While the application itself may internally use a scene-graph with multiple nested hierarchy levels, the OpenRT library internally "flattens" this multi-level scene graph to the two-level organization as described above (see Figure 9.2).

In the following experiments, all examples are rendered at video resolution of 640 × 480 pixels. Ray tracing is performed with costly programmable shaders featuring shadows, reflections and texturing.

9.7.1 BART Kitchen

The kitchen scene contains hierarchical animation of 110.000 triangles organized in 5 objects. It requires negligible network bandwidth and BSP construction overhead. Overlap of bounding boxes may results in a certain overhead, which is hard to measure exactly but is definitely not a major cost factor[7].

The main cost of this scene is due to the need for tracing many rays to evaluate shadows from 6 point lights. There is also a high degree of reflectivity on many objects. Due to fast camera motion and highly curved objects (see Figure 9.7), these rays are rather incoherent. However, these aspects are completely independent of the dynamic nature of the scene and are handled efficiently by our system.

We achieve interactive frame rates even for the large amount of rays to be shot. A reflection depth of 3 results in a total of 3.867.661 rays/frame. At a measured rate of 912.000 rays traced per second and CPU in this scene, this translates to a frame rate of 7.55 fps on 32 CPUs. As can be seen in Table 9.1, scalability is almost linear – using twice as many CPUs results in roughly twice the frame rate.

CPUs	2	4	8	16	32
OpenGL-like	3.2	6.4	12.8	25.6	> 26
Ray Tracing	0.47	0.94	1.88	3.77	7.55

Table 9.1: Scalability in the kitchen scene in frames/sec.

9.7.2 BART Robots

The robots scene features a game-like setting with 16 animated robots moving through a city. The scene consists of 161 objects: 16 robots with 10 animated

[7] Note that the robot, museum, kitchen, and terrain scenes are only available in a dynamic version, and can thus not be compared to a static version.

body parts each, plus one object for the surrounding city. All dynamic motion is hierarchical with no unstructured motion at all. Therefore, the BSP trees for all objects have to be built only once, and only the top-level BSP have to be rebuilt for every frame.

Using the algorithms described above, rebuilding the top-level BSP is very efficient and takes less than one millisecond. Furthermore, updating the transformation matrices requires only a small network bandwidth of roughly 20 kb/frame for each client.

CPUs	2	4	8	16	32
OpenGL-like	2.8	5.55	10.8	21	> 26
Ray Tracing	0.54	1.07	2.15	4.3	8.6

Table 9.2: Scalability in the robots scene in frames/sec.

With such a small transmission and reconstruction overhead, we again achieve almost-linear scalability (see Table 9.2) and high rendering performance. Using 32 CPUs, we achieve a frame rate of 8 frames per second. Again, the high cost of this scene is due to the large number of reflection and shadow rays. Using a simple OpenGL-like shader (see Figure 9.8) results in frame rates of more than 26 frames per second.

9.7.3 BART Museum

The museum has been designed mainly for testing unstructured motion and is the only BART scene featuring non-hierarchical motion. In the center of the museum, several triangles are animated on predefined animation paths to form differently shaped objects. The number of triangles undergoing unstructured motion can be configured to 64, 256, 1k, 4k, 16k, or 64k. Even though the complete animation paths are specified in the BART scene graph, we do not make use of this information. User controlled movement of the triangles – i.e. without knowledge of future positions – would create the same results.

This scene also requires the computation of shadows from two point lights as well as large amounts of reflection rays. All of the moving triangles are reflective and incoherently sample the whole environment (see Figure 9.9). As the dynamic behavior of a scene is completely transparent to the shaders, integrating all these effects does not require any additional effort except for the cost for tracing the rays.

As expected, unstructured motion gets costly for many triangles. Building the BSP tree for the complex version of 64k triangles already requires

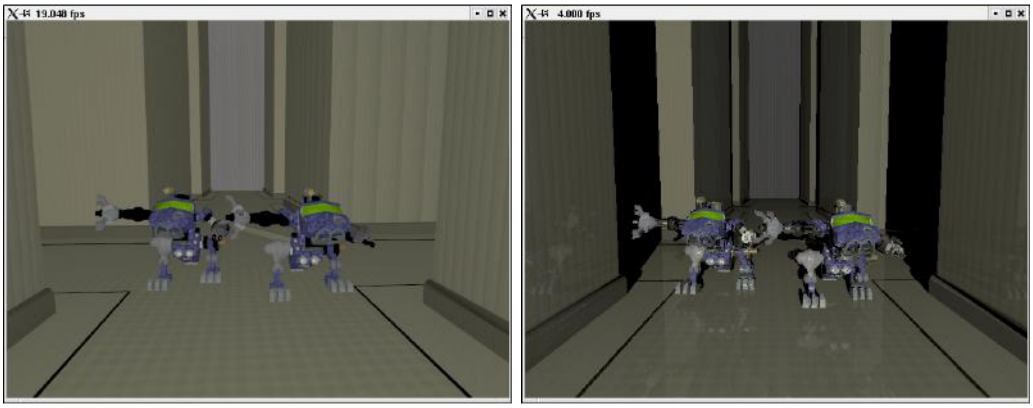


Figure 9.8: BART robots: 16 robots consisting of 161 objects rendered interactively. a.) Ray casting at > 26 fps on 32 CPUs. b.) Full ray tracing at > 8 fps at 32 CPUs.

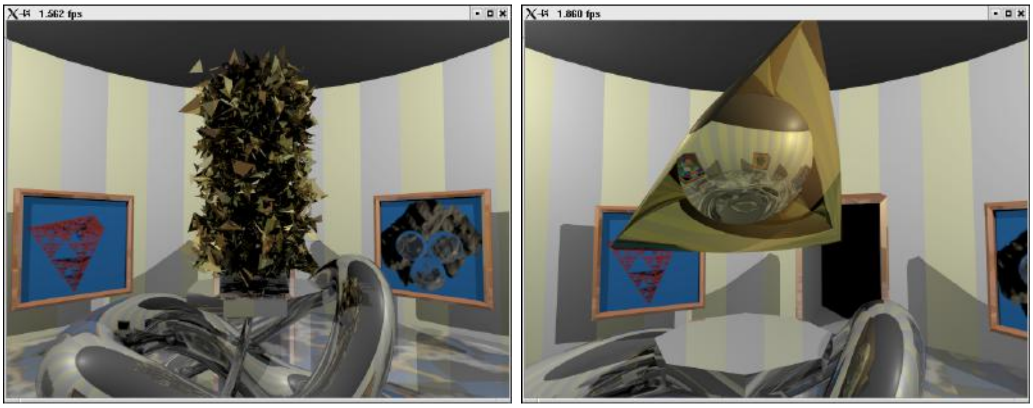


Figure 9.9: Unstructured motion in the BART museum: Up to 64,000 triangles are moving incoherently through the museum. Note how the triangles reflect the entire environment.

more than one second (see Table 9.3). Note, however, that our current algorithms for building object BSPs still leave plenty of room for further optimizations.

num tris	64	256	1k	4k	16k	64k
reconst.time	1ms	2ms	8ms	34ms	0.1s	> 1s
bandwidth/client	6.4k	25.6k	102k	409k	1.6M	6.5M

Table 9.3: Unstructured motion in different configurations of the museum scene. Rows specify reconstruction time for the top-level BSP, and data sent to each client for updating the triangle positions.

Furthermore, the reconstruction time is strongly affected by the distribution of triangles in space: In the beginning of the animation, all triangles are equally and almost-randomly distributed. This is the worst case for BSPs, which are best at handling uneven distributions, and construction is consequently costly. Furthermore, the randomly distributed triangles form many singularities when intersecting themselves, which is extremely bad for typical BSP trees. During the animation, the triangles organize themselves to form a single surface. At this stage, reconstruction time is much faster. Note that the numbers given in Table 9.3 are taken at the beginning of the animation, and are thus worst-case results.

9.7.3.1 Network Bottleneck

Apart from raw reconstruction cost, significant network bandwidth is required for sending all triangles to every client. Since we use reliable unicast (TCP/IP) for network transport, using 4096 triangles and 16 clients (32 CPUs), requires to transfer roughly 6.5 Mb (16 clients $\times 408kb$, see Table 9.3) – for *every frame*. Though in theory this does not yet totally saturate the network, the network load is not equally distributed over time: Network bandwidth is especially high at the beginning of each frame, when all the scene updates have to be communicated to the clients. During this time, the network is already completely saturated when sending 16k triangles to the clients, implying that the performance of the server is already significantly affected during this time. Consequently, we can no longer scale linearly any more when dynamically updating more than a few thousand triangles (see Table 9.4).

9.7.3.2 "Geometry Shaders"

Note that this problem would be significantly reduced on a shared-memory platform, or even with the availability of a reliable hardware multicast. On a cluster configuration, the update problem could probably also be solved by using "geometry shaders" that can generate the triangles directly on the clients. Though this can only be used for a limited set of applications, it would already allow for many important and interesting applications. So far however this approach has not yet been sufficiently investigated.

Due to the discussed problems – high communication cost for the scene updates and high reconstruction cost for the dynamic object's BSP – the museum scene is the most problematic of all the scenes encountered so far. Even so, with all these effects – unstructured motion, shadows, and highly incoherent reflections in the animated triangles – the museum can still be

	with GL-like shading					with full ray tracing				
num CPUs	1	2	4	8	16	1	2	4	8	16
robots	2.8	5.55	10.8	21	26⋆	0.54	1.07	2.15	4.3	8.6
kitchen	3.2	6.4	12.8	25.6	26⋆	0.47	0.94	1.88	3.77	7.55
terrain	1.3	2.5	4.8	8	15	0.9	1.77	3.39	6.5	12
museum:										
w/ 1k	2.7	5.4	10.2	19.5	26⋆	0.6	1.2	2.4	4.8	9.3
w/ 4k	2.5	4.5	7.5	4.5	2.5	0.55	1.1	2.2	4.2	2.5
w/ 16k	1.6	2.4	1.7	1	0.5	0.45	0.9	1.65	0.98	0.53

Table 9.4: Scalability of our method in the different test scenes (BART robots, BART kitchen, Outdoor Terrain, and BART museum with 1k, 4k, and 16k dynamic triangles). "⋆" means that the servers network connection is completely saturated in our network configuration, and thus no higher performance can be achieved. The numbers in the left half of the table correspond to pure OpenGL like shading, the right half is for full ray tracing including shadows and reflections. As can clearly be seen, for scenes with hierarchical animation scalability is almost linear up to the maximum network bandwidth for the final pixel data. With an increasing amount of unstructured motion (museum 1k–16k), the required network bandwidth for sending the changed triangles to the clients soon becomes a bottleneck. In that case, adding more CPUs even reduces performance, as data has to be sent to even more clients. An overview of these scenes can be found in Figure 9.6

rendered interactively: Using 8 clients, we achieve 4.8 fps for 1024 triangles, and still 4.2 fps for 4096 triangles in video resolution (see Table 9.4). Again, the frame rate is dominated by the cost for shadows and reflections. Using an OpenGL-like shader without these effects allows to render the scene at 19 frames per second on 8 clients.

9.7.4 Outdoor Terrain

The terrain scene has been specially designed to stress scalability with a large number of instances and triangles. It contains 661 instances of 2 different trees, which correspond to more than 10 million triangles after instantiation. A point light source creates highly detailed shadows from the leaves (see Figure 9.6). All trees can be moved around interactively, both in groups or individually. The large number of instances results in construction times for the top-level BSP of up to 4 ms per frame. This cost — together with the transmission cost for updating all 661 instance matrices on all clients — limits the scalability for a large number of instances and clients (see Table 9.4).

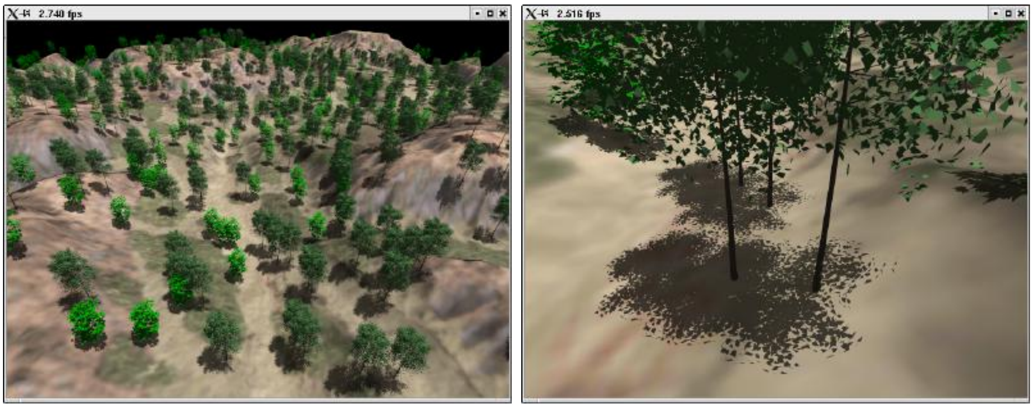


Figure 9.10: Terrain scene with up to 1000 instances of 2 kinds of complex trees (661 instances in the shown configuration, as some trees have been interactively moved off the terrain). Without instantiation, the scene would consist of roughly 10 million triangles. a.) Overview of the whole scene, b.) Detailed shadows from the sun, cast by the leaves onto both floor and other leaves.

9.8 Discussion

The above scenes have been chosen to stress different aspects of our dynamic ray tracing engine. Together with the terrain experiment, our test scenes contain a strong variation of parameters – from 5 to 661 instances, from a few thousand to several million triangles, from simple shading to lots of shadows and reflections, and from hierarchical animation to unstructured motion of thousands of triangles (for an overview, see Figure 9.6). Taken together, these experiments allow for a detailed analysis of our method.

9.8.1 Transformation Cost

For mainly hierarchical animation, the core idea of our method was to trade the cost for rebuilding the acceleration structure for the cost to transform the rays to the local coordinate system of each object. This implies that every ray intersecting an object has to be transformed via matrix-vector multiplications for both ray origin and direction (for every object encountered), potentially resulting in several matrix operations per ray. With a ray tracing performance of up to several million rays per second (see Chapter 7), this can amount to many million matrix-vector multiplications per frame! For example, the *terrain* and *robots* scenes at 640 × 480 pixels require 1.6 and 1 million matrix operations, respectively (see Figure 9.5). Furthermore, more transformations are often required during shading, e.g. by transforming the shading normal or for calculating procedural noise in the local coordinate system.

	office	terrain	robots
objects	9	661	161
matrix ops	480k	1.6M	1M

Table 9.5: Number of matrix-vector multiplies for our benchmark scenes (resolution 640x480). A matrix operation can be performed in only 23 cycles even in plain C code, which is negligible compared to traversal cost.

However, the cost for these transformations in practice is quite tolerable: Even for a straight-forward C-code implementation, a matrix-vector operation costs only 23 cycles on an AMD AthlonMP CPU, which is rather small compared to the cost for tracing a ray (which is in the order of several hundred to a few thousand cycles). Note that the matrix-vector multiplies are ideally suited for fast SSE implementation. This reduces this cost even further, and makes the transformation overhead almost negligible.

9.8.2 Unstructured Motion

As could be expected, the museum scene has shown that unstructured motion remains costly for ray tracing. A moderate number of a few thousand independently moving triangles can easily be supported, but larger numbers still lead to high reconstruction times for the respective objects (see Table 9.3). As such, our method is still not suitable for scenes with strong unstructured motion.

To support such scenes, algorithms for faster reconstruction of dynamic objects have to be developed. Note that our method could also be combined with Reinhard's approach [Reinhard00] by using his method only for the unstructured objects. Even then, lots of unstructured motion would still create a performance problem due to the need to send all triangle updates to the clients. This is not a limitation of our specific method, but would be similar for any algorithm in a distributed environment.

9.8.3 Bounding Volume Overlap

One of the stress cases identified in [Lext00] was *Bounding Volume Overlap*. In fact, this does create some overhead, as in the overlap area of two objects, these two objects have to be intersected *sequentially* by each ray. As a result, a successful intersection found during traversal of the first object may later-on be invalidated by a closer one in the other object. In fact, this partially disables "early ray termination" and thus negatively affects the occlusion culling feature of ray tracing[8].

Though it is easy to construct scenarios where bounding volume overlap would lead to excessive overhead, it is rarely significant in practice. In fact, bounding volume overlap *does* happen in *all* our test cases, but has never shown to pose a major performance problem. In fact, overlapping objects are exactly what happens all the time in bounding volume hierarchies (BVHs) [Rubin80, Kay86, Haines91a], which have also proven to work rather well in practice.

9.8.4 Teapot-in-a-Stadium Problem

The teapot-in-a-stadium problem is handled very well by out method: BSPs automatically adapt to varying object density in a scene [Havran01]. This is true for both object and top-level BSPs. In fact, our method can even increase performance for such cases: If the 'teapot' is contained in a separate object, the shape of the 'stadium' BSP is usually much better, as there is no need any more for several BSP subdivisions to tightly enclose the teapot.

9.8.5 Over-Estimation of Object Bounds

Building the top-level BSP requires an estimate of the bounding box of each instance in world coordinates. As transforming each individual vertex would be too costly, we conservatively estimate these bounds based on the transformed bounding box of the original object.

This somewhat over-estimates the correct bounds and thus results in some overhead: During top-level BSP traversal, a ray may be intersected with an object that it would not have intersected otherwise. However, this overhead is restricted to only transforming and clipping the ray: After transformation to the local coordinate system, such a ray is first clipped against the correct bounding box, and can thus be immediately discarded without further traversal.

[8] Note that for shadow rays, finding intersections in the wrong order is not a problem, as *any* valid intersection is sufficient to determine occlusion of a ray. Even so, Bounding Volume Overlap may still lead to the situation that the object containing an occluder will be traversed rather late during traversal of the toplevel structure.

9.8.6 Scalability with the Number of Instances

Apart from unstructured motion, the main cost of our method results from the need to recompute the top-level BSP tree. As such, a large number of instances is expensive, as can be seen in the terrain scene. Thus, the number of instances should be minimized in order to achieve optimal performance. Usually, it is better to use a small number of large objects instead of many small ones. For example, all static triangles in a scene should best be stored in a single object, instead of using multiple objects. This is completely different to approaches commonly used in OpenGL, in which many, small display lists are used. Thus, some amount of manual adjustment and optimization may be required when porting applications from OpenGL to OpenRT.

Even so, even the thousand complex instances can be rendered interactively, and top-level reconstruction has not yet proven a real limitation in any practical application. For moderate numbers of objects, top-level reconstruction is virtually negligible.

On the other hand, supporting instantiation (i.e. using exactly the same object multiple times in the same frame) is a valuable feature of our method, as this allows for rendering complex environments very efficiently: With instantiation, memory is required for storing only one copy of each object to be instantiated, plus the top-level BSP, allowing to render even many million triangles with a small memory footprint (see Figures 9.5 and 9.10). For triangle rasterization, all triangles would still need to be handled individually by the graphics hardware even when using display lists.

9.8.7 Scalability in a Distributed Environment

As can be seen by the experiments in Section 9.7, we achieve rather good scalability even for many clients except for scenes that require to update a lot of information on all clients, i.e. for a high degree of unstructured motion (where every moving triangle has to be transmitted), and for a large number of instances.

In the terrain scene, using 16 clients would require to send 676 Kb[9] per frame simply for updating the 661 transformation matrices on the clients. Though this data can be sent in a compressed form, load balancing and client/server communication further adds to the network bandwidth. Without broadcast/multicast functionality on the network, the server bandwidth increases linearly with the number of clients. For many clients and lots of updated data, this creates a bandwidth bottleneck on the server, and severely limits the scalability (see Table 9.4). In fact, performance could even *drop*

[9] 661 instances×16 clients×(4×4) floats

when adding many more CPUs, as each new client increases the network load. In principle, the same is true for unstructured motion, where sending several thousand triangles to each client also creates a bandwidth bottleneck. On the other hand, both problems are not specific to our method, but will apply similarly to any method running on such a distributed hardware architecture.

9.8.8 Total Overhead

In order to estimate the total overhead of our method, we have compared several scenes in both a static and dynamic configuration (also see Table 7.6 in Section 7). As there are no static equivalents for the BART benchmarks, we have taken several of our static test scenes, and have modified them in a way that they can be rendered in both a static configuration with all triangles in a single, static BSP tree, as well as in a dynamic configuration, where triangles are grouped into different objects that can then be moved dynamically.

For simple scenes, the total overhead is relatively high[10], compared to the small cost of rendering a static version of these scenes, and even reaches up to a factor of two in total rendering time (also see Chapter 7). For more realistic scene sizes, however, the *relative* overhead is significantly less, and in practice is often in the range of 20 to 30 percent, sometimes even less. This is very fortunate, as the overhead is worst for simple scenes in which *absolute* performance is highest anyway, and is relatively small for more costly scenes in which a high overhead would hurt most. Furthermore, most practical applications use rather complex scenes, and thus have a small overhead. As such, we believe this overhead to be a reasonable price for the added flexibility gained through supporting dynamic scenes.

9.9 Conclusions

The presented method is a simple and practical approach to handling dynamic scenes in an interactive distributed ray tracing engine. It can handle a large variety of dynamic scenes, including all the BART benchmark scenes (see Figure 9.6). It imposes no limitations on the kind of rays to be shot, and as such allows for all the usual ray-tracing features like shadows, reflections, and even global illumination (see Section 11 and Part III).

[10] Note that *total* overhead includes *all* the previously mentioned sources of overhead, e.g. including toplevel reconstruction, bounding volume overlap, traversal cost, multiple traversal setup cost, etc.

For unstructured motion, the proposed method still incurs a high reconstruction cost per frame, that makes it infeasible for a large number of incoherently moving triangles. For a moderate amount of unstructured motion however (in the order of a few thousand incoherently moving triangles), it is well applicable and results in frame rates of several frames per second at video resolution. For mostly hierarchical animation our method is highly efficient and achieves interactive performance even for highly complex models with hundreds of instances, and with millions of triangles per object [Wald02a]. This is especially furtunate since many of todays scene graph libraries (especially in VR/AR and other industrial applications) mostly use only this kind of animation.

The proposed technique forms a core part of the RTRT/OpenRT core, and has been used exclusively in all of the published applications of this system (also see the applications in Section 11 and Part III that all use this technique). Though it is certainly possible to construct cases in which the proposed method breaks down, so far is has been successfully able to handle all the dynamic scenes that have been encountered in practical applications of RTRT/OpenRT.

In conclusion, the proposed method of handling dynamic scenes is still limited but nonetheless already good enough for a large class of applications. Furthermore, the support for dynamic scenes in ray tracing is likely to improve significantly in the near future as more researchers start looking at this problem. Still, there remains a vast potential for future research in this area.

9.10 Future Work

At the moment, the main remaining scalability bottleneck lies in communicating all scene updates to all clients, making the total bandwidth linear in the number of clients. Thus, future work will investigate to use network broadcast/multicast to communicate the scene updates. As almost all of the updated data is the same for every client, this should effectively remove the server bottleneck. Furthermore, the above-mentioned concept of “geometry shaders” seems to be an interesting option for reducing the scene update bandwidth.

On the clients, the main bottleneck is the cost for reconstructing objects under unstructured motion. This could be improved by designing specialized algorithms for cases where motion is spatially limited in some form, such as for skinning, predefined animations, or for key-frame interpolation.

For the top-level BSP, it could be highly beneficial to investigate the use of cost functions. This especially includes finding ways of building such high-quality BSPs in a fast and efficient manner.

Apart from these technical issues, it is also important to investigate how existing applications can be mapped to our method, e.g. by evaluating how a scene graph library such as OpenInventor [Wernecke94], OpenSG [OpenSG01], OpenSceneGraph [OSG] or VRML [Carey97] can be efficiently implemented on top of our system. Preliminary results seem promising [Dietrich04].

Finally, it is an obvious next step to integrate this techniques into a hardware ray tracing architecture such as the SaarCOR architecture [Schmittler02] (Section 5.3). As such an architecture avoids most of the distribution problems we have in our PC cluster, such a mapping should be highly successful. First results are encouraging.

Chapter 10

The OpenRT Application Programming Interface

In the preceding chapters, all the basic constituents of a complete realtime ray tracing engine have been described: A highly efficient ray tracing kernel for modern CPUs (Section 7), its efficient parallelization (Section 8), and a simple yet efficient framework for handling dynamic scenes (Section 9).

Once these "building blocks" have successfully been merged, essentially all the *technical* requirements for realtime ray tracing are fulfilled. However, a key issue for reaching the scenario of realtime ray tracing on everybody's desktop is widespread application support, which requires a standardized and commonly accepted API. Roughly speaking, having a powerful new technology is one thing, having a good means of making the power of this technology available to the "average end user" is a totally different story. For hardware rasterization, this role of a powerful and widely accepted API has been taken by OpenGL [Neider93][1], which today is used by almost any graphics application, and which is well-known to virtually any graphics developer. Ideally, one could simply adopt OpenGL for ray tracing, in which case any existing OpenGL application could transparently render its images using ray tracing. Thus, one could (in theory) write an OpenGL "wrapper library" in the spirit of WireGL/Chromium [Humphreys02], that would perform the state tracking, would extract a ray-tracing suitable scene description from that, and would then call the ray tracer. The extended capabilities of ray tracing – namely shaders and global effects – could then be made available to the graphics programmer via the use of the well-known OpenGL extension mechanism (i.e. by offering a "GL_ARB_RAYTRACE" extension).

Unfortunately, this is but a mere theoretical option. as OpenGL and

[1] And, more recently, also by DirectX and Direct3D [DirectX].

similar graphics APIs are too closely related to the rasterization pipeline. These APIs clearly reflect the stream of graphics commands that is fed to the rasterization pipeline, and as such also closely reflect the capabilities *and* limitations of the rasterization approach. In contrast to OpenGL, RenderMan [Pixar89, Apodaca90, Hanrahan90, Upstill90, Apodaka00] offers a more suitable high-level API that also supports ray tracing. However, these APIs offer almost no support for interactive applications, and are thus not well suited for driving interactive applications.

Another option would be the use of an existing high-level scene graph library such as Performer, OpenInventor, OpenSG, OpenSceneGraph, or others [Rohlf94, Wernecke94, OpenSG01, OSG] for driving the ray tracer. This would already enable many new applications and would already reach a large number of potential users. However, the level of abstraction of such high-level scene graphs is too high for a generic ray tracing API, often is too application specific, and is unnecessarily restrictive. Being a low-level API, OpenGL allowed for all the “non-intended” uses (e.g. multipass rendering) while still allowing for layering higher-level APIs on top of it. In order not to unneccessarily restrict the potential uses of the API, it seems appropriate to follow this approach and design the API to be as low-level and flexible as possible. This allows for performing all the tasks that it is mainly thought for today, while still being flexible enough to adapt to potentially changing demands in the future.

As discussed above, none of the commonly available graphics APIs could be easily adopted for our ray tracing engine without unnecessary restrictions of its functionality. As such, we have decided to design a new API explicitly for realtime ray tracing. Ideally, such an API for realtime ray tracing should *not* only be an API specific to a certain implementation (i.e. the RTRT kernel), but should be both general and powerful enough to support the upcoming trend towards more widespread use of realtime ray tracing in general. Thus, we have designed our API with the following guidelines in mind:

- The API should be as low-level as possible in order to be able to layer higher-level scene graph APIs on top of it.

- It should be syntactically and semantically as similar to an existing, widely used graphics API as possible, in order to facilitate porting of existing applications and for leveraging programmers’ experiences. Being the most commonly adopted graphics API, we have chosen OpenGL as a “parent API” to our new API, and have thus called it “OpenRT”.

- It should be as powerful and flexible as RenderMan for writing shaders in order not to restrict the kinds of shading functionality that can be realized through this API.

The OpenRT API [Wald02a, Wald03e, Dietrich03] has been designed explicitly with these key observations in mind. While OpenRT so far has been implemented only on top of the previously mentioned RTRT kernel, it is not specific to this system. For example, the entire cluster infrastructure of the RTRT system has been completely abstracted from, and is not reflected in the API. Already today, two different implementations of this API are available, a distributed cluster version and a "stand-alone" shared-memory version (although both actually build on the same RTRT core). In the near future, it is planned to also use this OpenRT API for driving the SaarCOR architecture (Section 5.3).

10.1 General Design of OpenRT

As briefly mentioned above, one problem in designing OpenRT was choosing the right "parent" API to inherit ideas from: While it is generally a good idea to stay close to popular APIs – allowing to draw from a wide range of experienced people – there is the open question what API exactly to inherit from. On one side, OpenGL is the favorite API for writing interactive applications – it is very powerful, many people are experienced in OpenGL, and there is a huge amount of documentation and practical applications using OpenGL. On the other hand, OpenGL does not really fit a ray tracing engine: For example, it is mostly an immediate-mode API, and does not have any support for specifying shaders or for handling secondary effects (reflections, refraction) in a sensible manner.

In contrast to OpenGL, there are many APIs (like RenderMan[2], POV-Ray, etc.) that allow for taking advantage of all the benefits of ray tracing, but which are usually not applicable to interactive settings.

On the other hand, writing shaders and writing applications are usually two fundamentally different (though inter-playing) parts that can be realized with different APIs. As such, it is possible to take the best of both worlds, by using a RenderMan like API for writing shaders, and an OpenGL like API for writing the application. With this in mind, OpenRT has not been designed as *one* single graphics API, but actually consists of two mostly independent parts: One part is concerned with application programming,

[2] RenderMan was originally not designed to be a "ray tracing" API, but mainly to drive the REYES [Cook87] architecture. However, its flexibility and powerful shading language allow for also using it for ray tracing and global illumination, see e.g. [Gritz96, Slusallek95].

i.e. specifying geometric primitives, handling transformations and user input, handling textures, loading and communicating with shaders (but not writing them!), etc. This part of the API has been designed to be as close to OpenGL as possible. The second part of OpenRT describes how shaders are written – essentially describing a *shading language* – which has inherited much functionality from the RenderMan language, though it is not yet as flexible as “full” RenderMan.

10.1.1 Shader API – Application API Communication

All that is required to use this concept of having two separate parts of the same API is a minimal interface between these two subsystems. In OpenRT, this interface is realized via shader parameters (see below): Shaders are written independently from the application, and are stored in shared library files. Each shader “exports” a description of its shader parameters which control its shading calculations (e.g. the surface material properties to be implemented by this shader) but otherwise performs all its computations independently from the application. The application API then offers calls for loading these shaders, for binding them to geometry, for acquiring “handles” to their parameters, and for writing data to their parameters. For a closer description of this process, see below.

Having a clear abstraction layer between application interface and shading language it is also possible to exchange any of these two parts without affecting the other. For example, it would be possible to use different shading languages like e.g. Cg [Mark03, Fernando03], OpenGL 2.0 Shading language [Kessenich02], or RenderMan [Upstill90, Apodaca90, Apodaka00] for writing the shaders, while still using the same application interface.

Instead of adopting another API as a parent API, it would also have been possible to create a completely new, independent API from scratch. Such approaches however tend to reinvent the wheel, and often have problems getting widely accepted (and used) by the users.

10.2 Application Programming Interface

As just described the application programming part of OpenRT has been designed to be as close to OpenGL as possible. As a rule of thumb, OpenRT offers the same calls as OpenGL wherever possible (albeit using “rt” as a prefix instead of “gl”), and only uses different calls where a concept of ray tracing has no meaningful match in OpenGL (or vice versa). In particular any calls for specifying geometry (i.e. vertices or primitives), transformations, and textures have identical syntax and semantics as OpenGL. This simplifies porting of applications where large parts of the OpenGL code can be reused without changes.

10.2.1 Semantical Differences

Note however that OpenRT is *not* compatible with OpenGL. In fact, the general rule often has been to use the same syntax where possible but not supporting all semantical details that do not easily fit a ray tracer. In practice that means that there are several concepts in which OpenRT can be used just as a "typical" user would use OpenGL, even though the actual semantics slightly differ. For example, the viewpoint in OpenGL is usually specified via calls to `gluLookAt` and `gluPerspective`. While OpenRT offers exactly the same functions with the same parameters (consequently called `rtLookAt` and `rtPerspective`) that also specify the camera position, OpenRT does *not* exactly follow the OpenGL semantics of having these functions change a "perspective transform" which in OpenGL could also be used for other applications, e.g. projective textures. Supporting these semantical details in OpenRT does not make much sense, as a ray tracer uses the much more general and flexible concept of a camera shader instead of a perspective transformation.

Though there are actually several of such low-level semantical differences, most are actually not very important for practical applications, as they usually appear only for concepts in which the ray tracer offers a more general concept (such as a freely programmable camera shader) anyway. In fact, many users of OpenRT take quite a while to discover the first of these differences at all. While these semantical differences obviously make porting more complicated, the two main goals of making OpenRT similar to OpenGL are not successfully realized with this approach: First, to the average user, OpenRT appears quite similar to OpenGL, and thus is easy to learn, understand, and accept as a new API. Second, none of the flexibility, features and functionality of ray tracing have to be sacrificed in order to be comply to OpenGL features that simply don't match.

10.2.2 Geometry, Objects and Instances

The main issue with using OpenGL for ray tracing is the fact that no information is available about the changes between successive frames. In OpenGL, even unchanged display lists can be rendered differently in successive frames due to global state changes in-between the frames[3]. This however does not map to a ray tracing engine, which needs information on which parts of a scene did or did not change since the last frame in order to achieve interactive performance (see Section 9). Therefore, instead of display lists OpenRT offers *objects* (see Figure 10.1). Objects encapsulate geometry together with references to shaders and their attributes. In contrast to display lists, objects may not have any side effects and as such changing the definition of one object can never affect the shape or appearance of any other object. On the other hand, global side effects are still possible (and usually beneficial) for the appearance of an object: As the primitives only store *references* to shaders, changing a shader at any later time will immediately change the appearance of all the primitives that this shader is bound to[4].

Objects are defined using an `rtNewObject(id)/rtEndObject()` pair. Each object is assigned a unique ID that is used to instantiate it later by a call to `rtInstantiateObject(ID)`. Note how this is (intentionally) very similar to OpenGL's way of handling display lists (i.e. `glNewList(id),glEndList()` and `glCallList(id)`).

Essentially, an instance consists of a reference to an object, together with a transformation matrix that this object is subject to (see Figure 10.1). Therefore, re-instantiating an object with a different transformation will change the position of the object in the scene (also see the part on handling dynamic scenes in RTRT, Section 9).

In order to support unstructured motion, each object can be redefined at any time by calling `rtNewObject` with the same object ID. Note that here too, global side effects can take place once an object is changed: Redefining an object automatically changes all the instances that have instantiated the redefined object. Note that this API functionality perfectly matches the requirements of the previously proposed method to handle dynamic scenes as outlined in Section 9.

Here again, the detailed semantics of OpenGL display lists and OpenRT objects/instances are slightly different. For example, certain "special features" (such as the above-mentioned global state changes) are not supported by OpenRT objects. However, the way that the "average user" uses a display

[3] Actually, this "feature" of changing the effects of a display list by global state changes often even happens in the same frame.

[4] In OpenRT, the *shape* of the object (i.e. its triangles and vertices) is captured in the kd-tree, and will not be affected by any global state changes lateron. The *appearance* of the object if described by its references to the respective shaders (and, of course, by the global light sources shaders), and thus can change lateron by changing these respective shaders (see Figure 10.1. Even though this allows for side effects, it is conceptually slightly different from side effects through global state changes in OpenGL.

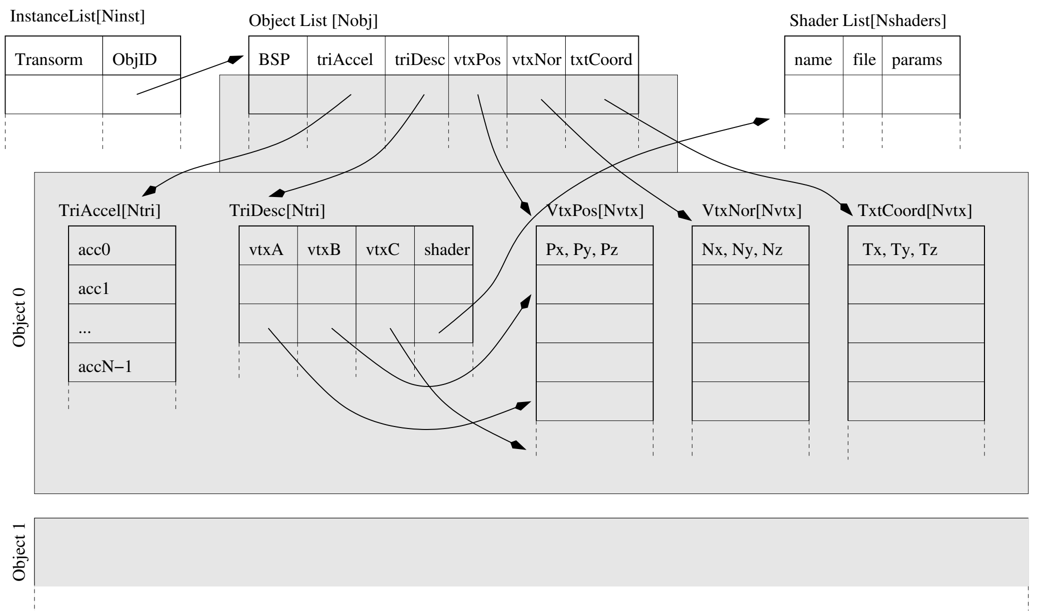


Figure 10.1: In the RTRT/OpenRT system, all geometry is encapsulated in objects*. Each object (the grey block) contains the vertices, triangle description records, as well as their local acceleration structure. Each triangle contains references to its three vertices, as well as to its globally defined shader. In fact, each of these objects corresponds exactly to the RTRT Kernel data structures as described previously in Section 7. In order to take effect, objects are instantiated, where each instance consists of an object ID and a transformation that this object is subject to (which corresponds to our method for handling dynamic scenes, as described in Section 9 and Figure 9.2). The entire scene then consists of the list of objects, the list of shaders, and the list of instances. Objects, shaders, and instances reference themselves by ID only, thereby allowing for dynamic and fully automatic side effects when changing any of these records.*

list (i.e. for encapsulating a certain part of a scene graph for faster rendering) corresponds exactly to what an OpenRT object is being used for. As such, most users will hardly see the difference at all.

10.2.3 Shading, Shaders and Lighting

In order not to be limited by the fixed reflectance model of standard OpenGL, OpenRT does not offer or emulate the OpenGL shading model at all, but rather supports *programmable shaders* similar to RenderMan [Pixar89]. Shaders provide a flexible "plug-in" mechanism that allows for modifying almost any functionality in a ray tracer, e.g. the appearance of objects, the behavior of light sources, the way that primary rays are generated, how radiance values are mapped to pixel values, or what the environment looks like. In its current version, OpenRT supports all these kinds of programmability by offering support for "surface", "light", "camera", "pixel", and "environment" shaders, respectively.

In terms of the API, shaders are named objects that receive parameters and can then be referenced lateron by name or ID, e.g. in order to attach ("bind") a surface shader to geometry. The syntax and functionality are essentially the same as the functionality to specify texture objects in OpenGL: A set of shader IDs is allocated by `rtGenShaders()`, and a shader with a certain ID is then loaded by `rtNewShader(ID)`. lateron, a previously defined shader can be activated at any time by `rtBindShader(ID)`, e.g. in order to assign to some geometric primitives. Binding shaders to geometry works similarly to how materials properties are "assigned" in OpenGL: The application just binds a certain shader, at which stage all primitives issues after this call get this respective shader assigned to them.

Once the primitive is issues, the ID of the shader bound to the respective triangle is stored with the respective triangle. As changing individual triangles is only possible by redefining the respective object containing that triangle, this shader-primitive binding can not be changed any more without redefining the object and re-issueing the primitives with a differently bound shader. Note however that the triangles actually store only *references* to their respective shader (in fact, the *ID* of the shader). As such, changing the shader associated to these triangles itself (i.e. loading a new shader with the same ID as the original one) thus allows for changing the appearance of the respective triangle or object without having to touch any triangle or object at all.

10.2.3.1 Parameter Binding

For communicating with the applications, shaders export "parameters", each parameter having a symbolic name (e.g. "diffuse"). The application can then register a handle to a specific shader parameter (`rtParameterHandle()`), and can write to that parameter with a generic `rtParameter()` call. Note that the syntax and semantics for defining and accessing shader parameters is almost exactly the same as proposed in the Stanford shader API [Proudfoot01, Mark01]. A shader can specify its parameters to reside in different "scopes", i.e. a shader can be specified to be stored per vertex, per triangle, per object, or per scene. For example, a Phong shader would most likely want to have its material parameters stored per shader, whereas a radiosity viewer might want to store certain radiosity values in the vertices[5]. These different ways to specify parameters allow for optimizing shaders and minimize storage requirements.

Using a parameter binding by name allows for a very flexible way of having an application communicate with many different kinds of shaders. For example, if a VRML viewer [Dietrich04]) follows the convention to always assign the diffuse component of its VRML material to the "diffuse" parameter of a shader, all that different shaders have to do to get access to the applications material model is to implement and export the respective shaders. In this example, the same diffuse parameter value can be used for both a simple flat shader as well as for a shader implementing interactive global illumination. Neither application nor shader have to know anything else about each other except that they follow this convention[6]. Overhead due to binding by name is not an issue: Once the "handle" to the parameter has been acquired by the application, the assignment itself does not have to consider any symbolic names any more.

10.2.3.2 Lighting

The same argument given for materials is actually true for lighting: The OpenGL lighting model simply is too limited for a ray tracer to be useful. As such, all lighting calculations are implemented via programmable light source shaders (see below).

For convenience and compatibility, the OpenRT library comes equipped with default implementations for all the typical OpenGL (or VRML) light source types like point lights, spot lights, directional lights, or ambient lights. Even so, loading these shaders is different from specifying a light source in OpenGL (via `glLight()`), and requires special handling when porting applications.

10.2.4 A Simple Example

Obviously, this thesis can not give a complete description of the full OpenRT API with all its details. However, for readers being familiar with both

[5] Obviously, it could do this also by storing them in the triangles vertex colors

[6] If the application tries to assign a value to a parameter that a shader never actually exported, this "invalid" assignment will be detected and ignored. This can be very useful for many applications: For example, a typical VRML application [Dietrich04] might simply assign the typical VRML material properties to each shader (writing to parameters named "diffuseColor", "specularColor", etc.). If the shader writer wants to have access to the VRML materials "diffuseColor", it simply has to export a parameter with that name.

OpenGL and with the concepts of a ray tracer, the following simple example should give a good overview of how OpenRT is used in practical applications [7].

```
// EightCubes.c:
// Simple OpenRT example showing
// eight rotating color cubes
#include <rtut/rtut.h> // include GLUT-replacement
#include <openrt/rt.h> // include OpenRT header files

RTint createColorCubeObject()
{
  // Create an object for our
  // vertex-colored cube

  // Step1: Define the *class* of a vertex color shader
  int cid = rtGenShaderClasses(1);
  //allocate one slot for a shader class
  rtNewShaderClass(cid,''VertexColor'',''libVertexColor.so'');
  // load shader class ``VertexColor'' from a shared library

  // Step2: Create one instance of that shader class
  int sid = rtGenShaders(1);
  // allocate one slot for a shader instance
  rtNewShader(sid); // creates an instance of the
                    // currently bound shader class
  ...

  // Step3: Define the object
  RTint objId = rtGenObjects(1);
  rtNewObject(objId, RT_COMPILE);
    // Step3a: Bind the shader
    rtBindShader(sid);
    // Step3b: Specify transforms
    rtMatrixMode(RT_MODELVIEW);
    rtPushMatrix();
    rtLoadIdentity();
    // scale the cube to [-1,1]^3
```

[7] The example given below uses a slightly outdated version of the OpenRT API (pre-1.0). In the most up-to-date version (currently 1.0R2), the example would look slightly different.

```
    rtTranslatef(-1, -1, -1);
    rtScalef(2, 2, 2);
    // first cube side
    // Step3c: Issue geometry
    rtBegin(RT_POLYGON);
      rtColor3f(0, 0, 0);
      rtVertex3f(0, 0, 0);
      rtColor3f(0, 1, 0);
      rtVertex3f(0, 1, 0);
      rtColor3f(1, 1, 0);
      rtVertex3f(1, 1, 0);
      rtColor3f(1, 0, 0);
      rtVertex3f(1, 0, 0);
    rtEnd();
    // other cube sides
    ...
    rtPopMatrix();
  rtEndObject(); // finish building the object
  return objId;  // return object's ID to the caller
}

int main(int argc, char *argv[]) {
  // Init, open window, etc.
  // virtually exactly the same as any GLUT program
  rtutInit(&argc, argv);
  rtutInitWindowSize(640, 480);
  rtutCreateWindow("Simple OpenRT Example");

  // set Camera
  rtPerspective(65, 1, 1, 100000);
  rtLookAt(2,4,3, 0,0,0, 0,0,1);

  // generate object *once*
  objId = createColorCubeObject();
  for (int rot = 0; ; rot++) {
    // instantiate object eight times,
    // re-instantititate object for every frame
    // with different transformation
    rtDeleteAllInstances();
    for (int i=0; i<8; i++) {
      int dx = (i&1)?-1:1;
      int dy = (i&2)?-1:1;
      int dz = (i&4)?-1:1;

      // position individual objects
      rtLoadIdentity();
      rtTranslatef(dx,dy,dz);
      rtRotatef(4*rot*dx,dz,dy,dx);
      rtScalef(.5,.5,.5);
      rtInstantiateObject(objId);
    }
    // start rendering and display the image
    // frame buffer automatically handled by RTUT
    rtutSwapBuffers();
  }
  return 0;
}
```

After opening a window, the “main” function first generates a vertex-colored RGB cube with a shader that just displays the interpolated vertex color. The cube is generated by first loading the “VertexColor” shader class from its shared library file, creating a single instance of it, and defining an object containing the geometry for the sides of the triangle. After the object has been completed, the “for”-loop creates eight rotating instances of this cube by re-instantiating each of the eight instances with a different transformation in subsequent frames. In fact, this simple example already features most of the important features of OpenRT: Specifying objects and instantiating them, issuing geometry, loading shaders, animating the objects, specifying the camera, and opening and using a window[8].

Being similar to OpenGL, this example should be easy to understand – and extend – by any slightly experienced OpenGL programmer. Of course, this is but a very simple example, and real programs will be considerably more complex. For example, a real program also has to load textures, specify light shaders, assign shader parameters, aso. Still, using advanced ray tracing effects in OpenRT is significantly simpler than generating the same effect in an OpenGL program: For example, rendering a scene once with global illumination effects and once without only requires to load a different shader – e.g. changing the shader name in “rtNewShaderClass” from “VertexColor”

[8] Though the example uses RTUT (a GLUT) replacement, it is not required to use this interface. It is also possible to directly get access to the ray tracers frame buffer, and to display this e.g. via OpenGL

to “InstantGlobalIllumination” (see Part III) – without having to touch any other code in the program.

10.2.5 Semantical Differences to OpenGL

As already mentioned before, there are several issues on which OpenRT semantically differs from OpenGL.

10.2.5.1 Retained Mode and Late Binding

For example, OpenRT differs from the semantics of OpenGL when binding references. OpenGL stores parameters on its state stack and binds references immediately when geometry is specified. This is natural for immediate-mode rendering, but does not easily fit a ray tracer. OpenRT instead extends the notion of identifiable objects embedding state, similar to OpenGL texture objects. However, this binding is performed only during rendering once the frame is fully defined. This approach significantly simplifies the reuse of unchanged geometric objects across frames, thus getting rid of the need to redefine such unchanged objects every frame. On the other hand this means that any changes to an objects or shader defined in a previous frame might also affect the appearance of geometry defined earlier. For example, changing a shader parameter will automatically change the appearance of all triangles that this shader is bound to, even if those triangles have been specified in an earlier frame. Similarly, redefining a geometric object will automatically and instantly change the shape of all instances of that object, even if those have been defined in a previous frame. Though this sounds obvious, it can lead to somewhat unexpected results for people being used to OpenGL. For example, the code sequence

```
rtGenNewShaderClass(‘‘Diffuse’’,’’libDiffuse.so’’);
RTint diffuse = rtParameterHandle(‘‘diffuse’’);
rtParameter3f(diffuse, 1.f,0.f,0.f);
<triangle A>
rtParameter3f(diffuse, 0.f,1.f,0.f);
<triangle B>
rtSwapBuffers(); // render frame
```

will actually result in two triangle that are *both* green[9], which is not what an OpenGL-experienced programmer would expect.

[9] Both triangles share *the same* shader. Until the two triangles are actually rendered during `rtSwapBuffers`, that shaders diffuse parameter has been set to green. Whether or not that parameter has had a different value when specifying triangle A does not make a difference.

Thus, these semantics are natural for a ray tracer but require careful attention during porting of existing OpenGL applications. More research is still required to better resolve the contradicting requirements of rasterization and ray tracing in this area.

10.2.5.2 Unsupported GL Functionality

Finally, some OpenGL functions are meaningless in a 3D ray tracing context and consequently are not supported in OpenRT. For instance, point and line drawing operations are not (currently) supported, and effects like "stipple bits" and "fill modes", as well as 2D frame buffer operations make little sense for a ray tracing engine either. Similarly, fragment operations, fragment tests, and blending modes are no longer useful and can be better implemented using surface and pixel shaders if necessary. Traditionally ray tracing writes only a single "fragment" to each pixel in the frame buffer after a complete ray tree has been evaluated. Thus the usual ordering semantics of OpenGL and its blending operations that are based on the submission order of primitives are no longer meaningful, either.

However the lack of this functionality so far has not been a problem for any of the applications already written on top of OpenRT: While these unsupported operations are very important for triangle rasterization, their main use is for multi-pass rendering. With the powerful shader concept offered by OpenRT, multipass-rendering is not neccessary any more, so this functionality so far has not been missed yet.

10.2.5.3 Frame Buffer Handling

Instead of writing the pixels to a hardware frame-buffer OpenRT renders into an application-supplied memory region as a frame buffer. This, however, is only due to the current hardware setup which uses a software implementation. For more dedicated ray tracing hardware, this is likely to change. For example, an OpenRT application on top of the SaarCOR architecture (Section 5.3) would most likely have the option to use a hardware frame buffer with direct VGA output instead of always transferring the rendered pixel values back to the application for display.

The above described "late binding"[10] also results in up to one frame of additional latency compared to OpenGL. The rasterization hardware can already start rendering as soon as the first geometric primitive is received by the renderer, and renders each primitive directly once it is specified (except

[10] Sometimes also called "frame semantics" to stress its difference from "immediate mode semantics"

for some buffering in the driver). Once all primitives have been sent to the graphics card, the resulting image as such is already finished. In contrast to this, the ray tracer has to wait for the full scene to be completely finished before it can actually start tracing any rays.

10.3 OpenRTS Shader Programming Interface

As motivated in the introduction of this chapter, the shader API in OpenRT (called "OpenRTS") has intentionally been designed to be mostly independent of the core API for writing applications. In order to allow for all the typical ray tracing effects that users are already used to, this API is as similar to RenderMan as possible, and closely reflects the "shader concept" as described in Section 2.2.4.1.

10.3.1 Shader Structure Overview

The base class of all shaders in OpenRT is the "OpenRT Plug-in", i.e. an entity that can be loaded dynamically from a file, and which offers functionality for registering itself and exporting its parameters[11]. Once a parameter has been exported, the application can lateron bind a 'handle' to this parameter, and can assign values to it (see the above OpenRT example). Apart from registration and parameter export, all `RTPlugin`s are equipped with an `Init` and NewFrame method that can be overwritten by its subclasses.

All other shader types – i.e. surface, light, camera and pixel shaders, and the rendering object (see below) – are derived from this base class, and as such can all be parameterized by the application.

10.3.1.1 Surface Shaders

The most common shader types in OpenRT obviously are surface shaders. Surface shaders have a virtual "`Shade`" function that is expected to return the color of the ray it got passed. For the shading operations, the surface shader has access to an extensive API for accessing scene data (e.g. vertex positions, normals, or texture coordinates) and for querying data concerning the ray and hit point (such as the shading normal, the ray origin and direction, the transformation that the hit object is subject to, etc). To differentiate these

[11] For convenience, we only speak about C++ classes for specifying shaders. Though OpenRT in principle also allows for writing pure C-code shaders, C++ classes are actually more natural for implementing a shader concept and as such are usually preferred.

shader API functions from those of the core OpenRT API, all these functions (except class methods) start with the prefix “rts”.

10.3.1.2 Accessing Light Sources

In order to access light sources, a surface shader can query a list of light shaders over which it can iterate. The surface shader can then call back to each light shader (via `rtsIlluminate(...)`) to ask it for an “illumination sample”, or “light sample”. A “light sample” consists of all 3 values required for doing the lighting calculations in the surface shader: The direction towards the light, its distance (possibly infinite), and the intensity with which it influences the hit position.

Once a light shader has returned its light sample, this sample forms a complete shadow ray description with origin, direction, and maximum distance. This shadow ray description can then (but does not have to) be used by the surface shader to compute shadows by calling `rtsOccluded(...)` with this light sample, which in turn uses the ray tracing core to cast a shadow ray. If semi-transparent occluders are used, the surface shader can also use `rtsTransparentShadows()` instead of `rtsOccluded`, which will iterate over all the potential occluders along the shadow ray to compute the attenuated contribution of the light source.

10.3.1.3 Casting Secondary Rays

Except for casting shadow rays via `rtsOccluded()` (or via `rtsTransparentShadows()` for computing transparent shadows), further secondary rays can also be shot via `rtsTrace`. This `rtsTrace` shoots an arbitrarily specified ray, determines the hit point, calls the respective shader at that hit point, and returns the color computed by that shader. In case the ray did not hit any objects, `rtsTrace` automatically calls the *environment shader* for computing the color of that ray.

While `rtsTrace` already allows for all kinds of rays to be generated and shot, OpenRT offers several “convenience functions” for the most often used kinds of secondary rays, like e.g. `rtsReflectionRay()`, `rtsRefractionRay()`, `rtsTransparencyRay()`, etc.

10.3.1.4 Light Shaders

Similarly to the “Shade” function of the surface shaders, light shaders have a virtual `Illuminate` method that can be overridden to write new kinds of light shaders. As described above, OpenRT already comes equipped with the most common light source shaders like point, spot, and directional lights. For global illumination purposes, OpenRT also contains a few area light source shaders. As the surface shader expects illuminate to return a *single* light sample, these area light shaders take a list of pseudo-random numbers that they got passed from the surface shader to create a light sample. If a surface sample needs multiple samples from the same light source, it has to call `rtsIlluminate` several times with different random numbers.

10.3.1.5 Camera Shader

Camera shaders work in a similar way as surface and light shaders: Each camera shader has a single virtual function for initializing and returning a primary ray through the pixel, which will then be cast into the scene via `rtsTrace()`.

10.3.1.6 Environment Shader

Environment shaders are automatically called for all rays traced via `rtsTrace` that did not hit an object. In fact, an environment shader is a shader like any other (i.e. with a `Shade()` function), except that it does not make any sense to query any hit point information within the shading code.

10.3.1.7 The Rendering Object Concept

Whereas all the surface, light, camera, and environment shaders are typical shader types in any programmable shader concept, OpenRT additionally offers the concept of a "rendering object". A rendering object is responsible for actually computing pixel values, and as such enables the user to completely change the way that the ray tracer works. Typically, a rendering object will call a camera shader to generate a primary ray through each pixel, will call `rtsTrace`, and will let the respective surface shaders do the rest.

For special applications however, the rendering object can skip this flexible though costly shader concept, and can perform the rendering in a more "hard-coded" way, e.g. by directly using the fast RTRT packet tracing code with a hard-coded shading model. Similarly, many global illumination algorithms do not easily fit the above surface shader concept[12], but can be quite efficiently implemented as a rendering object. As such, rendering objects greatly extend the range of applications that can be realized with OpenRT.

[12] For example, the above shader concept expects a shader to compute the color of a ray, whereas many global illumination algorithms require evaluation of a BRDF with given incoming and outgoing directions (such as bidirectional path tracing), or sampling of a BRDF (e.g. for photon shooting or generation of light- and eye-paths).

However, rendering objects are an advanced concept of OpenRT, and should be used with extreme care.

10.3.2 A Simple Shader Example

Obviously, the above explanation is but a very brief sketch of the OpenRT shader concept. The complete description of the shader API is beyond the scope of this thesis. More information on OpenRT and OpenRT shading can also be found in the respective OpenRT manuals and tutorials (see e.g. [Wald]).

As for the previously described application part of the OpenRT API, how the OpenRT Shader API is actually used in practice can best be described with a simple example. As such, the following example implements some simple (though typical) OpenRT shaders, one light shader and one surface shader. The surface shader implements a simple diffuse shader, parameterized by a diffuse color and an ambient term). The light shader implements a typical point light source consisting of position and intensity, and with a hard-coded quadratic intensity falloff.

10.3.2.1 Simple Diffuse Shader

```
class SimpleDiffuse : public RTShader {
  RTVec3f diffuse;
  RTVec3f ambient;

  RTvoid Register() {
    // register parameters
    rtDeclareParameter("diffuse", PER_SHADER,
                       offsetof(diffuse),sizeof(diffuse));
    rtDeclareParameter("ambient", PER_SHADER,
                       offsetof(ambient),sizeof(ambient));
  }

  RTvoid Shade(RTState *state)
  {
    RTVec3f color = ambient;      // init with ambient color
    RTVec3f P;                    // surface hit position
    RTVec3f N;                    // normal
    rtsGetHitPosition(state,P);
    rtsFindShadingNormal(state,N);// interpolate normal, make
                            // sure it faces toward the viewer
    RTState shadow = *state;      // init shadow ray state
    RTenum *light; RTint lights;
    lights = rtsGlobalLights(&light);
    for (int i=0;i<lights;i++)
    { // iterate over all light sources
      rtsIlluminate(light[i],P,&shadow,NULL);
      if (rtsOccluded(&shadow))
         continue; // test for shadows

      Vec3f L; // light direction
      Vec3f I; // light intensity
      rtsGetRayDirection(&shadow,L);
      rtsGetRayColor(&shadow,I);
      RTfloat cosine = N * L;     // dot product
      I *= diffuse;               // component-wise mult.
      color += (cosine * I);
    }
    rtsReturnColor(state,color);
  }
};
rtsDeclareShader(SimpleDiffuse, SimpleDiffuse);
```

10.3.2.2 Simple PointLight Shader

```
class SimplePointLight : public RTLight {
  RTVec3f position;
  RTVec3f intensity;

  RTvoid Register() {
    rtDeclareParameter("position",
         offsetof(position),sizeof(position));
    rtDeclareParameter("intensity",
         offsetof(intensity),sizeof(intensity));
  }

  RTvoid Illuminate(RTState *state) {
    RTVec3f P;                    // surface hit point
    rtsGetRayOrigin(state,P);

    RTVec3f L = position - P;     // direction towards light
    RTfloat distance = Lenght(L);
    Normalize(L);

    RTVecf3 I = intensity * 1./(distance * distance);
    // quadratic distance attenuation

    rtsSetRayDirection(state,L);
    rtsSetRayMaxDistance(state,length - Epsilon);
    rtsReturnColor(state,I);
  }
};
rtsDeclareShader(SimplePointLight, SimplePointLight);
```

10.4 Taking it all together

Having now described all the different parts of the API, it is important to briefly summarize how these different parts actually play together. To do this, we will briefly go – step by step – through the process of rendering a frame:

1. First, the application specifies the scene itself, i.e. it loads and parameterizes shaders, specifies objects and instances, issues geometry, sets the frame buffer, etc. All the time, the OpenRT implementation makes sure that all these calls get executed on all rendering clients, be it the local CPU, remote cluster clients, or a hardware architecture.

2. Once the scene is specified, the application calls `rtSwapBuffers` to tell the ray tracer that any scene updates are finished and that it should render a frame.

3. Upon `rtSwapBuffers` the OpenRT library calls the user-programmable rendering object to actually perform the rendering computations. In a single-CPU or shared-memory version, the rendering object will simply render a complete frame. In the distributed cluster version, the ray tracer will automatically perform the load distribution, load balancing, and communication between the clients and the server. As such, it will automatically request each client's respective rendering object to render one or more tiles.

4. The rendering object iterates over all the pixels in its frame (respectively tile), and calls the user-programmable camera shader to generate a primary ray through that tile.

5. Once a valid primary ray has been generated, the rendering object tells OpenRT to trace this ray and compute its color. To do this, OpenRT uses the RTRT kernel to trace the ray and find a hit point.

6. If no valid hit could be found, OpenRT automatically calls the (user-programmable) environment shader to shade the ray. If a hit was found, OpenRT determines the respective surface shader and calls its `Shade` method.

7. The (user-programmable) surface shader uses the shader API to call back to the library while performing its shading computations, e.g. by asking OpenRT for the list of active lights, or for the shading normal of the hit point. This also includes asking OpenRT for a light sample from a given light shader. OpenRT will then look up that light shader, and call its respective`Illuminate` function.

8. The light shader generates this light sample (probably with some additional calls into the shader API), and returns this – via OpenRT – to the surface shader.

9. Having processed all light samples, the surface shader may tell OpenRT to shoot some additional secondary rays, for which stages 5–9 are recursively repeated[13].

10. Once the entire shading tree has been processed, the rendering object has the color of the hit point as determined by the surface shader. It may now do some final operations on this ray (in the spirit of a "pixel shader"), e.g. for performing tone mapping. Once this is done, it writes the pixel to the frame buffer. Again, in the distributed version all these pixels that have been computed on different machines get automatically communicated back to the server (where they can be again manipulated by a user-programmable routine).

11. Once all pixels have been computed, the OpenRT library returns the frame buffer to the application, and returns from the `rtSwapBuffers` call.

12. The application can now display the frame buffer, and can start over by starting to specify the next frame.

[13] Obviously, the secondary rays can be shot at any time, not only at the end of the shader routine.

Though this is in fact exactly the same ray tracing pipeline that any decent ray tracer uses as well, two things are important to note: first, the modularity and programmability of this framework, and second, the hardware abstraction model used in OpenRT.

10.4.1 Modularity and Programmability

First of all, taking a closer look at the above topics makes clear that OpenRT is a highly flexible API in which almost all parts are user programmable and can be arbitrarily replaced. Surface, light, environment and camera shaders, the rendering mode, and to a certain degree even the parallelization can be changed by the user.

The OpenRT library in fact provides only the basic infrastructure – such as abstracting from the distributed architecture, automatic handling of all parallelization and communication, scene management etc – and nicely glues the different user-programmable parts together. Last but not least, the OpenRT library also drives the ray tracing kernel and makes it available to all the respective subsystems.

Of course, for all these user-programmable parts (such as generating the tiles in the rendering object, generating primary rays, or assembling the pixels to the final image) there are optimized default routines. Most users will never make contact with any of these advanced issues, and will concentrate on writing surface and/or light source shaders.

10.4.2 Hardware Abstraction Model

The second important issue to mention is how this design carefully abstracts from the underlying hardware. For example, the shader-application communication works *entirely* over the shader parameter concept, and never assumes any direct communication between shaders and application. As such, the shaders can either be located on the same machine as the application, or could run on another, remote machine that does not even know about the application. It would just as well possible that the shaders themselves are not software C++ classes at all, but might reside directly on a ray tracing hardware architecture such as SaarCOR.

Similarly, the shader API (i.e. the API used by the shader programmer) is strictly kept apart from the main OpenRT application API. As such, the same application program could be used even if the shader API changes. For example, the SaarCOR architecture (see Section 5.3) obviously will not use the same C/C++ shader API that is currently used on the CPU[14].

10.5 Conclusions and Future Work

In summary, OpenRT is a simple yet highly flexible API for realtime ray tracing. It is simple to use and flexible enough to support all typical ray tracing effects though a RenderMan like shading API and a highly modular user-programmable plug-in concept.

While the application API is not actually semantically 100% compatible to OpenGL, the syntax and semantics for typical programs are still very similar. Thus, novice OpenRT users with (some) previous OpenGL experience so far found OpenRT easy to learn and use. In fact, many concepts (e.g. shaders) appeared easier and more natural to these users. Because of this, OpenRT so far has shown to be well accepted by current users. However, highly experienced OpenGL users (which tend to know – and use – all the subtle details of OpenGL) sometimes found it hard to understand that certain concepts are different (e.g. that a `rtLookAt` call does not have any side effects on the matrix stack that could then be exploited for projective textures). Though some open questions remain, OpenRT has already been used for several practical projects, and so far has been very successful for those projects. For example, it has been used exclusively for all examples and applications throughout this thesis (especially see Chapter 11).

For really widespread use, however, still some more work has to be invested: First, it would be desirable if more different implementations of the OpenRT API would be available, e.g. on the SaarCOR architecture, on a GPU-based implementation (e.g. [Purcell02]), or on an open source ray tracer.

Furthermore, it has to be evaluated whether – and how – the remaining differences between OpenGL and OpenRT could be bridged. Eventually, it would be a highly interesting option to somehow combine OpenGL and OpenRT, e.g. by making OpenRT to be an OpenGL extension. Due to fundamentally different semantics as discussed above, it yet unclear if this is possible at all, let alone in which way.

An even more important issue to work on is an efficient shading API that supports coherent packets of rays. As described in Section 7, the full performance of the RTRT core can only be unleashed if SIMD packet traversal

[14] Though it is still imaginable to use the same "shading language" for both the software and the hardware implementation, e.g. by using different shading language compilers (with the same syntax) for the different platforms.

with efficient SIMD shading can be used. In its current form, however, the OpenRT shader API actually supports only the shading and tracing of single rays. For those having the actual RTRT sources, it is still possible to use both packet traversal code and OpenRT API at the same (e.g. by performing the packet traversal code inside a rendering object), but a clean external API does not exist. As it is not yet even clear how such packet shading could be efficiently performed at all (see the discussion in Section 7), it seemed premature to already discuss its API issues.

Finally, probably the biggest challenge for the success of OpenRT is to create new, powerful interactive applications. This also implies making it available to a much wider range of users to actually build these applications. Though all our experiences with OpenRT so far have been highly encouraging, only once many different kinds of users will actually use if for solving their everyday practical rendering problems will it be possible to objectively evaluate the real potential – and the limitations – of this API.

As most applications in fact operate on a much higher level of abstraction – usually working on scene graphs rather than directly on the API level – making OpenRT available to a wider range of users also implies to investigate how scene graphs can be efficiently mapped to the new API. Preliminary work has already investigated how a VRML-based scene graph (the XRML engine [Bekaert01]) can be mapped to OpenRT [Wagner02, Dietrich04]. However, an even deeper investigation of this problem has yet to be performed.

Chapter 11

Applications and Case Studies

"To infinity - and beyoooonnnndddd...."

Buzz Lightyear, Toy Story, Pixar

.

In the previous chapters, all the basic building blocks for a complete rendering engine have been outlined: A fast ray tracing core that achieves high rendering performance and which supports freely programmable shaders, even higher performance using parallelization that is transparent to the user, support for hugely complex scene, the ability to handle dynamic scenes, and finally a powerful API to make all this functionality available to the user.

The availability such a rendering engine now enables a whole list of new applications that cannot as easily be realized using a different technology. This chapter will briefly summarize some of these applications that have already been realized with the RTRT/OpenRT engine. Though many of these examples are also described in more detail elsewhere in this thesis, this summary provides a good overview of the capabilities and potential of the RTRT/OpenRT realtime ray tracing engine.

On todays technology, most applications are still limited in frame rate and resolution. If not noted otherwise, all examples are running at 640×480 pixels. Furthermore they currently lack sufficient performance on a single desktop PC [1], but require the combined performance of multiple PCs. Given the current trend of improvements in both ray tracing performance and hardware technology, it is likely that these applications will be available on a single desktop PC in only a few years from now.

[1] Note that pure ray casting, e.g. for simply visualizing complex models with a simple shading model, can already reach sufficient performance on a single state-of-the-art dual-CPU desktop machine!

11.1 Classical Ray Tracing

Clearly, the most obvious application of interactive ray tracing is to interactively compute classical ray tracing – i.e. “whitted-style” [Whitted80] recursive ray tracing with programmable shaders, perfect specular reflections and shadows from point light sources. For this application, the big advantage of ray tracing is its ability to support plug-and-play shading: Shaders for certain specialized effects can be written independently of all other shaders, and can still automatically and seamlessly work together with the rest of the scene. For example, Figure 11.1 shows a typical office scene with several advanced shaders, like procedural wood and marble shaders, procedurally bump-mapped reflections on the mirror, reflections on the metal ball, and even shaders performing lightfield [Levoy96] and volume rendering [Parker99a].

Note how all of these effects work seemslessly and automatically together: For example, the volume data set (the skull) casts a transparent shadow onto the procedural wood shader on the table, the reflection of which can be seen in the metal ball. Similarly, the volume object can be seen in the bump-mapped mirrow on the wall, and the lightfield object looks different depending on which direction the rays are coming from[2]. All this happens automatically, without any shader knowing about any of the other shaders. Furthermore, everything is fully recomputed from scratch every frame. For example, moving the volume object will automatically update all reflections or shadows cast by this object. As such, even when interactively changing the scene, all primary and secondary effects are automatically recomputed in a correct way in each individual frame; this happens fully automatically and without any further intervention by user or application program.

This automaticity and ease of combining several individual effects is of huge importance for applications like games, in which the combination of different effects - especially indirect effects like shadows or reflectios - currently require manual tuning of both code and scene to generate a desired effect. With ray tracing, many of these tasks could be simulated both more correctly and more easily, while, if neccessary, still allowing for “manual” intervention through the use of specially designed textures.

Using graphics hardware, *each* of these effects can often trivially be realized at realtime frame rates, but combining several such effects usually is highly non-trivial. Even if this is possible at all on graphics hardware, performance typically degrades rapidly with each added effect, as each additional

[2] Note how from the camera position one sees the back of the dragon’s head, whereas the reflection in the mirror shows the dragon’s mouth. Though being natural, tihs effect would not take place if the dragon were a simple texture.

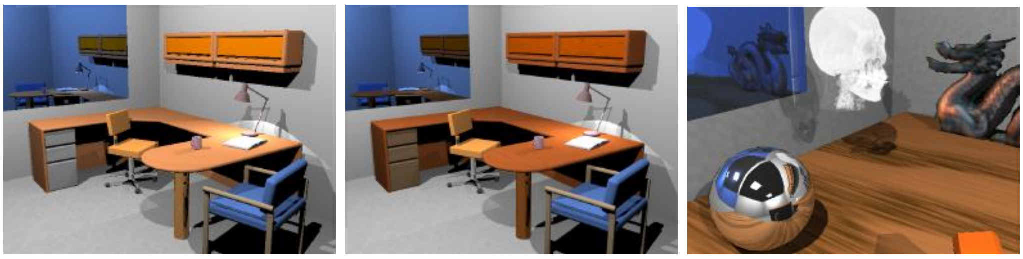


Figure 11.1: An office environment with different shader configurations (from left to right): (a) A typical Phong shader computing reflections and shadows from three lights, (b) enhanced with additional procedural shaders for wood, and (c) with additional volume and lightfield objects and procedural bump mapping on the mirror. All the different shaders were written independently but can be easily combined in a plug and play manner. Note that all optical effects work as expected: For example, the volume casts transparent shadows, the lightfield is visible through the bump-mapped reflections on the mirror, etc. All objects can be manipulated and moved interactively.

effect usually requires several additional rendering passes for computing this effect [3]. Furthermore, indirect effects like shadows and reflections have to rely on coarse approximations, and often require manual tuning to reach convincing results and performance. In ray tracing, combining the different effects is trivially and automatically handled by the ray tracer, and performance usually degrades only linearly with the number of rays traced.

11.2 Physically Correct Simulation of Reflection and Refraction

As seen in the previous application, it is quite easy to use ray tracing for generating high-quality images. However, ray tracing can not only be used for generating "good-looking" images – the ability to automatically and correctly combine different shaders can also be used to visualize objects in a physically correct way. In contrast to classical rendering – whose task usually is to quickly generate nicely looking images – many applications require to compute *quantifyable* and *verifyable* results.

For example, industrial design or virtual prototyping applications need to use rendered images to decide whether a virtual prototype is to be physically built or not, and thus have to trust the renderer to generate a "correct" image.

[3] Plus several rendering passes for merging that effect 'into' the previously computed image

Thus, these application domains usually use offline photorealistic rendering packages, which in turn usually build on ray tracing. Today, however, most of these applications are limited to offline visualization, which poses quite some limitation for the designers.

With the advent of realtime ray tracing, physically correct lighting simulation at interactive rates now becomes a possibility. In a proof-of-concept application [Benthin02], interactive ray tracing has been used for physically correct simulation of the reflection and refraction behaviour in a car headlight produced by a German headlight manufacturer (see Figure 11.2). Using the RTRT/OpenRT engine together with appropriately written shaders allowed to simulate up to 25 levels of reflection and refraction even in an 800,000 triangle model at interactive rates.

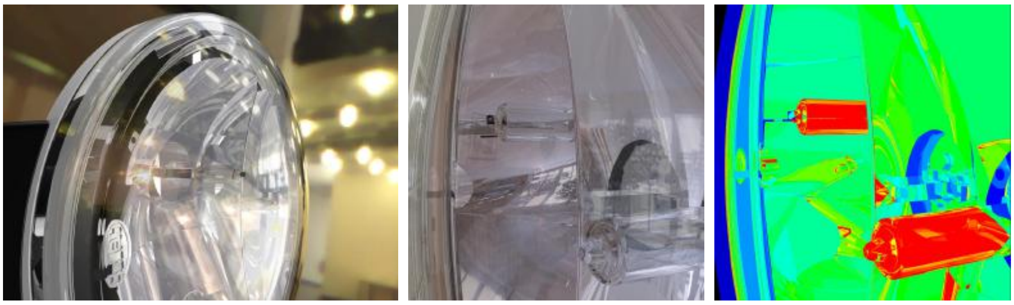


Figure 11.2: Interactive, physically correct visualization of the reflection and refraction behaviour inside the complex glass body of a car headlight. The car headlight is rendered with lighting from a sourrounding high-dynamic range environemnt map, and is displayed with interactive tone mapping. The simulation includes complex glass effects, and is performed with up to 25 recursive levels of reflection and refraction. a.) An example screenshot of the whole lamp. b.) Another view zooming in on the actual lamp. c.) False-color visualization of the number of reflection levels per pixel in image b. (black: 0, red: 25+).

11.3 Visualizing Massively Complex Models

Apart from high-quality rendering as shown in the previous examples, one of the most obvious applications of ray tracing is visualizing massively complex models. In practice, the computational complexity of ray tracing is logarithmic in scene size (i.e. in the number of triangles), making it an efficient tools for rendering scenes with millions of individual triangles. For example, this allows for rendering complex objects such as the 12.5 million triangle

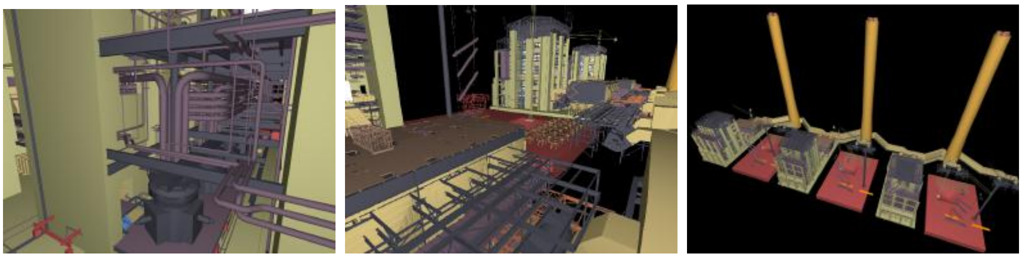


Figure 11.3: Three UNC power plants consisting of 12.5 million individual triangles each. A total of 37.5 million triangles is rendered at roughly 10 fps on 4 clients. Frame rates remain interactive no matter whether we zoom in on specific details or view the model as a whole (left to right).

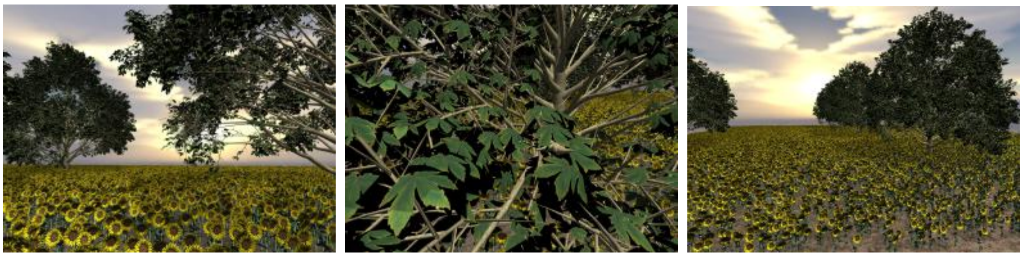


Figure 11.4: Instantiation: The "Sunflowers" scene consists of about 28,000 instances of 10 different kinds of sunflowers with roughly 36,000 triangles each together with several multi-million-triangle trees. The whole scene consists of roughly one billion triangles. The center image shows a closeup of the highly detailed shadows cast by the sun onto the leaves. All leaves contain textures with transparency which increase the number of rays needed for rendering a frame. The whole scene renders at roughly 7 fps on 24 dual PCs at video resolution. All objects including the sun can be manipulated interactively.

UNC "PowerPlant" scene, a model of a coal power plant with 48 stories of complex geometry. Other, non-ray-tracing approaches to rendering such models [Aliaga99, Baxter III02] have to rely on extensive simplifications and approximations, which may result in artifacts like missing detail, or popping. Furthermore, the extensive precomputations required for computing the simplified version of the model ususally restrict these approaches to static scenes. Adding indirect effects (like shadows) to these approaches is possible, but requires sophisticated algorithms even for shadows from a single point light source [Govindaraju03].

In contrast to this, with ray tracing the whole model can be interactively renderered *directly*, i.e. without the need for any geometric simplifications (see Figure 11.3). Grouping the geometry into different objects and using the techniques proposed in Section 9, it is even possible to interact with the power plant, e.g. by moving or transforming different parts of the scene. Finally, using ray tracing it would be quite simple to add secondary effects like shadows or reflections. In fact, even global illumination can be interactively computed in the power plant [Benthin03] (see Section III).

As the cost for ray tracing correlates much more with the shading complexity than with the model complexity, even such complex scenes are rather cheap when rendered with simple shading. For example, the power plant scene with simple OpenGL like ray casting can be rendered at 20+ frames per second on only 3 to 4 up-to-date PCs (see Section 8 and Figure 13.12).

While ray tracing already supports model sizes of several million individual triangles, instantiation allows for even further increasing the scene complexity. For example, the "Sunflowers" scene (see Figure 11.4) contains 28,000 instances of several kinds of sunflowers, plus several highly detailed trees, totalling one *billion* triangles. Note that this complexity is not *only* due to instantiation – even a single tree in this scene contains more than a million individual triangles.

Of course, the geometric complexity does not limit the kinds of effects that can be computed. For example, the sunflowers scene is rendered including shadows and transparency from semi-transparent leaves. Of course, all objects as well as the sun can be moved around interactively.

11.4 Interactive Ray Tracing for Virtual and Augmented Reality Applications

As shown in the previous examples, ray tracing offers many advantages in classical rendering. However, ray tracing may also have an impact on different kinds of applications that have not yet been widely thought of. For example, in a recently published prototype implementation, Pomi et al. [Pomi03] have shown that ray tracing can also be successfully employed for mixed and augmented reality applications. Such applications usually aim at mixing virtual and real objects by combining virtual 3D objects with images and (live) videos from the real world. This has been shown to generate compelling results in offline rendering [Debevec97, Debevec98], but could usually not be realized at interactive rates.

Using the RTRT/OpenRT realtime ray tracing engine, Pomi et al. have shown that it is indeed possible to compute such applications interactively, e.g. by rendering a virtual car into a real environment (see Figure 11.5), or

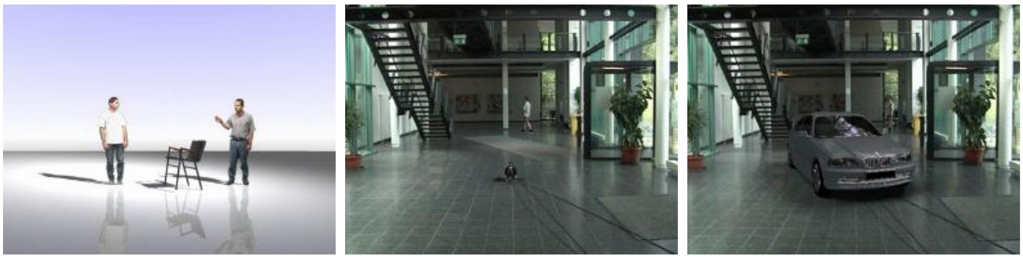


Figure 11.5: Augmented Reality Applications. Left: Video billboards with ray traced shadows and reflections. The chair is a synthetic object, but the two persons are real persons on video billboards, with compositing computed in the billboard shader. Note the automatically correct shadows and reflections, also from the video billboards. Center: An HDR image of a real scene, with a captured HDR environment map from a fisheye lens. Right: A synthetic car ray traced into this HDR environment.

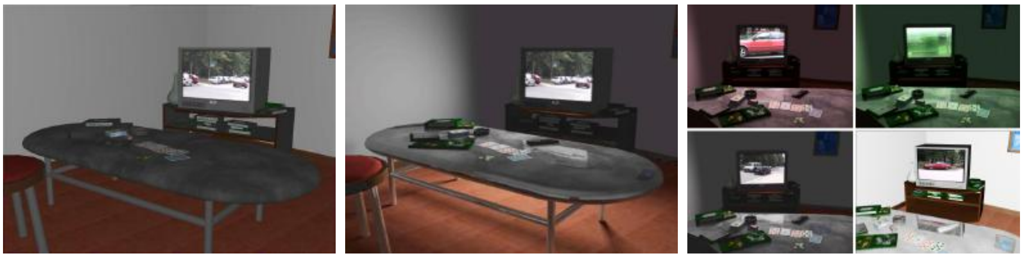


Figure 11.6: Lighting from live Video Textures. Left: A scene with a video texture on the TV set, without illumination. Center: With illumination from the TV set. Right: With four different frames of the video texture on the TV. Note how the illumination in the room automatically changes with a change of the frame displayed on the TV set, thereby producing the familiar "flickering" effect of a TV set at night.

by lighting a virtual scene by a real-world video (Figure 11.6). Using ray tracing for this task, all the advantages of ray tracing are now also available for these applications. For example, video billboards are a classical technique in VR/AR applications, but can now also be rendered with shadows, reflections, and refractions (see Figure 11.5). Similarly, the car in Figure 11.5 did not have to be simplified for this task, and can be rendered with the real environment being reflected in the car paint.

Finally, Figure 11.6 shows a video texture on the TV set: On the left image, the video texture simply appears on the TV set as it would appear in a typical VR application computing only direct effects. In the center and right images, the realtime ray tracer is used to compute global effects as well:

The TV set correctly reflects off the partially specular table, and the TV set – once used as a light source – casts smooth shadows into the scene. Once the "texture" on the TV set changes, the color and intensity of the illumination is correctly and automatically updated, producing the familiar "flickering" effect of a real TV set at night (see the right four images in Figure 11.6).

To realize these applications, the most complicated task was to seamlessly integrate a live video stream into the distributed ray tracer in a way that is both efficient and synchronized [Pomi03]. Once this has been accomplished, most of the actual applications have been straightforward to implement using programmable shaders. Currently, the main limitations seem to be networking issues (i.e. missing bandwidth, insufficient reliability, and high latency) in correctly and efficiently distributing the video stream to all the distributed rendering clients (for more details, please see [Pomi03]). Though these demand further attention before making that system useful for practical applications, most of these issues would simply disappear on a shared memory architecture, as currently becoming available even for the PC market.

Note that the presented applications are rather simple proof-of-concept applications. It has yet to be evaluated how far that concept can be pushed, for example by using much more complex scenarios, or by computing physically correct global illumination from a real-world environment. However, even these simple applications demonstrate how much realtime ray tracing can add to the toolbox of techniques for eventually "real-looking" virtual and augmented reality applications.

11.5 Interactive Lighting Simulation using Instant Global Illumination

Perhaps the most imporant application that becomes possible with realtime ray tracing is the interactive simulation of global illumination. In contrast to classical ray tracing – which is often limited to simple point light sources and direct diffuse illumination – global illumination computes the illumination in a scene in a physically correct way, by also including effects like smooth shadows from area light sources, indirect diffuse illumination, or color bleeding.

Virtually all global illumination algorithms make heavy use of ray tracing, and have traditionally spent a large fraction of their time in ray tracing computations. With the advent of realtime ray tracing, the cost for ray tracing drops dramatically, and opens the potential for global illumination at interactive frame rates. However, not all global illumination algorithms can equally benefit from realtime ray tracing, as the respective algorithms have to meet several constraints imposed by the design of a distributed realtime ray tracing engine. In fact, most of todays algorithms – like radiosity [Hanrahan91, Cohen93, Smits94, Bekaert99], (bidirectional) path tracing [Kajiya86, Lafortune96, Lafortune93, Veach97], or photon mapping [Jensen96, Jensen95, Jensen97, Jensen01] do not fit at all into such an environment.

Recently, Wald et al.[Wald02b] and Benthin et al. [Benthin03] (in cooperation with Thomas Kollig and Alexander Keller from Kaiserslautern University) have shown that it is indeed possible to interactively compute global illumination with a distributed interactive ray tracing engine using a specially designed global illumination algorithm. This algorithm builds on Kellers "Instant Radiosity" [Keller97], and has therefore lateron been dubbed "Instant Global Illumination".

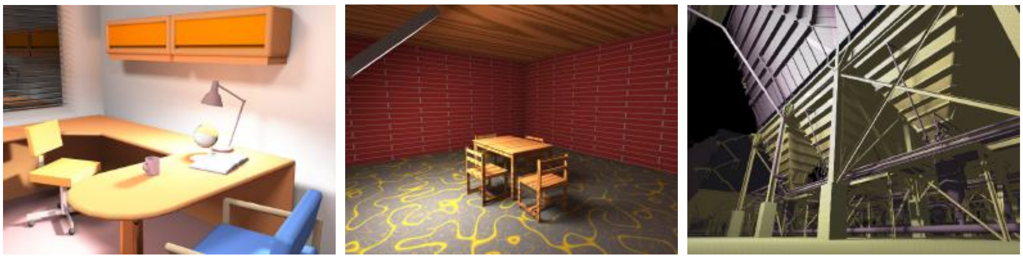


Figure 11.7: An interactive global illumination application in three different environments. From left to right: (a) an office environment with reflections, caustics, and other lighting and shading effects, (b) the extended Shirley 6 scene, with global illumination and complex procedural shaders, and (c) global illumination in a scene featuring three power plants with a total of 37.5 million triangles. All scenes render at several frames per second. For more examples of this technique, also see Figures 13.6, 13.7, 13.8, 13.12, and 14.8.

While the "Instant Global Illumination" method will be described in more detail lateron in this thesis (see Part III), Figure 11.7 already shows some example snapshots of scenes rendered with Instant Global Illumination on to of the RTRT/OpenRT engine: Instant Global Illumination captures the most important effects like smooth shadows from area lights, diffuse interreflection, reflections, refractions, and color bleeding. Even complex lighting effects like caustics can be simulated to a certain extent. Additionally, instant global illumination supports programmable shaders, efficient anti-aliasing, and allows to efficiently scale in both model complexity (of up to millions of triangles) and numbers of CPUs. Finally, instant global illumination does not have to rely on any precomputed values, but recomputes the entire solution every frame, thus allowing for arbitrary changes to geometry, material properties, or camera settings every frame.

The availability of global illumination at interactive rates – much more than classical ray tracing itself – bears a huge potential for practical applications in the design, architecture, simulation, and VR/AR industry, and is likely to enable a new set of graphical applications that cannot be realized with contemporary (i.e. other than realtime ray tracing) technology.

Part III

Instant Global Illumination

Chapter 12
Interactive Global Illumination

Even though classical ray tracing considers only direct lighting effects it already allows for highly realistic images that make ray tracing the preferred rendering choice for many animation packages. An even higher degree of realism can be achieved by including indirect lighting effects computed by global illumination.

Global illumination algorithms account for the often subtle but important effects of indirect illumination in a physically-correct way [Cohen93, Dutre01] by simulating the global light transport between all mutually visible surfaces in the environment. Due to the need for highly flexible visibility queries, virtually all algorithms today use ray tracing for this task.

As traditional ray tracing has historically been too slow even for interactively ray tracing images with simple shading, the full recomputation of a global illumination solution has been practically impossible. Now, with the advent of realtime ray tracing, it should become possible to also compute full global illumination solutions at interactive rates. However, most of todays global illumination algorithms do not fit into the parallel and distributed framework of a realtime ray tracing engine, and consequently cannot fully benefit from the available of a realtime ray tracing engine[1]. To solve this problem, we (together with Thomas Kollig and Alexander Keller from Kaiserslautern University) have developed the "Instant Global Illumination" method [Wald02b, Benthin03]. This Instant Global Illumination (IGI) method describes a global illumination algorithm that has been especially

[1] Note that this "parallel and distributed" argument is not restricted to our software implementation running on distributed-memory PC clusters. Instead it is likely that future hardware architectures for desktop systems will also be highly parallel, and will also not allow for shared-memory communication between different shaders and the application. In fact, many issues will be even more problematic for such architectures than for a software based system.

designed to fit the demands and restrictions of a realtime ray tracing engine, which therefore allows to compute the most important global illumination effects at realtime rates.

Before discussing this method, however, it is important to review related approaches. As realtime ray tracing has become available but recently, earlier research could not yet build on this technology, and had to rely on fundamentally different approaches.

12.1 Alternative Approaches

Roughly speaking, there are two fundamentally different approaches to global illumination: Sample-based techniques (i.e. ray- or path-based techniques) on one side [Kajiya86, Lafortune96, Lafortune93, Veach97, Jensen01], and finite-element based techniques on the other side [Hanrahan91, Cohen93, Smits94, Sillion94, Bekaert99, Stamminger99]. Because they do not depend on discretization, Ray- and path-based techniques generally allow for more flexibility and usually achieve superior image quality. However, before the advent of interactive ray tracing, these ray-based techniques have simply not been an option for *interactive* global illumination, as it was not even possible to interactively ray trace the geometric models themselves even without having to perform costly global illumination computations.

12.1.1 Radiosity Based Approaches

Thus, the first approaches to interactive global illumination have been based on finite-element techniques: Once the finite element solution hase been acquired, the readily lighted polygonal patches can easily be displayed interactively using rasterization graphics hardware. In its most trivial form, this approach consisted of a simple interactive display of a precomputed radiosity solution of the scene[2] However, relying on precomputed radiosity values only allows for simple walkthroughs, as any interactive change to the environment would require recomputing the radiosity solution. Except for trivial scenes, recomputing the whole solution from scratch every frame is not feasible at interactive rates.

[2] Obviously, the same approach is possible with interactive ray tracing. Additionally, using ray tracing for this task allows for adding reflections and refractions in a second pass (in the same spirit as [Chen91]). Furthermore, the ability to efficiently render highly complex scenes with ray tracing allows for using extremely fine tesselations.

12.1.1.1 Interactive Radiosity using Line Space Hierarchies

In order to avoid this full recomputation, Drettakis and Sillion [Drettakis97] have proposed to incrementally update a radiosity solution using a line space hierarchy. This hierarchy is generated by augmenting the hierarchical radiosity links with "shafts", each of which represents all the lines that pass through the two connected hierarchy elements. Traversing this data structure then allows for easily identifying the links that are affected by a dynamic change to the scene, and thus allows for quickly updating the radiosity solution. Additionally, it simultaneously allows for cleaning up subdivisions in the hierarchical radiosity solution that are not required any more after an update to the scene (e.g. for representing a shadow border that is no longer present after the occluder has been moved away).

However, the algorithms are quite complex. Additionally, like all radiosity systems the proposed system is limited to diffuse light transport, suffers from tesselation artifacts, and does not easily scale to complex geometries.

12.1.1.2 Instant Radiosity

In 1997, Keller presented a totally different approach to interactive radiosity. In "Instant Radiosity" [Keller97], a small set of virtual point lights (VPLs) is computed using a quasi random walk from the light sources. These VPLs are then used to illuminate the scene using a shadow algorithm. In instant radiosity, most computations (including the shadow generation) can be performed on the graphics card. For moderately complex scenes, this makes it possible to recompute fully lighted images at interactive rates without having to rely on complicated algorithms and data structures.

Additionally, instant radiosity avoids many of the typical tesselation artifacts: Instead of discretizing the geometry, instant radiosity performs discretization by using only a small number of discrete virtual point light positions. As such, the tesselation only becomes visible after one additional light transport step (i.e. the connection from the surface to the VPL): In the worst case, this may lead to a visible discretization artifacts (blocky shadows or hard shadow borders) in what should be smooth shadows. This is usually much less disturbing than directly seeing the discretization in the polygonal patches. Furthermore, performing the discretization only in the VPL positions completely avoids having to adapt the tesselation to match the lighting features (e.g. discontinuity meshing or visibility skeleton [Heckbert92, Lischinski92, Lischinski93, Durand97, Durand99]) which in practice is often problematic.

With these properties, instant radiosity has significant advantages over traditional radiosity. However, relying on rasterization graphics hardware instant radiosity is limited to purely diffuse scenes, and also suffers from lacking performance in realistically complex scenes. Finally, it is not clear how to use instant radiosity in a hierarchical scheme such as hierarchical radiosity [Hanrahan91, Cohen93, Sillion94] and clustering [Smits94, Bekaert99].

12.1.2 Subsampling-based Techniques

Though radiosity-based systems already allow for interactively rendering dynamic scenes with indirect lighting, they all suffer from similar problems: First, they are rather slow and often do not scale to reasonable scene complexities. Additionally, radiosity based systems are inherently limited to purely diffuse light transport, which often gives them a rather 'flat' and unrealistic appearance.

To avoid these limitations, most offline global illumination systems use ray- and path-based techniques. These, however, usually demand tracing hundreds of rays for each pixel, and many millions of rays for the whole frame, which obviously is not affordable at interactive rates. Even if it is not possible to recompute every pixel every frame, interactivity can still be achieved by using approximative or subsampling-based techniques. The basic idea behind these techniques is to subsample the full solution (e.g. by computing only a fraction of all pixels) and using a separate display thread for interactively reconstructing an image from the subsampled information. As the display process usually runs much faster than the sampling process (usually 10 – 100 times as fast), the display thread has to cope with heavy subsampling of the image, which often results in artifacts.

12.1.2.1 Render Cache

One of the first systems to use this approach was Walter et al.'s "Render Cache" [Walter99, Walter02]. The render cache stores a cache of previously computed illumination samples – roughly twice as many as pixels in the image. For each new frame, these are reprojected to the new camera position, and stored in the frame buffer. As this can result in artifacts (e.g. holes in the image or disocclusion) several heuristics have to be applied to reduce such artifacts.

The rendering thread runs asynchronously and decoupled from the display and reconstruction thread, and simply generates new illumination samples as fast as possible. Such generated samples are then inserted into the render cache (thereby replacing some old samples) and can be used for future frames. The main limitation of the render cache is the speed with which it can reproject the old samples and reconstruct the new frame. The high cost for these operations – which is usually linear in both framerate and number of pixels – has limited the original system to only moderate resolutions. Though recently proposed optimizations [Walter02] allowed for significant speedups, generating full-screen images is still too costly for realtime frame rates.

The render cache becomes beneficial for rendering algorithms where the cost for computing a new sample is very high. For a fast ray tracer with a simple illumination model it is often cheaper to simply trace a new ray than reprojecting and filtering old samples. Similar techniques have also been proposed in the form of Ward's Holodeck [Larson98, Ward99], and Simmons' et al.'s Tapestry [Simmons00] system.

12.1.2.2 Shading Cache

A similar approach has recently been proposed by Tole et al. [Tole02]. Instead of using an image-based technique for reconstructing the image from previously computed samples, they use an object-based technique that uses triangles for representing the illumination samples. Image reconstruction and interpolation between shading samples can then be performed using graphics hardware, which allows for producing realtime frame rates at full-screen resolutions.

However, their approach suffers from similar problems as the render cache: Depending on old (and thus potentially outdated) samples results in disturbing delays until an interaction takes effect: While some geometric object can be moved at realtime frame rates, the global effects caused by this object – e.g. its shadow – will be computed much slower, and will "follow" the object with noticeable latency of up to several seconds, and with visible artifacts during the transition phase.

This makes it hard to use the shading cache in totally dynamic scenes with constantly changing illumination from dynamic lights, scenes, and materials. Finally, the shading cache requires as least one sample per visible polygon, and thus is not suitable for highly complex geometries.

12.1.2.3 Edge-and-Point Images (EPIs)

The newest variant of this idea of interactively generating high-quality images based on using sparse samples has recently been proposed by Bala et al. [Bala03]. The "edge and point image" (EPI) essentially is an extension of the render cache (Section 12.1.2.1) that uses – and preserves – analytic discontinuities during reconstruction of the image.

The EPI analytically finds the most important shading discontinuities – in its current form these are object silhouettes and shadow boundaries – using efficient algorithms and data structures, and then respects these discontinuity edges during reconstruction. This allows for high-quality reconstruction even including anti-aliasing at interactive rates.

12.1.3 Hybrid Ray-/Patch-based Techniques

In order to avoid the qualitative limitations of pure radiosity approaches, several approaches have been undertaken to augment radiosity solutions with specular effects such as reflections and refractions, for example by using corrective textures [Stamminger00] or corrective splatting [Haber01]. These techniques are similar to the above-mentioned subsampling-approaches, and have already been covered in greater detail in the 2001 Eurographics State of the art report (STAR) on interactive ray tracing, see [Wald01b].

12.1.3.1 Fast Global Illumination including Specular Effects

Another interesting way of combining radiosity with specular effects has been proposed by Granier et al. [Granier01]. They augment a radiosity solution with specular effects – like e.g. caustics – using particle tracing. In its core, their system uses hierarchical radiosity [Hanrahan91] with clustering [Smits94] for quickly and efficiently computing the diffuse light transport. Non-diffuse effects are computed by integrating particle tracing into the gather step of hierarchical radiosity. Particle tracing is performed only where necessary, by shooting particles only over links that connect to specular surfaces.

In simple scenes, their system allowed for interactive viewing of up to 2 frames per second for global diffuse illumination and caustic effects. However, interactive viewing required using graphics hardware, thereby limiting the system to diffuse plus caustics only. Reflection and refraction could not be supported in interactive mode, resuling in some “dull” appearance [Granier01]. Though combining their technique with the render cache [Walter99] (cf. Section 12.1.2.1) allowed to push framerate and add reflections and refractions, the render cache itself introduced other artifacts.

12.1.3.2 Selected Photon Tracing

A different way of interactively computing global illumination solutions has recently been proposed by Dmitriev et al [Dmitriev02]: In “selective photon tracing”, radiosities of triangular patches are computed by shooting photons from the light sources into the scene and depositing their energy in the vertices of the triangular mesh (see Figure 12.1). The lighted triangles can then be displayed interactively with graphics hardware. To hide some of the tesselation artifacts, direct illumination from point lights is computed separately on the graphics hardware using hardware shadow algorithms [Crow77].

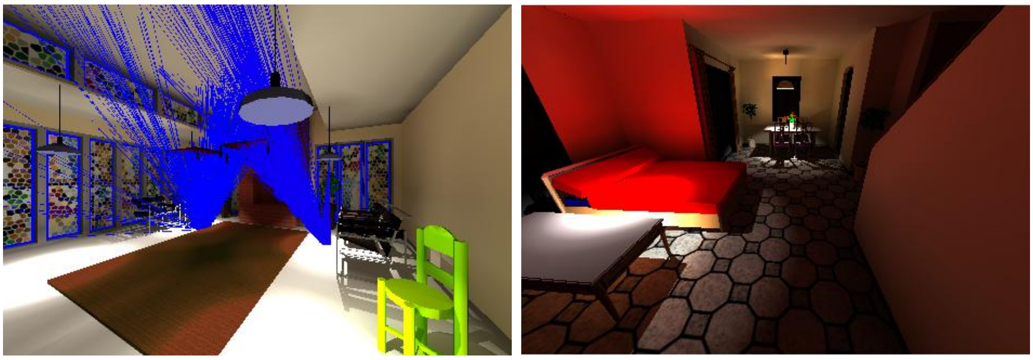


Figure 12.1: Selected photon tracing. Left: Once some pilot photons have detected a change in the scene geometry, similar photons (blue) are selectively re-traced. Right: An example frame while interactively changing the scene.

Due to the limited speed of the photon tracer, only a certain amount of photons can be retraced every frame. Therefore, selected photon tracing uses a clever way of determining which photon paths have probably been affected by a scene update: A small subset of photon paths (called "pilot photons") is shot into the scene to determine which parts of a scene have been updated. If a pilot photon hits an object that it did not hit in the previous frame (or vice versa), selected photon tracing selectively re-traces similar photons by generating only photon paths that are similar to the original path (see Figure 12.1). Similar photons are generated by exploiting similarities in the quasi random number generator used for determining the photon paths.

Being a hybrid of both radiosity and subsampling based techniques combines many of the advantages of both techniques, and allows for interactively displaying scenes with global illumination while allowing for interactive manipulations to the scene.

However, selected photon tracing also inherits some of the inherent problems of radiosity and subsampling: First, the speed decreases linearly in the number of patches, allowing only a rather coarse tesselation that does not capture high-frequency details such as caustics or indirect shadow borders. Second, as only a subset of all photons is re-traced every frame, drastic changes in the illumination caused by user interaction can take several seconds to take effect.

12.2 Realtime Ray Tracing for Interactive Global Illumination – Issues and Constraints

All the approaches discussed above had to be undertaken because compute power and ray tracing performance did not suffice to compute full ray tracing based global illumination solutions at interactive rates. With the recent availability of realtime ray tracing, however, it should eventually become possible to compute such images interactively: In theory, most global illumination algorithms spend most of their time tracing rays, thus combining such an algorithm with a realtime ray tracer should “automatically” result in interactive global illumination performance.

In practive however, global illumination algorithms are inherently more complex than classical ray tracing and thus not all algorithms will automatically benefit from a much faster ray tracing engine. In order to take maximum profit from the availability of fast ray tracing, appropriate global illumination algorithms have to be designed to meet several constraints:

Parallelism: Future realtime ray tracing engines will probably (have to) make heavy use of the inherent parallelism of ray tracing. For classical ray tracing, parallelism can be exploited easily by computing pixels separately and independently. Many global illumination algorithms, however, require reading or even updating global information, such as the radiosity of a patch [Cohen93], entries in a photon map [Jensen01], or irradiance cache entries [Ward92]. This requires costly communication and synchronization overhead between different ray tracing ’units’, which quickly limits the achievable performance.

Efficiency at small sample rates: Even the abitiliy to shoot millions of rays per second leaves a budget of only a few rays per pixel in order to stay interactive at non-trivial frame resolutions. Thus, an algorithm must achieve sufficiently good images with a minimum of samples per pixel. Given the performance of current realtime ray tracing engines, only an average of about 50 rays per pixel is affordable. Thus, the information computed by each ray has to be put to the best possible use.
Note that this number of affordable rays per pixel is likely *not* to change significantly in the near future: Though future ray tracing systems (especially ray tracing hardware) will offer even higher ray tracing performance, it is likely that for practical applications this performance increase will first be spent of higher resolutions and frame rates.

Realtime capabilities: For real interactivity – i.e. arbitrary and unpredictable changes to the scene made by a user (including geometry, materials and light sources) – algorithms can no longer use extensive preprocessing. Preprocessing must be limited to at most a few milliseconds per frame and cannot be amortized or accumulated over more than a few frames as the increased latency of lighting updates would become noticeable.

Focus on Ray Tracing: The availability of a realtime ray tracing engine can only save time previously spent on tracing rays. Thus, performance must not be limited by other computations, such as nearest-neighbor queries, costly BRDF evaluations, network communication, or even random number generation.

Independence From Geometry: In order to fully exploit the ability of ray tracing to scale to complex geometries (see Chapter 7), the global illumination algorithm itself must be independent from geometry, and may not store information on individual patches or triangles.

Within the above constraints most of todays global illumination algorithms cannot be implemented interactively on a realtime ray tracing engine: All radiosity style algorithms [Cohen93, Hanrahan91, Smits94, Drettakis97, Granier01] require significant preprocessing of global data structures which seems impossible to implement under these constraints. In principle, it should be possible to use the ray tracer for augmenting one of the above interactive radiosity systems with specular effects and accurate shadows in the spirit of [Wallace87]. This, however, would not easily fit into the distributed framework that the ray tracer is running on, and would still suffer from tesselation artifacts.

Pure light-tracing or path-tracing [Kajiya86] based approaches benefit mostly from fast ray tracing, but usually suffer from random noise. For decent image quality, they would require far too many rays per pixel at least for non-trivial lighting conditions. Finally, photon mapping [Jensen96, Jensen95, Jensen97, Jensen01] requires costly preprocessing for photon shooting and creation of the kd-trees as well as expensive nearest neighbor queries during rendering. It also uses irradiance caching, which imposes similar problems.

Chapter 13
Instant Global Illumination

The discussion in the preceding chapter shows that a new, appropriately designed algorithm is required to take advantage of a fast ray tracing engine for interactive global illumination.

13.1 The Instant Global Illumination Method

Because of this, we (together with Thomas Kollig and Alexander Keller from Kaiserslautern University) have developed a new global illumination framework – called "Instant Global Illumination" (or, shorter: IGI) – that has explicitly been designed to run efficiently on a realtime ray tracing engine. In essence, Instant Global Illumination is a combination of 6 different techniques that have been combined that very well fits the constraints outlined in the previous chapter (also see Figure 13.1):

- **Instant Radiosity (IR)** for approximating the diffuse illumination in a scene using virtual point lights (VPLs),

- **Realtime Ray Tracing** for quickly and accurately performing the IR shadow computations and for adding reflections and refractions,

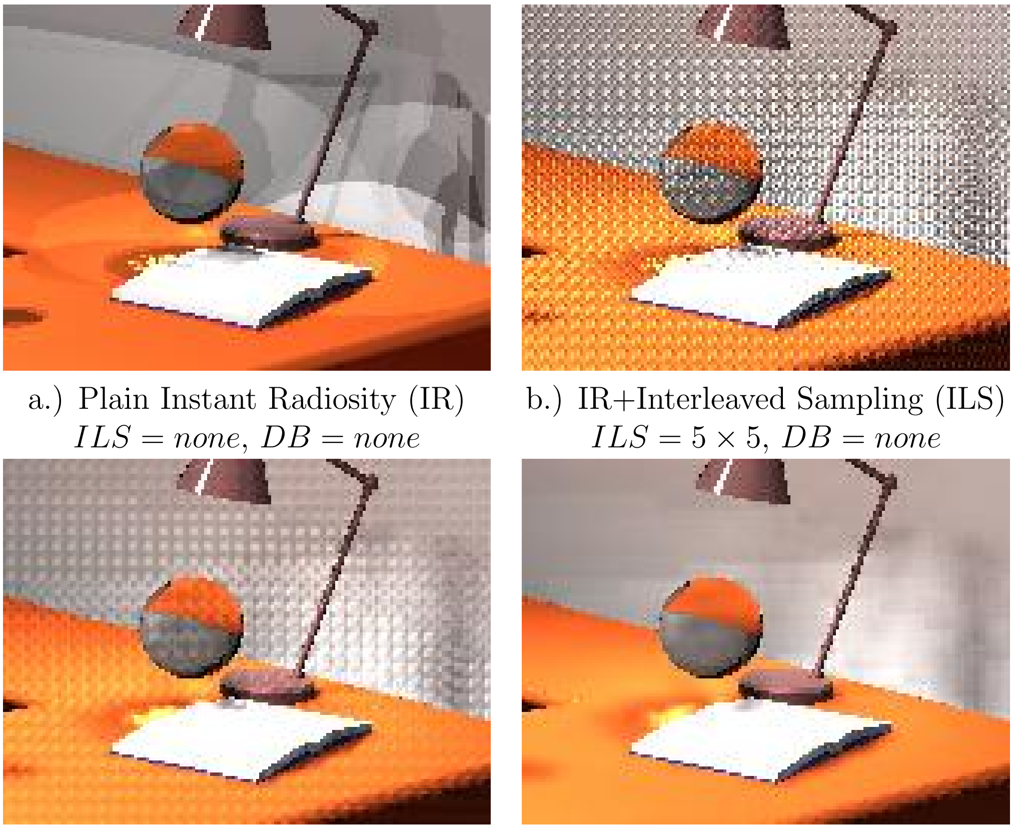
- **Hashed Caustic Photon Mapping** for optionally adding simple caustics,

- **Interleaved Sampling** for breaking up the discretization artifacts of instant radiosity and trading them for structured noise,

- **Discontinuity Buffering** for removing this structured noise by an image-based filtering technique,

- **Quasi Monte Carlo** for generating optimally distributed VPL sets.

In the following, we will briefly describe each of these basic ingredients.



a.) Plain Instant Radiosity (IR)
$ILS = none$, $DB = none$

b.) IR+Interleaved Sampling (ILS)
$ILS = 5 \times 5$, $DB = none$

c.) IR+ILS+Discontinuity Buffer
$ILS = 5 \times 5$, $DB = 3 \times 3$

d.) IR+ILS+Discontinuity Buffer
$ILS = 5 \times 5$, $DB = 5 \times 5$

Figure 13.1: Interleaved sampling and the discontinuity buffer: All close-ups have been rendered with the same number of shadow rays for each pixel. a) Plain instant radiosity with only one set of point light sources and caustic photons (computed with ray tracing and photon mapping to add reflections, refractions and caustics). b) $5 \times 5 = 25$ *independent such sets (each of the same size as a) have been interleaved. c.) and d.) Discontinuity buffering to remove the ILS artifacts. Choosing the filter size appropriate to the interleaving factor completely removes the structured noise artifacts: b.) no filtering at all, c.) smaller filter than interleaving size, d.) matching filter and interleaving size.*

13.1.1 Instant Radiosity

As just mentioned, Instant Global Illumination builds on Kellers Instant Radiosity [Keller97] as a core technique: Using instant radiosity (IR), the illumination in a scene is approximated by a small set of "virtual point lights" (VPLs) that are generated in a preprocessing step by performing a random walk from the light sources. During rendering then, the irradiance at a surface point is computed by casting shadow rays to all VPLs and adding their contribution. For a graphical explanation of this procedure, see Figure 13.2.

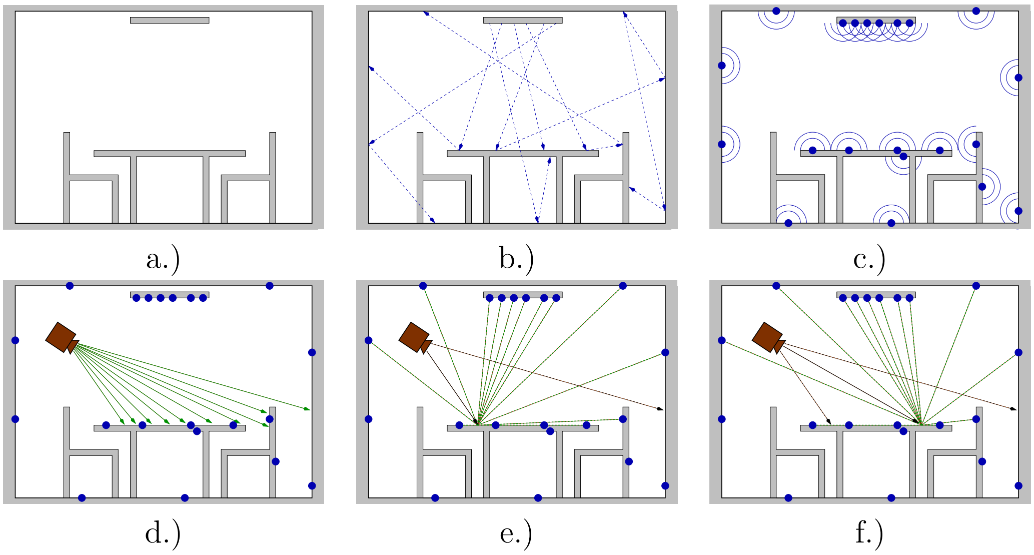


Figure 13.2: Instant Radiosity: In order to compute instant radiosity in a scene (a), a small number of particles is started from the light sources, and traced into the scene (b). At each particle hitpoint, a "virtual point light" (VPL) is generated, which acts as a point-light source with a cosine-weighted directional power distribution (c). In the main rendering step, rays are cast from the camera into the scene (d) (probably including reflection and refraction rays, but without *diffuse bounces). At each eye ray's hitpoint, the irradiance can be calculated by summing up the contribution from all (visible) VPLs, which requires shooting shadow rays from each hit point to all VPLs (e.g. e and f).*

Similarly to other radiosity-style techniques, instant radiosity (IR) generates smooth images: Being in principle an unbiased algorithm [Keller97], IR will eventually converge to a smooth image after taking enough samples. Before reaching that smooth solution however any intermediate result suffers from a residual error. For typical Monte Carlo-based rendering algorithms this random error changes from pixel to pixel, and thus is usually quite visible as random pixel noise. In contrast to this, radiosity-style techniques usually trade this random noise for discretization error.

With traditional radiosity-style algorithms, this discretization error is usually clearly visible in the discretized, polygonal patch representation of the radiosity solution. With instant radiosity, however, the discretization is not performed on the surfaces themselves, but rather by using a discrete set of VPLs. As each of these VPLs contributes to the full image, this discretization error is smoothly distributed over the entire image plane, and as such often not (as) visible to the plain eye. Unfortunately some discretization error is still visible in shadow boundaries: Each individual VPL is a point light and as such casts a hard shadow border. If enough VPLs are used, the overlapping hard shadows again appear as a smooth shadow. When using too few VPLs, however, these hard shadow borders are clearly visible, as can be seen in Figure 13.1a.

In general however, instant radiosity inherits the smoothness from radiosity algorithms, while mainly avoiding its surface discretization problems.

13.1.2 Realtime Ray Tracing

Apart from these "technical" advantages, Instant Radiosity as a core algorithm perfectly fits the demands of a realtime ray tracer: It performs most of its time evaluating visibility from point light sources, which generates highly coherent rays and thus allows the ray tracer to use the fast packet traversal code (see Section 7). Mostly shooting shadow rays also implies that the usually high cost for shading can be amortized over many shadow rays, and that the system perfectly benefits from any increase in ray tracing speed. As shadow rays – especially coherent ones – are usually less costly than 'standard' rays, the ray tracer can perform at its best.

The preprocessing cost for Instant Radiosity is minimal (requiring but the generation of only a few dozen VPLs) and thus lends itself easily to parallelization. Finally, IR does not need access to global data (except for the VPLs which can easily be generated on each client), is totally independent of geometry, and parallelizes trivially per pixel.

Being implemented on top of a realtime ray tracing system also allows Instant Global Illumination to easily and cheaply integrate important specular effects like reflections and refractions, which have traditionally been a problem in fast global illumination. Finally, the combination with interleaved sampling and discontinuity buffering (see below) also only becomes possible through the use of a ray tracer, which can selectively use different VPLs for different pixels.

13.1.3 Fast Caustic Photon Mapping using Hashed Photon Mapping

In order to be able to support at least simple caustics, instant global illumination can also be combined with caustic photon mapping. To reduce the high cost for photon mapping (which can be several times as expensive as shooting a ray), we have proposed a method that approximates photon mapping by storing the photons in a grid instead of a kd-tree [Wald02b]. If we fix the photon query radius in advance (instead of determining it implicitly by finding a fixed number of photons as usually done in photon mapping [Jensen01]), one can select the grid radius such that exactly 8 voxels have to be accessed to find all photons in the search radius.

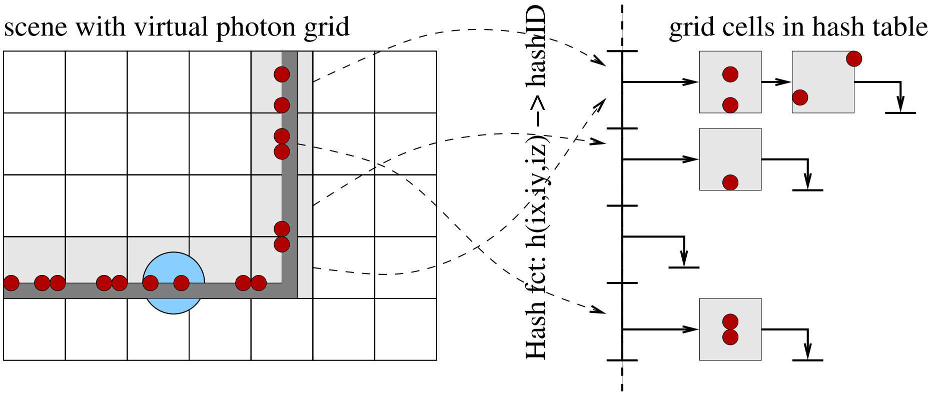

scene with virtual photon grid
Hash fct: h(ix,iy,iz) -> hashID
grid cells in hash table

Figure 13.3: Hashed photon mapping: All photons are stored in a hashed grid data structure. Each grid cell (ix, iy, iy) has a unique hash ID $h(ix, iy, iz)$ with which it is stored in a hash table. Only those cells that actually receive any photons (light gray) get stored in the hash table, which allows for a very fine grid structure with small memory consumption. Some amount of hashing collisions can be tolerated, but should be minimized by making the hash table sufficiently large (roughly as large as the number of grid cells occupied by photons). If the grid resolution can be guaranteed to be bigger or equal to the maximum query ball diameter (light blue ball), at most 8 hash-lookups are required (obviously only four lookups in this two-dimensional example).

Storing a complete 3D grid of the required density is obviously not possible due to limited memory space. As such, we store the photons in a "hashed" grid, in which each voxel gets a unique ID with which it can be looked up in a hash table. Voxels are stored in the hash table only if they actually contain valid photons. As there are typically rather few such voxels[1], even reasonably

[1] Voxels can receive photons only if they overlap surface geometry.

high grid resolutions can be used with a very small memory consumption.

Storing photons in this hashed grid can be performed quite quickly (the photon only has to be added to the respective hashed voxel cell), which allows for completely rebuilding the data structure every frame. As such, we can also interactively change the photon query radius from frame to frame.

Using this simplified form of photon mapping, at least simple caustics can be supported. However even this quite simplified form of photon mapping is quite costly due to the need for shooting thousands of rays for generating the photons. This higher preprocessing cost usually leads to scalability and performance problems, and thus is only useful for extremely simple scenes. Efficient support for complex caustics – especially in the context of interactive frame rates and in a highly parallel setup – still requires further investigation.

13.1.4 Interleaved Sampling

Even ignoring any preprocessing cost for generating VPLs or caustics, only a small number of shadow rays to VPLs is affordable in each pixel in order to remain interactive. As discussed above, this small sampling rate leads to visible discretization error in the form of hard shadow boundaries (see Figure 13.1a).

These hard shadow boundaries can be "broken up" by using Interleaved Sampling [Keller01]: Instead of using the same set of n VPLs for every pixel, the algorithm generates $N_{ILS} \times N_{ILS}$ (usually 3×3) such sets (of N_{VPL} VPLs each). These different VPL sets are then interleaved between pixels: Neighbouring pixels use different sets, and the same set is repeated every N_{VPL}'th pixel (see Figure 13.4). The only additional cost for this scheme is that instead of only one VPL set we now have to generate $N_{VPL} \times N_{VPL}$ of these sets; the main cost factor during rendering – i.e. the number of shadow rays per pixel – does not change.

Even though the rendering cost does not increase, the image quality is greatly enhanced: As neighbouring pixels use different VPL sets, sharp shadow boundaries are broken up. Roughly speaking, using 3×3 different sets of VPLs effectively increases the number of VPLs used for computing an image by a factor of 9. In a certain sense, this provides "free mileage", as the visual appearance of the image is improved (compare Figures 13.1a and 13.1b) without increasing the cost per pixel.

However, interleaved sampling can not actually *remove* any discretization error, but instead only trades it for structured noise. Though this is usually better than not performing interleaved sampling at all, the structured noise is still clearly visible, especially in otherwise smooth image regions, see Figure 13.1b.

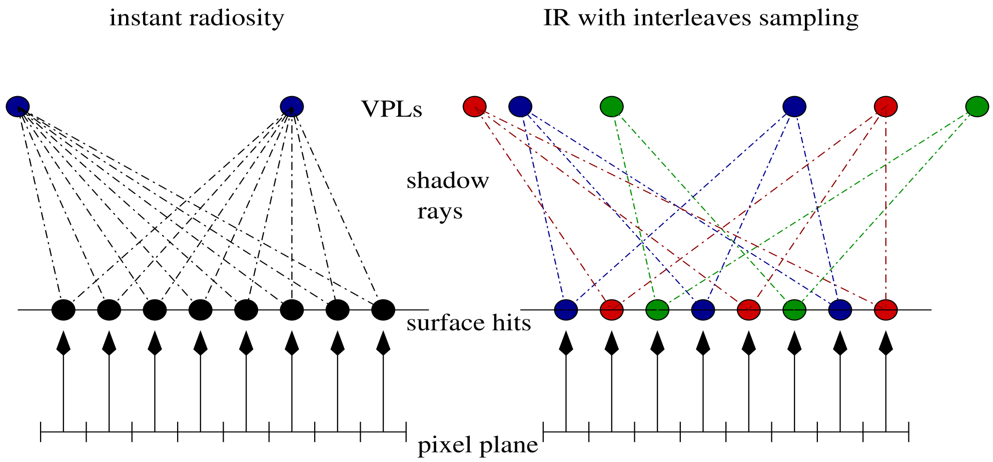


Figure 13.4: Interleaved Sampling: a.) In instant radiosity, each pixel uses the same set of VPLs (two in this example). b.) With interleaved sampling, neighbouring pixels use different VPL sets (which are repeated every N_{ILS} pixels, and cast shadow rays only to their respective VPL set ($N_{ILS} = 3$ sets of two VPLs each in this example). This essentially increases the number of VPLs by a factor of N_{ILS} ($N_{ILS} \times N_{ILS}$ in 2D), removes the correlation between neighbouring pixels, and thus breaks up hard shadow borders.

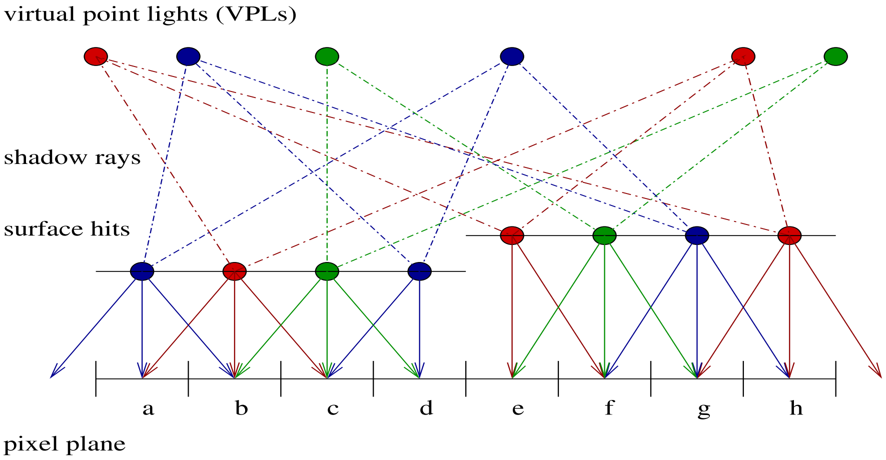


Figure 13.5: Discontinuity Buffering: After interleaved sampling, the image suffer from structured noise artifacts, as the irradiances of neighbouring pixels are computed with different VPLs. In continuous image regions (pixels a.–c. and f.–h.), the discontinuity buffer removes this structured noise by filtering the neighouring pixel irradiances. No filtering is performed in discontinuous regions, e.g. where the distances to the hit points vary too much (d.–e.).

13.1.5 Discontinuity Buffering

The structured noise resulting from interleaved sampling can then be removed by "discontinuity buffering", a technique that removes structured noise by filtering the image, but which uses several heuristics in order not to filter across image discontinuities (see Figure 13.5). In continuous image regions (pixels a.–c. and f.–h. in Figure 13.5), the discontinuity buffer filters the neighouring pixel irradiances. If the filter size is chosen to exactly match the size of the interleaving pattern the structured noise can be entirely removed (see Figure 13.1b–d), as then each pixel is once again influenced by all VPLs (even though it does not have to compute shadow rays to each VPL).

In order to avoid excessive blurring across discontinuities, the discontinuity buffer uses several heuristics to detect discontinuities in the image plane (e.g. the distance to the hit point), and does not filter across these (pixels d. and e.). As such, discontinuous regions have slightly less image quality, as they still show the structured noise from interleaved sampling. At these discontinuities, however, this structured noise is usually less visible than it would be on smooth, continuous surfaces. Even though, discontinuity buffering obviously can only help in continuous image regions, and is problematic for scenes that do not feature any smooth areas. Fortunately this is rarely the case for practical scenes. However, some structured noise can remain in areas where the heuristics do not permit filtering, e.g. on highly curved sufaces.

Note that even in regions where the filtering *is* actually performed, it is *only the irradiances* that are filtered, *not* the actual pixel values themselves. As such, the discontinuity buffer does *not* blur across material boundaries, nor does it blur textures, as the diffuse component is only multiplied to the irradiance *after* the irradiances have been filtered.

In smooth image regions, the combination of interleaved sampling and discontinuity buffering improves the performance by roughly an order of magnitude (in fact, a factor of $3 \times 3 = 9$). This allows for achieving reasonably high image quality even when using only rather few VPLs – and thus shadow rays – per pattern.

In the current implementation, we only support a filter size of 3×3 pixels. Instead going to 5×5 might give another factor of $\frac{5\times5}{3\times3} \approx 2.8$ in image quality, but is also more likely to generate visible artifacts due to the increased blurring. However, more blurring may still be tolerable for larger resolutions. As such, larger filter sizes of 5×5 or 7×7 may be beneficial when eventually using instant global illumination at fullscreen resolutions.

13.1.6 Quasi Monte Carlo for Generating the VPL Sets

For generating our VPL sets, we use the quasi-monte carlo techniques as descibed in [Keller98, Keller03, Kollig02]. While – for the correct functioning of the method – it does not actually matter wheter the VPL sets are generated with pseudo-random numbers or with quasi-monte carlo techniques, low-discrepancy point sets (see e.g. [Keller98, Keller03, Kollig02]) are usually preferrable. For example, pseudo-random numbers usually suffer from "sample clustering". In the context of instant radiosity sample clustering results in several VPLs being placed unnecessarily close to each other, which in turn requires to shoot several costly shadow rays to compute almost the same effect. In contrast to this, low-discrepancy point typically result in a much "better" distribution of the VPLs, which usually results in faster convergence, or – in other words – in better image quality when using the same number of VPLs. For our approach, this is especially interesting because of the small number of samples that are affordable at interactive rates.

Apart from faster convergence, low-discrepancy point sets can often be computed faster then the typical "`drand48()`" random number generator. Furthermore, quasi-random numbers lend well to parallelization, as the same quasi-random sequence can be easily reproduced by different machines without the need for any communication at all.

Last but not least, quasi monte carlo techniques allow for generating the different VPLs sets in an optimal way: In order to use the combination of instant radiosity, interleaved sampling and discontinuity buffering as described above it usually does not matter how exactly the VPL sets have been generated. For example, one can either generate $N \times N$ independent sets of M randomly generated VPLs each, or can instead generate one large set of $N \times N \times M$ VPLs that are then randomly distributed into $N \times N$ sets. With quasi random techniques (to be more specific, with *TMS sequences* [Keller03]) however it is possible to generate the VPL sets such that – while each VPL set on its own already uses a low-discrepancy set – all VPL sets combine nicely to a higher dimensional pattern. As this optimal combination of multiple sets so far is only possible for 2-dimensional point sets [Keller03], we follow Kollig et al.'s approach [Keller03, Kollig02] of using padded replication sampling to "extend" this technique to higher dimensions. In essence, we use *exactly* the same techniques as described by Keller and Kollig in [Keller03, Kollig02].

13.2 Implementation Issues & Design Decisions

In most aspects, implementing Instant Global Illumination on top of a distributed ray tracing system is rather simple. All global illumination computations are implemented as "shaders" applied to the surfaces and to the pixels of the image (see Chapter 10). From the application point of view, the global illumination system is a shader like any other.

Being realized as a shader, the global illumination code can also abstract from the distributed framework and from issues such as dynamic scenes, which are handled transparently by the underlying ray tracing engine (see Section 9). As such, dynamic scenes or interactive user modifications to the scene are automatically supported (as long as the global illumination code can generate a new image from scratch every frame), and does not require any special handling.

13.2.1 Filtering on the Server vs. Filtering on the Clients

Most computations are spent on shooting highly coherent shadow rays, which perfectly fits the underlying ray tracing engine, and which can be easily implemented as a programmable shader. The most critical design decision for a distributed implementation is to decide where the discontinuity buffering is to be performed.

If this final filtering pass is to be performed on the final image (i.e. on the server), it is possible to assign only pixels with the same interleaving pattern to each client. Thus, each client only has to generate the VPLs and photons for its respective interleaving set. If this preprocessing phase is quite costly – e.g. if several thousand rays have to be shot for producing convincing caustics – this efficiently avoids replicating the preprocessing cost on all clients, and significantly improves scalability. Essentially, each client only has to compute one nineth of all VPLs and photons that are eventually used for computing the final image. However, this strategy implies that clients do not have access to neighbouring pixel values – as these would have to be computed by different interleaving sets – and that therefore filtering has to be performed on the server. This however can create a scalability bottleneck, as the server can only receive and filter a quite limited number of pixels per second.

The alternative strategy is to perform the filtering on the clients. This effectively avoids the server bottleneck, but requires clients to have access to neighbouring pixels. First of all, this incurs some overhead at tile boundaries where clients have to compute pixels across their tile boundaries. Even worse, this requires each client to generate all VPLs and photons for all interleaving patterns. While generation of the VPLs is cheap enough to be replicated on each client, generating a sufficient number of caustic photons is no longer affordable in this strategy.

13.2.2 The Original Instant Global Illumination System

Therefore – in order to be able to compute caustics – the original IGI system was performing filtering on the server (see Figures 13.6 and 13.7). Due to interactivity constraints, the number of caustic photons that was affordable to be retraced per frame was limited to only several hundred to a few thousand[2]. Fortunately, the combination of interleaved sampling and discontinuity buffering also increased the quality of the caustics, and as such allows for reasonably good caustics even with such few photons.

13.2.2.1 Caustic quality

Even though simple caustics can thus be supported, highly detailed caustics – such as a peaked cusp resulting from a cup, glass, or metal ring (see [Jensen01]) – can not be generated with as few photons as are affordable during interactive update rates[3].

13.2.2.2 Overall image quality

Apart from its limitations in caustic quality, the original IGI system already supported all of the most important kinds of light transport. For example, Figure 13.7 shows an animation sequence in the office scene. In this animation sequence, a glass ball rolling over the table and casts a caustic while doing so. Furthermore, a book is being moved towards the light source, casting a shadow that automatically changes from sharp to smooth while moving away from the desk. Most imporantly, all indirect illumination is correcty accounted for: As the book moves closer to the light source, its bright surface automatically starts to act as an indirect light source, thereby illuminating the cupboard from below and casting shadows and illumination patterns on the right wall (see Figure 13.7c).

[2] Theoretically it is possible to exploit temporal coherence by only re-tracing a fraction of the photons in each frame, and thus also using “old” photons in each frame. Though this has already been shown to work reasonably well for animations (see e.g. [Myszkowski01, Dmitriev02, Damez03], this in practice is hard to implement in our system, due to its parallel and distributed nature.

[3] Of course, such high-resolution caustics can still be added incrementally during times at which the scene does not change, as has also been proposed in [Purcell03].

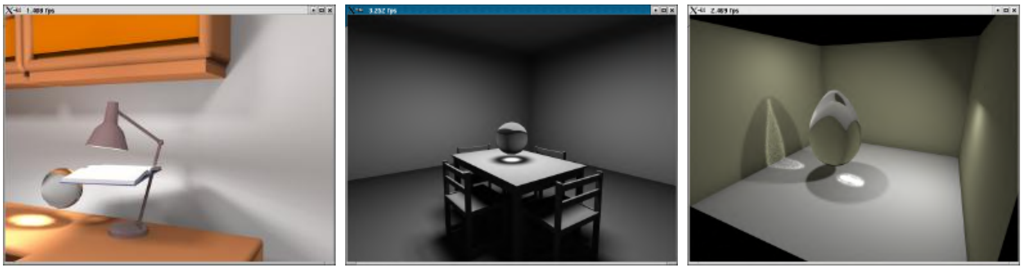


Figure 13.6: Caustics in the original Instant Global Illumination system. While really complex caustics (such as the sharp cusp from a metal ring) can not be sufficiently be represented with as few caustic photons as affordable during interactive frame rates, simple caustics can be well represented.

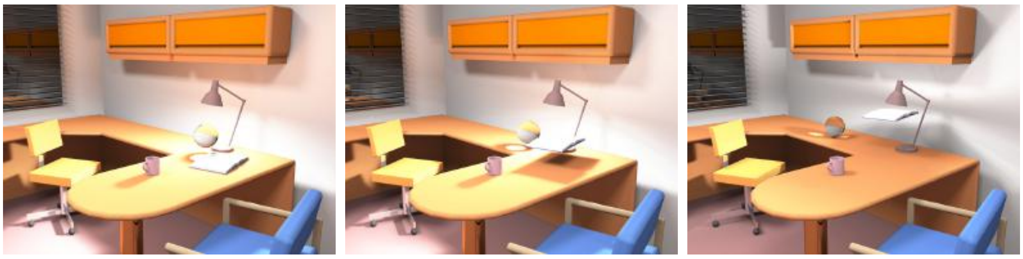


Figure 13.7: Instant Global Illumination in an animated office scene. In this animation sequence, the glass ball is rolling over the table, and the book is moved towards the light source. Especially note how the caustic is correctly moving, how the shadow from the book is changing from sharp to smooth, how the indirect illumination from the book casts an indirect shadow on the wall, and how all these effects are correctly reflected in the window.

During the entire animation, all these effects are automatically reflected in the window[4], or in the other reflective parts of the scene (e.g. in the legs of the rolling chair).

13.2.2.3 Scalability problems

As expected by the discussion in Section 13.2.1, filtering on the server created a bottleneck that in practice limited performance to roughly 5 frames per second at 640 × 480 pixels [Wald02b]. However, the original system was limited *only* in the maximum frame rate, *not* in the number of clients used for reaching this frame rate. As such, the system still scaled very well in

[4] The window has been modelled with an artifically high reflectivity coefficient to best show the effect

image quality: Roughly speaking, using twice as many CPUs, allowed for using twice the number of VPLs while still achieving the same frame rate. At the same frame rate, this means that adding more PCs still pays off through higher rendering quality.

13.2.2.4 Performance limitations

Though the original IGI system already achieved interactive performance, it was mainly designed as a proof-of-concept system, and did not exploit the full performance of the ray tracer. For example, though it was explicitly designed to generate coherent rays to point light sources, the original implementation did not yet use the fast SSE packet traversal code for tracing these rays. Thus, the performance of the original system was severely limited, thereby allowing only a rather small number of VPLs at interactive rates. As this resulted in visible artifacts in the image, the IGI system additionally allowed for progressively refining the quality of an image as soon as user interaction stopped: In that case, the refinement process automatically updated the seeds for the quasi random number generators, and progressively increased the number of VPLs, caustic photons, number of diffuse interreflections, and samples per pixel. This eventually resulted in high-quality, artifact-free, and antialiased images after only a few seconds of convergence.

13.3 A Scalable Implementation

However, due to the limited performance of the original system, this image quality could not be maintained during user interaction. In order to remove these limitations, Benthin et al. [Benthin03, Wald03e] have recently proposed several important improvements that now allow for improved image quality, significantly higher performance, and added features.

13.3.1 Improved Performance

One of the most obvious improvements of this "IGI2" system is its significantly increased rendering performance than originally published in 2001 [Wald02b]. First of all, the ray tracer has been significantly improved since that publication [Wald03e] (also cf. Section 7). By design, the IGI algorithm spends most of its time in tracing rays. As such, these speed improvements have directly translated to higher global illumination performance. Apart from this "free mileage", the IGI subsystem itself has also been completely rewritten from scratch to remove the bottlenecks and speed deficiencies of the original implementation (see previous section), and to even better exploit the possibly achievable performance of the underlying ray tracing system.

As already outlined in Chapter 7, optimal ray tracing performance can only be achieved if the fast SSE packet tracing code is being used, and if the cost for shading is kept as small as possible. As all the rays shot in IGI consist of highly coherent shadow rays to point light sources, packets of coherent rays can be easily maintained most of the time, thereby allowing to trace these rays with the fast SIMD code. Furthermore (in contrast to a typical recursive ray tracer), all rays undergo exactly the same shading computations, and as such can also be computed with fast SSE code, without the need and overhead for breaking the SSE packets up into individual rays. As a result, almost all computations in the new IGI system work in a streaming manner, operate on packets of rays, and are implemented with fast SSE code. Though shading is still costly even in SSE (cf. Table 7.6 in Section 7), the *overall* impact of shading is rather small, as the shading operations have to be performed only once for each primary ray, and can then be amortized over all shadow rays.

At peak performance, the new system achieves rates of up to 8 million rays per second on each AthlonMP 1800+ CPU[5].

These speed improvements – together with the recent improvements in ray tracing performance mentioned in Chapter 7.4 – now allow the new system to significantly outperform the old system, thereby achieving much higher image quality and higher frame rates. For example, the animation shown in Figure 13.8 can be rendered with up to 25 frames per second.

13.3.2 Removing the Server Bottleneck by Filtering on the Clients

With the much higher performance of the clients the scalability bottleneck on the server could no longer be tolerated. Therefore, the filtering computations have also been moved to the clients, where they are also computed with fast SIMD code. As discussed in Section 13.2.1, this implies that the new system is no longer able to support caustics. Obviously, the same caustic photon mapping technique could still be used as before. However, as each client now has to compute *all* the caustic photon sets, even less photons are affordable during each frame. As the caustic quality was already problematic in the

[5] Note that 8 million rays per second on an AthlonMP 1800+ CPU (running at 1.5GHz) is significantly more than the performance data given in Section 7, where this performance could only be achieved on a significantly faster Pentium-IV 2.5GHz CPU. This discrepancy is due to the fast that the IGI system shoots mainly shadow rays, which can be traced much faster than the primary rays measured in Section 7.

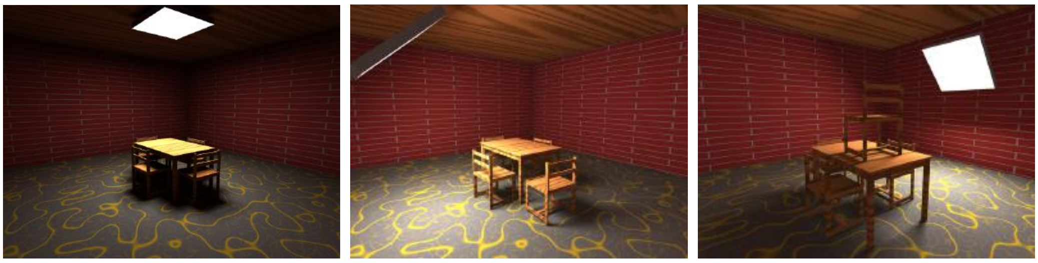


Figure 13.8: An IGI2 animation sequence in the ERW6 scene. Note how the most important effects – sharp as well as smooth shadows, indirect diffuse illumination, color bleeding, and procedural shading – are correctly computed in each frame. As the ray tracer itself supports dynamic scenes (see Chapter 9), the scene can be dynamically reconfigured, e.g. by moving the light, or by putting the chair on the table. The depicted animation sequence can be rendered at this quality *with up to 10 frames per second, and – at slightly reduced quality – with up to 40 frames per second*

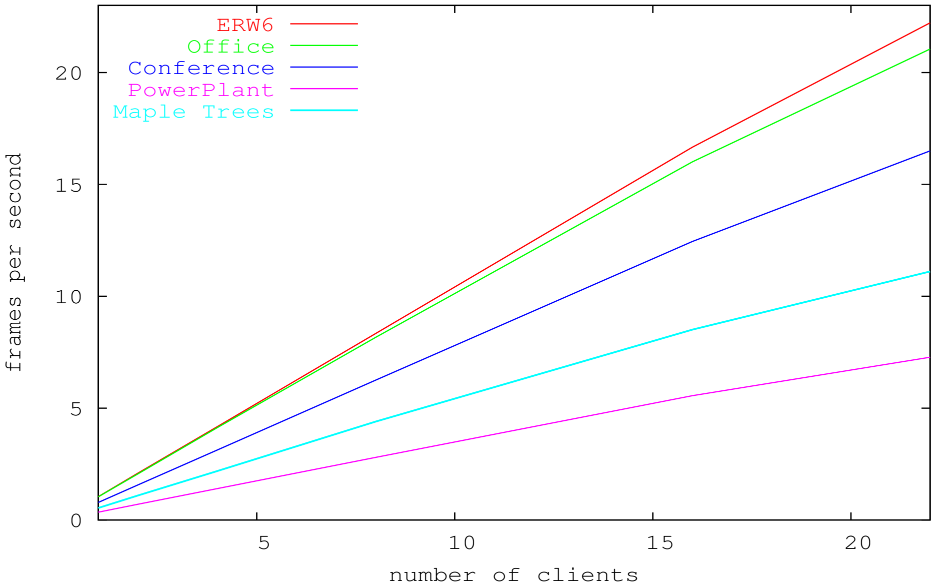


Figure 13.9: Scalability of the new IGI system with the number of rendering clients: Performance is essentially linear up to 24 PCs/48 CPUs. This applies to scenes ranging from several hundred triangles (ERW6) up to the power plant with 50 million triangles (four instances). Also note how the new system scales in frame rate well beyond the original system, which was limited to at most 5 frames per second.

original version, this does not appear to be a reasonable option. It would still be possible to add caustics incrementally during times of no user interaction.

As a compensation for no longer being able to support caustics, filtering on the clients completely removes any computational load from the server. This in turn removes the server bottleneck, allows to server to concentrate on load balancing, and lets the system scale easily to more than 48 CPUs, and even to frame rates beyond 20 frames per second (see Figure 13.9). Currently, the systems performance is mainly limited by the bandwidth of GigaBit Ethernet, which in our somewhat outdated hardware configuration cannot transfer more than roughly 25–30 frames per second at 640 × 480 pixels (see Chapter 8).

13.3.3 Programmable Shading

Apart from the significantly higher performance and scalablity, the new IGI2 system now supports complex procedural shaders (like the "wood", "marble", and "brickbump" shaders that can be seen in Figure 13.10) also in the global illumination computation.

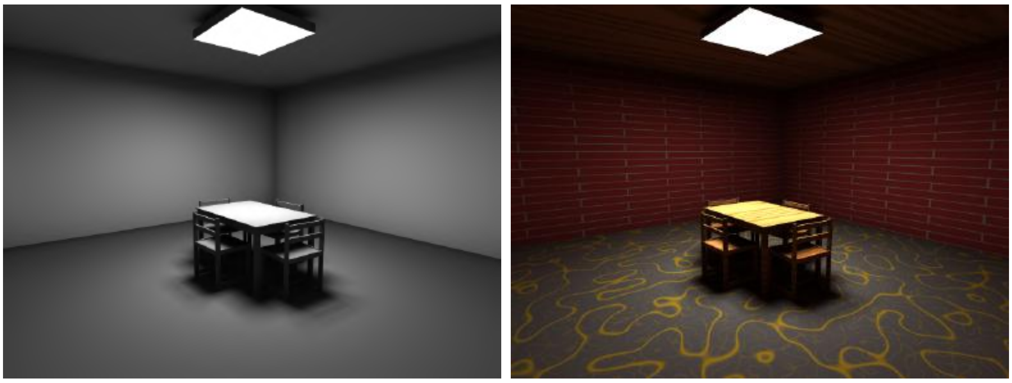


Figure 13.10: Freely programmable procedural shading in a globally illuminated scene. The standard "ERW6" test scene (left) and after applying several procedural shaders (marble, wood, and brickbump). Even with shaders that make extensive use of procedural noise the performance only drops to 3.7 fps compared to 4.5 fps with a purely diffuse BRDF.

In order to support programmable shading, the new system splits the shading part into a *material shader* and a *BRDF evaluation.* The material shader is freely programmable by the user, and is expected to return a BRDF description (e.g. a phong model) for each primary ray, in which the BRDF parameters can vary from ray to ray[6]. When computing the contribution from the VPLs, this BRDF is then evaluated for each shadow ray. During evaluation of the BRDF, all shadow rays undergo the same computations (except for different parameters). Thus, the BRDF evaluations can easily and efficiently be implemented for packets of rays using SSE (see Section 7). Following the same pattern as described above, four BRDFs are evaluated in parallel, achieving high SSE utilization and very efficient BRDF evaluations.

For convenience, the user-programmable material shaders operate on *single rays*, which consequently requires costly data-reorganization, and does not allow for SSE implementation. As a result, calling the user-programmable material shader can be quite costly. In fact, it can be several times as costly as shooting the primary ray itself (see Section 7).

However, as already discussed in the previous section, this expensive material shader is called only once for each primary, reflection, or refraction ray. The BRDF returned from this material shader then can be evaluated very efficiently for all shadow rays. Using this framework, the cost for programmable shading can efficiently be amortized over all shadow rays, and has only a minor impact for a reasonable amount of VPLs (see Table 13.1).

Num VPLs	0	1	2	4	8	16	32
Diffuse	1	0.59	0.50	0.38	0.26	0.16	0.09
Wood	0.27	0.24	0.22	0.20	0.16	0.11	0.07
Overhead	270%	146%	127%	90%	63%	45%	29%

Table 13.1: Impact of programmable shading on IGI2 performance in the ERW6 scene as shown in Figure 13.10 scene). Procedural shading ("Wood" in this example) can be quite costly, and in this example costs almost 4 times as much as a diffuse shader if no shadow rays are cast at all. Using the proposed "material shader" model, this cost can be amortized over many shadow rays, and even drops below 30% for 32 VPLs. Note that for more complex scenes, the overall overhead would be significantly less due to a better ratio of ray tracing cost to shader overhead.

13.3.4 Fast Anti-Aliasing by Interleaved Supersampling

The original system provided high-quality anti-aliasing using progressive over-sampling in static situations but suffered from artifacts during interaction. This was caused by the low image resolution and the fact that only

[6] In a "Wood" material, for example, the diffuse reflectivity of the phong model would be procedurally determined per ray

a single primary ray was used per pixel.

Efficient anti-aliasing is still an unsolved problem in ray tracing as the rendering time increases linearly with the number of rays traced. Anti-aliasing by brute-force super-sampling in each pixel is thus quite costly, in particular for an interactive context. On the other hand, methods like adaptive supersampling are problematic due to possible artifacts and the increased latency of refinement queries in a distributed setup.

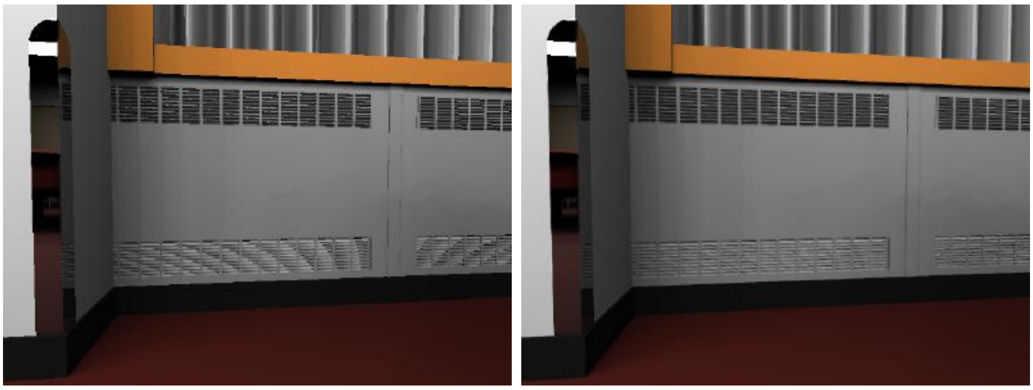


Figure 13.11: Efficient anti-aliasing. Left: A single primary ray per pixel, exhibiting strong aliasing artifacts. Right: 4 primary rays per pixel, resulting in an aliasing-reduced image. Using interleaved anti-aliasing, the performance of the right image with four-fold supersampling is only slightly lower than the left image, running at 3.2fps compared to 4.0 fps. As both images use the same total number of shadow rays per pixel, the quality of the lighting simulation is virtually indistinguishable.

The IGI2 system [Benthin03, Wald03e] can also perform supersampling with little performance impact using a similar interleaving approach as for sampling the VPLs. Instead of connecting each primary ray to all M VPLs in the current set, the VPLs are grouped into N different subsets with roughly M/N VPLs each. We then use N primary rays per pixel for anti-aliasing, each ray computing illumination only with its own subset of VPLs.

For typical parameters (i.e. $M >= 16$ and $N = 4$ or $N = 8$), the overhead of the $N - 1$ additional rays is usually in the order of 20% to 30%, which is well justified by the increase in image quality that is due to the 4- to 8-fold supersampling (see Figure 13.11).

13.3.5 Summary and Conclusions

In summary, the Instant Global Illumination Method is an efficient framework that combines several base ingredients – instant radiosity, fast ray trac-

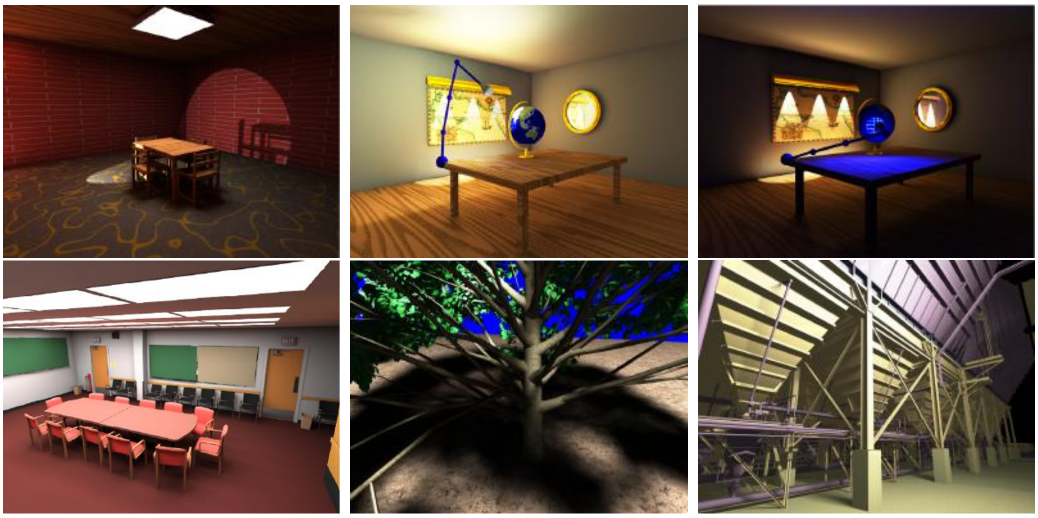


Figure 13.12: Screenshots from the new IGI2 system: a.) The ERW6 scene with programmable shaders (here with a spot light source to demonstrate that IGI is not limited to diffuse area light sources), b.) and c.) IGI2 in an animated VRML scene, in which color bleeding lets the entire scene get blueish once the light source approaches the blue globe, d.) Conference scene with 280,000 triangles and 220 light sources , e.) IGI in some multi-million-triangle maple trees with detailed shadows from the branches and leaves, and f.) Global Illumination in the 12.5 million triangle "power plant" scene. All scenes run interactively at several frames per second at 640*x*480*, with – depending on the actual quality settings – up to 25 fps in the simpler scenes (see Figure 13.9). On a different hardware configuration, even more than 40 fps could be achieved.*

ing, interleaved sampling, and discontinuity buffering – in order to interactively compute global illumination. While not all illumination effects are accounted for (most importantly: no caustics) the most important features of global illumination are correctly simulated: Direct lighting, soft as well as hard shadows, indirect illumination effects, color bleeding, reflections, and refractions (see Figure 13.12).

Due to a better implementation, a complete restructuring into a streaming framework, thorough use of SSE code, and a more scalable parallelization, the newest implementation of this method achieves significantly higher performance than the Instant Global Illumination system as originally proposed in 2001/2002 [Wald02b]. Apart from a very high performance on each individual CPU, the new system scales easily to more than 48 CPUs, and up to frame rates of 25 frames per second at video resolution. Note that this upper limit of 25 frames per second is only due to our (somewhat outdated) network infrastructure. On a newer network infrastructure (still GigaBit ethernet, but newer switch and network cards) the system has already reached frame rates beyong 40 frames per second.

Finally, the new system also features higher rendering quality: Apart from being able to use more samples at the same frame rate, the new system also supports programmable material shaders and an efficient means of supersampling.

Interestingly enough, efficient supersampling, a thorough streaming design with complete SSE implementation, as well an efficient means of shading packets of rays (using the proposed material shader model) allow the IGI2 system to typically achieve *both* higher image quality and higher performance *with* global illumination, than most scenes usually achieve with "standard" shaders computing only local illumination. This once again stresses the need for finding efficient means for shading packets of rays also for non-global illumination rendering modes.

Chapter 14

Efficient Instant Global Illumination in Highly Occluded Scenes using “Light Source Tracking”

As described in the previous chapter, the Instant Global Illumination method – running on the RTRT/OpenRT realtime ray tracing system – now allows for computing global illumination solutions at interactive rates. This availability of interactive global illumination now provides a new and important tool for many practical applications, e.g. in product design, virtual prototyping, and architecture, in which the accurate simulation of indirect effects can have a decisive influence on the evaluation process.

However, the just mentioned disciplines often require the realistic visualization of entire planes, ships, complete buildings, or construction sites. Such scenes often consist of many individual rooms and contain millions of triangles and hundreds to thousands of light sources (see Figure 14.1), that cannot easily be handled by todays algorithms.

Due to a high degree of occlusion, most of the different rooms in such scenes are typically influenced by only a small fraction of all light sources. For example, a room on the 7th floor of the building in Figure 14.1 will hardly be illuminated by a light bulb in the basement. The combination of high occlusions with large numbers of light sources are a challenge for most off-line and interactive algorithms because many samples are computed without a significant contribution to the final image. If the few important light sources could be identified efficiently, rendering would be performed much more efficiently. While it has been common to optimize the rendering

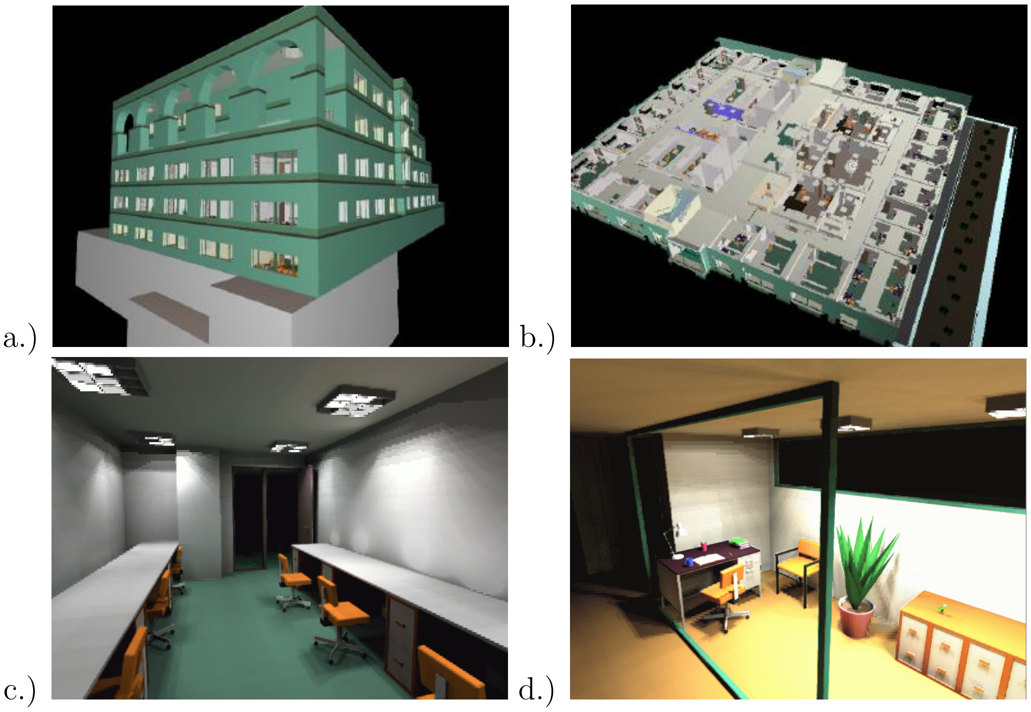


Figure 14.1: "Soda Hall", a 7 storey fully furnished building containing 2.2 million triangles and 23,256 light sources. a.) The entire building, simply shaded, b.) Overview of a single floor c.) and d.) Inside, with full global illumination. Note that almost all of the rooms inside this building are modeled in high detail. With our proposed technique, the right images run at 2-3 frames per second on 22 CPUs allowing to interactively explore the entire building with full global illumination.

process by manually disabling irrelevant light sources for off-line processing, this is not an option for interactive applications where the set of relevant light sources can change from frame to frame.

While it is clear that we are mostly interested in solving that problem for our interactive global illumination technique, this problem is actually not limited to online applications. In this chapter, we present a method that is applicable in both an offline and in an online context. This method exploits the special characteristics of such scenes by automatically determining the relevant light sources. We estimate the visual importance of each light source for the final image and use this estimate for efficiently sampling only the relevant and visible light sources. While we do not want to compromise on quality for the offline rendering setting, we tolerate a reasonable amount of artifacts in the interactive setting as long as the general impression remains correct.

We start with a detailed analysis of the problems that arise when calculating global illumination in complex and highly occluded scenes in Section 14.1, together with a short discussion of previous work in Section 14.2. In Section 14.3 we present the basic idea of our method before discussing how it can be applied to instant global illumination in Section 14.4. Finally, we conclude and discuss future work in Section 14.6.

14.1 Global Illumination in Realistically Complex Scenes

Realistic scenes such as those listed above share a number of characteristics: Massive geometric complexity of up to millions of triangles, hundreds to thousands of light sources, and high occlusion between different parts of the scene.

14.1.1 Geometric Complexity

Realistically complex scenes often consist of millions of triangles in order to accurately model the geometric detail (e.g. detailed pencils in the "Soda Hall" scene). Such geometric complexity has always been problematic for radiosity-style algorithms that have to store illumination information with the geometry of the scene. While clustering [Smits94] and the use of view importance [Aupperle93] do help in such scenes, these algorithms must still sample and process the entire scene geometry.

For all ray-based rendering algorithms – e.g. (bidirectional) path tracing [Kajiya86, Veach94, Lafortune93], instant radiosity [Wald02b, Keller97], or photon mapping [Jensen96, Jensen01] – the pure number of triangles is theoretically less of an issue, as such algorithms are sublinear in the number of triangles [Wald01a, Parker99b, Havran01]. This weak dependence on scene complexity allows for efficiently rendering even scenes with millions of triangles [Wald01c, Pharr97] (see Figure 14.2b) and at least in theory makes ray- and path-based algorithms mostly independent of geometric complexity. In practice, however, this holds true only if a high coherence of the rays can be maintained: As soon as the rays start to randomly sample the entire scene – as done by virtually all of todays global illumination algorithms – even moderately complex scenes won't fit into processor caches, resulting in a dramatic drop of performance. *Really* complex scenes may not even fit into main memory, leading to disk thrashing if sampled incoherently.

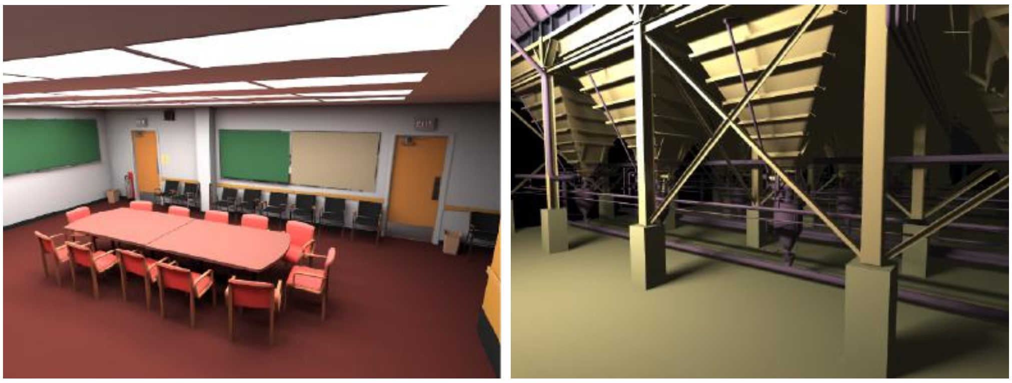


Figure 14.2: Neither geometric complexity, nor a moderate number of light sources is a problem for e.g. the instant global illumination method. Left: The "conference" model with 202 light sources, interactively illuminated at 5-10 fps. Right: instant global illumination in the 12.5 million triangle "Powerplant" at 2-3 fps. Though both problems individually can be handled very well, the combination of many triangles, many lights, and high occlusion is currently infeasible with such systems. Note that the power plant contains only a single, manually placed light source, and thus shows little occlusion and high coherence.

14.1.2 Many Light Sources

The second important cost factor is the large number of light sources. In reality, any room in a building usually contains several different light sources, resulting in hundreds to thousands of light sources for a complete building. Many algorithms require to consider and sample all light sources, (e.g. by starting paths or particles from each of them), thus requiring a prohibitively large number of rays to be shot for a single image. If these lights all contribute roughly equally to each point, subsampling the light sources works nicely (see Figure 14.2a), even if the degree of undersampling is quite severe. However, subsampling no longer works in our setting of highly occluded models, as the variation of the different samples then is too high.

Even worse, however, is the fact that the light sources are usually scattered all over the model such that it becomes hard to avoid sampling the entire geometry (see Figure 14.3). This is especially true for those kinds of algorithms that have to start rays, paths, or particles *from the light sources.* Unfortunately, this applies to almost all of today's global illumination algorithms.

14.1.3 High Occlusion

Finally, the above mentioned scenes are usually highly occluded, and only few light sources will actually contribute to each given point. For many algorithms, this results in wasted computations. For example, algorithms that work by tracing particles from the light sources will waste most of their time tracing particles into distant parts of the scene where they will not at all contribute to the current image.

Similarly, all algorithms that require to find valid connections between a surface point and a light (e.g. path tracing, bidirectional path tracing, or any algorithm computing direct illumination separately) have to generate lots of costly samples just in order to find the few unoccluded connection. Generating enough unoccluded paths to light sources to achieve a sufficient image quality requires to shoot far too many rays for reasonable rendering performance.

Conclusions

While each of the problematic characteristics – high geometric complexity, large number of light sources, and high occlusion – can be handled relatively well by at least one of the available techniques, their combination is hard to handle for any of these algorithms. An algorithm that would be able to handle scenes with the afore-mentioned properties would have to fulfill several requirements: First, it should be based on ray tracing to efficiently handle the geometric complexity. Second, it must generate coherent rays that touch only the relevant parts of the model. Especially, it *must not* start rays from light sources or at least limit these rays to actually visible lights in order to reduce the working set of the algorithms for very complex scenes. Finally, it should concentrate its computations to only those light sources that actually contribute to the image.

14.2 Previous Work

Each of the afore-mentioned problems – geometric complexity, many light sources, and occlusion – has received significant previous research, so that we can only review the most important contributions. For handling complex models all different kinds of ray tracing have proven to be very effective [Parker99b, Wald01a, Wald01c], as long as coherence of the rays is high and the working set of the rendering algorithm remains relatively small. Pharr [Pharr97] demonstrated an out-of-core system that can handle millions of triangles even for simulating global illumination. However, this system is not easily applicable to an interactive setting and did not specifically target highly occluded scenes.

For radiosity-style algorithms, hierarchical radiosity [Hanrahan91] and clustering [Smits94] have been introduced to improve the performance of radiosity methods for complex scenes. View importance [Aupperle93] has been used to concentrate computations to parts of the scene relevant to the image. However, this approach still iterates through the entire model in order to check for the propagation of importance. All these algorithms are at least linear in scene complexity and so far have shown to be difficult to adapt to interactive use.

For ray-based systems, Shirley et al. [Shirley96] have proposed several importance sampling technique for the efficient handling of many luminaires. Though these techniques are a basic requirement for *any* high-quality and high-performance global illumination algorithm, they do not account for visibility. Thus they cannot solve our problems with highly occluded scenes where the importance of each light source is much more determined by its visibility than by its extent and orientation.

Ward et al.[Ward91] introduced an algorithm to select the most relevant light sources during rendering of a single frame while the contribution of other light sources was estimated without additional visibility tests. Our approach uses the same idea but extends it to deal with complex scenes with high occlusion in an interactive context.

To account for occlusion, both Jensen et al. [Jensen95] and Keller et al. [Keller00] have proposed to use a preprocessing step for approximating the direct illumination using a kind of photon map. During rendering, this information could be used to estimate the importance of a light source. This allows for efficient importance sampling by concentrating samples to the actually contributing light sources. However, their methods require to store a photon map, and is therefore not easily applicable to an interactive setting[1]. Furthermore, the preprocessing step requires to first emit photons from *all* scene lights, which is not affordable for highly complex environments.

14.3 Efficient Importance Sampling in Complex and Highly Occluded Environments

Our approach is mainly targeted towards realistically complex scenes that combine high geometric complexity, many lights, and high occlusion. Sev-

[1] Also see Chapter 12.1 for a discussion about the suitability of photon mapping for an interactive distributed setting.

eral examples of such scenes – the same scenes we will use in our experiments later – can be seen in Figures 14.5 and 14.8: Both "ERW10" and "Ellipse" have only moderate complexity, but already contain many lights and high occlusion. Additionally, "Soda Hall" is a more realistical model of an existing building at Berkeley University, featuring 2.2 million triangles with highly detailed geometry, and 23,256 light sources scattered over seven storeys containing dozens of fully furnished rooms each.

As discussed above, high geometric complexity can be handled well by ray tracing systems, as long as the costly processing of mostly occluded lights, and random sampling of the whole model is avoided[2]. Achieving high rendering performance in such scenes requires us to efficiently determine the non-contributing lights without sampling the entire scene.

As a solution, we have chosen to use a two-pass approach: In a first step, we use a crude estimation step to roughly determine the importance of the different light sources for the image, and thus to identify the important light sources. For this step we use a modified path tracer as motivated below (also see Figure 14.3).

In the second step, this information is used for improving the rendering quality with importance sampling of the light sources. This not only concentrates samples on most likely unoccluded lights, but also avoids sampling those parts of the scene that are occluded.

Step 1: Crude Path Tracing to Estimate Respective Light Source Contributions

For the estimation step, we have chosen to use an eye path tracer with relatively low sampling rate. Though a path tracer may at first seem unsuited for this task (being well known for its noisy images), there are several good reasons for our choice: First, a path tracer is trivially parallelizable, which is an important feature to be applied in an interactive setting. It also easily enables to trade quality for speed by just changing the number of paths used per pixel.

Second, a path tracer only builds on ray tracing, and does not need any additional data structures. Thus, geometric complexity is not a problem as long as the rays remain mostly coherent. This coherence is, however, not a problem either. While path tracers are known for their *lack* of coherence (because the randomly chosen paths sample space incoherently), this is no

[2] Also see Table 7.6, which compares two of the test scenes used in the following experiments: Though Soda Hall is more than an order of magnitude more complex than ERW10, it is less than half as slow.

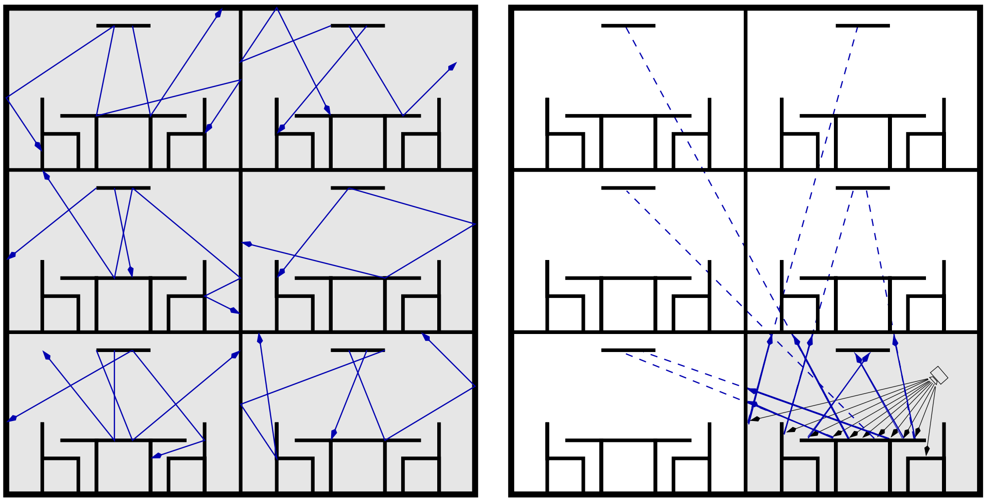


Figure 14.3: Estimating light importances with path tracing. This intentionally simplified example assumes a scene consisting of six simple, non-connected rooms. Left: If rays (or particles) would be started from all light sources (e.g. for bidirectional path tracing or photon mapping), the entire scene geometry would eventually be sampled. Right: Though a path tracer also evntually samples all light sources, it only samples visible geomtry. Shadow rays to occluded light sources (dashed blue) get blocked at the first (still visible) occluder (blue lines). After we have determined the visible light sources, we can also start particles from these light sources, e.g. for generating the VPLs needed for Instant Global Illumination.

longer true on a coarser scale: As a path tracer is purely view-importance driven, it will only sample geometry that will likely contribute to the image, and never even touches unimportant regions at all.

Of course, the path tracer eventually has to sample all lights with shadow rays, as it can not know which lights are unimportant. However, these shadow rays, if shot *from* the path *towards* the light source, will either reach the light source (in which case the light source is important) or are blocked in the visually important part of the scene in which it was started (see Figure 14.3). Thus, the actual footprint of data touched by the path tracer directly corresponds to those parts of the model that are actually important.

One potential problem with using a path tracer is that pure path tracing usually generates very noisy results, and requires lots of samples for reliable results. Even though this is undoubtedly true for the whole image, we are not interested in the actual pixel values, but only in the *absolute* importance of light sources for the entire image[3]. Then, even when shooting only a single path per pixel, rather high sample rates and reliable results can be produced: For example, if we render the ERW10 scene which contains 100 light sources at a resolution of 640 × 480 with only one sample per pixel, then more than 3,000 samples will be used per light source. Thus, even if a path-traced image with as few samples may be hardly recognizable at all, reasonably reliable estimates can still be expected (see Figure 14.4).

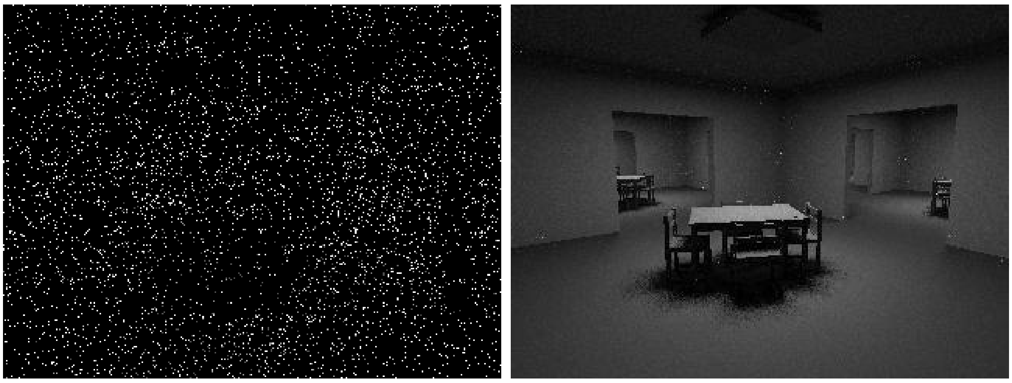


Figure 14.4: Quality of the estimate in the ERW10 scene. Left: Estimate as results from a path tracer using a single sample per pixel. Right: The same image rendered with 1024 samples per pixel. Though the estimate image is hardly recognizable, the contributions of the light sources – over the whole image – are estimated correctly up to a few percent.

Of course, some noise remains in the form of variance in the estimate. This, however is not a problem as the estimated importance will never be visualized directly but will only be used for importance sampling. A strongly varying estimate may, however, become problematic if used in an interactive context with low sampling rates for light sources, where it may lead to temporal artifacts like flickering. This problem will be addressed in more detail in Section 14.4.

Step 2: Constructing an Importance Sampling PDF

The method used for constructing the PDF from the information gathered during path tracing can greatly influence the rendering process. To obtain an unbiased estimate the PDF used for importance sampling should never be

[3] For instant global illumination, each light source always contributes to the whole image. For other algorithms, it might make sense to also estimate the change of importance with respect to different regions in the image.

zero for any actually contributing light. Though this could be easily guaranteed by just assigning a certain minimum probability for each light source, this would result in many light sources being sampled and their surrounding geometry being touched. At the expense of being slightly biased, this can be avoided by thresholding the importance of light sources. This will effectively 'turn off' light sources with a very small contribution. Even though being biased, this thresholding in practice is hardly noticeable if the threshold is chosen small enough.

Step 3: Importance Sampling during Rendering

After computing the PDF we can use it during rendering quite easily. For most algorithms, the only modification to the rendering phase is to simply replace the existing PDF used for sampling the light source.

While we want to eventually use our technique for interactive applications, it also works in an offline context: As a proof of concept we have first applied it to a simple bidirectional path tracer [Veach94, Lafortune93]. Integrating our method into the original bidirectional path tracer (BDPT) was trivial, as only the PDF for sampling the start point for the light ray had to be modified. For the estimation step, we usually use only a single path per pixel. As this is rather small compared to the $16 - 64$ bidirectional paths during rendering, the cost of the estimate does not play a major role for the offline rendering application. Of course, all comparisons between new and original version do include the estimation overhead.

Using our method allows for efficiently concentrating samples to important light sources. For ERW10, this translates to having to use only 8 instead of 100 lights after turning off lights with very small contributions. For Soda Hall the benefit is even bigger, reducing the number of lights in some views from 23,256 to only 66. Using the same number of paths, this results in significantly better quality, as can be seen in Figure 14.5. This can also be measured in terms of RMS error to a master image, where – after the same rendering time – the new method produces significantly less error (see Figure 14.6).

Note that the path tracer used for these simple proof-of-concept experiments was certainly not a highly sophisticated, production-style renderer. Such renderers would usually employ several different techniques to improve both rendering performance and image quality (such as e.g. clever sampling strategies, deterministic sampling, or others). However, any benefits from such techniques should apply equally well for both the original and the modified implementation. As such, we are confident that our method should also translate to more sophisticated renderers.

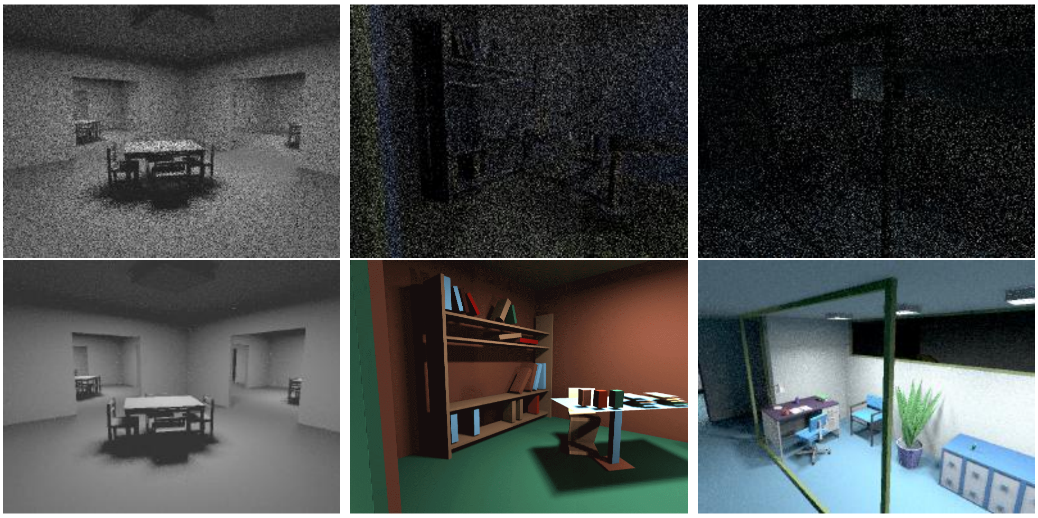


Figure 14.5: Quality comparison (rendered with a bidirectional path tracer) at same number of computed paths in the ERW10, Ellipse, and Soda Hall Scenes. Top: Original method. Bottom: Using the estimated importance. This example clearly shows the impact of our method on the overall image quality. The impact of our method increases with the degree of occlusion in the scene.

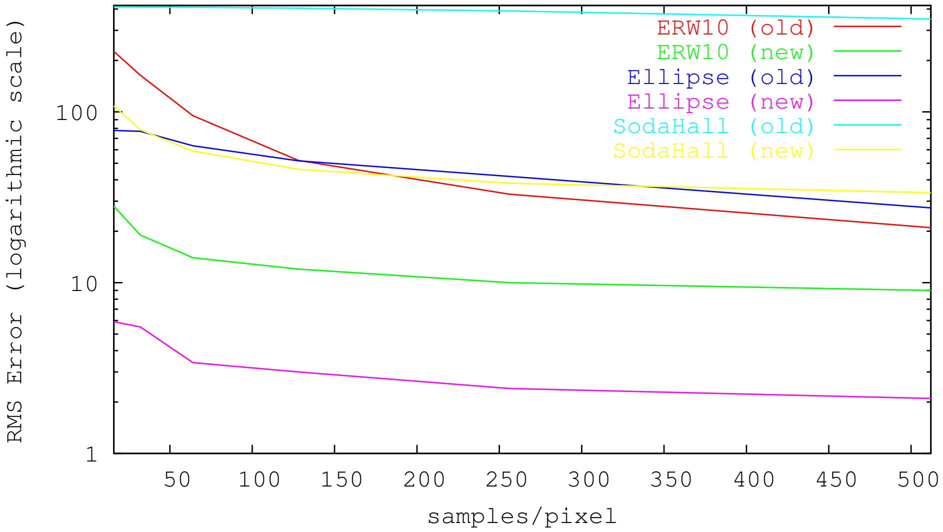


Figure 14.6: Convergence Speed: RMS error to a master image, over rendering time, for different scenes. As can be seen, the new method clearly outperforms the old method.

14.4 Application to Instant Global Illumination

As just seen in Figures 14.5 and 14.6, importance sampling using our estimation step can efficiently concentrate the samples to non-occluded light sources, and can thereby increase overall image quality also for offline algorithms. Obviously however our main interest lies in applying this algorithm to Instant Global Illumination, which currently cannot handle such scenes.

As already described in Chapter 13, Instant Global Illumination [Wald02b] (IGI) in its core builds on a variant of instant radiosity [Keller97], in combination with interleaved sampling [Keller01], and a filtering step: The lighting in a scene is approximated by a set of virtual point light sources (VPLs) that are created by tracing a few "big" photons or particles from the light sources. These VPLs are then used to illuminate the scene just as with normal point light sources.

Due to the underlying fast ray tracing engine, IGI can efficiently handle even complex scenes of several million triangles. However, its efficiency depends to a large degree on the occlusion of a scene: As the performance is directly proportional to the number of VPLs used, only a small number of VPLs can be handled efficiently (in the order of 40 to 100).

In highly occluded scenes, most of these few VPLs will likely be located in parts of the scene where they do not contribute at all to the current image to be rendered. In that case, many more VPLs than interactively affordable would have to be used to achieve a reasonably good coverage of the current view and obtain a good image quality. For example, consider rendering the ERW10 scene compared to rendering a single of its rooms, where one would have to use 100 times as many VPLs for the whole model in order to get the same quality as one would get for the single room.

Using the previously described importance sampling scheme, it should be possible to concentrate the few precious VPLs to the actually important rooms, and thus be able to render such scenes with good quality at interactive rates. However, integrating the new method into an interactive framework requires to solve new challenges due to the combination of an interactive setting with extremely low sample rates.

In principle, three problems have to be solved:

Distributed Implementation: The estimation and importance sampling steps have to be integrated into the distributed computing framework using a cluster of PCs that IGI runs on. Though future hardware platforms may not require this distribution framework, today this is unavoidable in order to achieve interactive performance.

Suitable construction of the PDF: Special care has to be taken when constructing the PDF in order to cope with the extremely small sampling rates. In particular, we have to make sure that light sources with small importance receive some samples at all. If, for example, one light source contributes 80 percent to the illumination in an image, giving it 16 out of a total of 20 VPLs would only leave 4 VPLs to represent all other light sources.

Handling of temporal artifacts: Most importantly we have to consider temporal artifacts of our method. This is especially important due to the low sample rates that lead to slight errors that are hardly noticeable in a still image but which can lead to strong flickering between successive frames.

14.4.1 Distributed Implementation

The distribution framework used in the IGI system uses a tile-based dynamic load balancing scheme, which lets different clients compute different image tiles [Wald01c, Wald02b] (see Chapter 8). In order to avoid seeing the tile borders in the final image, we have to make sure that all clients compute *exactly* the same image. In particular this means that the clients have to use exactly the VPLs for computing their respective image tiles. For the original system, this was simple to achieve by synchronizing the seed values for the random number generators on all clients. Using the same random numbers, all clients would generate exactly the same VPLs, and thus would compute the same image.

In our new version, however, we also have to make sure that all clients use exactly the same PDF. Using a different PDF – even when using the same random numbers to sample it – would result in different VPLs, which in turn would be clearly visible as a tiling pattern on the image plane where each tile is illuminated differently.

In order to synchronize the PDFs, there are basically two choices: First, one could let all clients perform the full estimation step on the whole frame, and again use synchronized random numbers. Even though this would work, this approach would not scale: Each client would have to shoot several hundred thousand rays just for the estimation step, which already exceeds the budget of rays it can shoot for a single frame[4]. As such, this approach is not applicable in an interactive setting.

[4] A full estimation step for the entire image can easily cost more than a second for reasonable quality parameters. If this cannot be amortized over different PCs, no interactive performance could be expected.

In the second approach, each client performs the estimation only for its current tile. This perfectly fits the load balancing scheme as clients computing more tiles also perform more estimation computations. On the other side, it results in different PDFs on each client and requires a global synchronization. Due to the long latency of the underlying commodity network the clients cannot wait for the server to have received and synchronized the estimate from all clients for the current frame. On the other hand, if we tolerate one frame of latency for the estimate to be used, we have a simple and efficient solution. Each client performs the estimation for its current tile and sends the results to the server in compressed form. The server combines the different estimates and broadcasts the global estimate by attaching it to the first tile sent to each client for the next frame. This combination of the estimates is extremely cheap and thus does not create a bottleneck at the server.

The latter approach obviously works only if the PDF for the next frame is not too different from the current frame. This is usually not a problem for walkthrough applications, but can result in an slight delay, e.g. when newly turning on a previously switched-off light source, or when newly entering a previously completely occluded room. In practice, however, the visual artifacts ususally remain small and tolerable.

14.4.2 Exploiting Temporal Coherence

For a typical walkthrough, successive frames usually do not differ too much. This temporal coherence can be exploited in several ways: First, we can smooth the importance estimate by combining the estimates from several successive frames. We currently do this by combining the old PDF with the current estimate using the weighed sum $P_{new} = \alpha P_{est} + (1-\alpha)P_{old}$. This allows for using fewer estimate samples per frame and thus to lower the estimation overhead.

Similarly, we can use the PDF from the last frame to also use importance sampling in the estimation process. This increases the reliability of the estimate in the interactive setting. However, we have to make sure that for the estimation step, each light source gets a minimum probability for being sampled. Otherwise, a light source that would once receive a zero PDF would never be sampled nor estimated again and would forever remain invisible. Though assigning a minimum sampling probability to each light source eventually samples all lights during estimation, the shadow rays are shot from the eye path towards the sampled sources, and thus will *not* touch far-away geometry around occluded light sources (see Figure 14.3).

Finally, another way of exploiting temporal coherence is to reuse eye rays that have to be traced anyway: As the estimate computed in the current tile will only be used in the next frame we can save time by not tracing separate primary rays for estimation *before* the rendering pass. Instead we reuse the primary rays already traced for rendering the current frame, thereby again reducing the estimation cost.

14.4.3 Avoiding Temporal Noise

The main problem to cope with in our approach is temporal noise, which may become visible as flickering of illumination. Even though both instant radiosity and our importance sampling step are unbiased in theory, the small number of samples (VPLs) affordable for interactive frame rates lead to a certain remaining error in the image. As this error is usually smoothly distributed over the entire image, it is often not noticeable in a still image. In an interactive setting however, two successive frames that are rendered with different VPLs may have their error in different parts of the image, resulting in visible flickering.

For the original system, this flickering could be controlled by using the same random numbers for successive frames, generating exactly the same VPLs for both frames. Using view-driven importance sampling, this is no longer possible, as any kind of user interaction – moving, turning, or interacting with a scene – will change the view importance for the next frame. As this view importance influences where the VPLs will be placed, any interaction will lead to “jumping” of the VPLs in between frames.

In order to minimize this temporal flickering, we use the temporal smoothing of the PDF mentioned above to minimize variations in the PDF. This however can also lead to an increased latency until a drastic change of a light source’s importance is taken into account.

Even more importantly, we make sure that the VPLs will be computed as coherent to the previous frame as possible. For example, the usual way of generating the VPLs would be to process all VPLs one after another by first sampling a light source (using our PDF), and then randomly placing a VPL on it. Then, however, it may easily happen that due to a change in the probability of another light source, a VPL will ’leave’ the light source it was placed on in the last frame. Now, even if the light source may receive another VPL, the new VPL will be generated by different random numbers, and thus be placed differently on the source. In practice, this leads to most VPLs jumping from frame to frame even with only small changes in the PDF (see Figure 14.7.

In order to avoid this effect we have reversed the process. We first compute the number of VPLs that start at each light source using an error

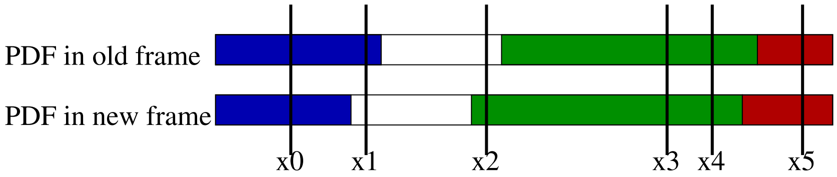


Figure 14.7: Small changes in the PDF leading to jumping in the VPLs: As the same random number is usually used for both sampling the light source (i.e. either the red, blue, green, or white bar in this example) as well as for determining the position on that light, even small changes to the PDF can have a significant impact. In this example, even if the same random numbers have been used, and even if the number of samples on the white light source remains the same, the single VPL on the white light source "jumps" from the right to the left end of that light source, which propuces disturbing flickering of the illumination.

diffusion process to make up for rounding errors. Then for each light source that has received some VPLs, we start generating the VPLs with a random seed that is unique to the light source. Thus, the VPLs on a light source will always be placed in exactly the same way no matter how the PDFs of other light sources change. Also, if the number of VPLs on a light sources changes from n to m the first $\min(n, m)$ VPLs will remain the same, leading to drastically reduced flickering. Instead of processing all interleaving patterns independently, we perform this VPL distribution step for all lights of all interleaving patterns in order to maximize the average number of VPLs per light source.

However, if there remain many more active lights than the number of VPL paths that we compute for IGI, this trick will no longer work. In this case we can no longer expect VPLs to stay on any particular light sources if the PDFs change. This could probably be solved by clustering nearby lights [Meneveaux03] and distributing the VPLs according to these clusters in the same way as discussed above. So far however this approach has not been investigated.

14.5 Results and Discussion

Due to the interactive context it is hard to quantify the results of the new method in tables or present them as still images on paper. The full impact of the improvements only become obvious when experienced interactively.

14.5.1 Temporal Artifacts

Temporal artifacts become most visible in the form of flickering and are mainly caused by the extremely small sampling rates used in an interactive context. As discussed above flickering is caused by some of the VPLs changing position between frames. The methods discussed in Section 14.4 use highly dependent solutions by placing the sampling in successive frames as consistent as possible. Essentially this tries to keep the remaining error as temporally consistent as possible. However, some change in the samples must be allowed in order to adapt the solution to the changing environment.

Another source of temporal artifacts is the occasional under-sampling of contributions by some light sources. This results in 'missing' illumination that may suddenly 'reappear' if the importance of the light source increases above the threshold. For example, imagine approaching a far away room that can only be seen through a small door: While far away this room may not receive any VPL and thus remains completely dark. When approaching its importance increases until it will receive its first VPL. This appears as if the light in this room had suddenly been “switched on”.

The temporal artifacts can best be judged using the video accompanying the original paper[5] [Wald03c], which shows several example walkthroughs through our test scenes with both the original and with the new method. Though our method cannot completely avoid *all* temporal artifacts, it significantly improves the overall image quality of such walkthroughs, and already provides a good impression of the lighting in the scene. Note that in all of our experiments all illumination is fully recomputed every frame, allowing to arbitrarily and interactively change any lighting parameters, materials, and geometry at any time.

14.5.2 Localization vs. Non-localization

One obvious extension of our method would be to localize the importance sampling procedure by having each pixel (or set of pixels) choose its own subset of VPLs. This would allow for using different VPLs in different parts of the scene, and should reduce the above-mentioned undersampling artifacts. However, this localization would incur a high additional per-pixel cost for determining the VPLs, and would destroy the streaming framework of the IGI system, which exploits the fact that all rays are doing exactly the same [Benthin03] (also see the preceding chapter). Furthermore, it is unclear how the “jumping” of VPLs from image region to image region could be handled efficiently.

[5] This video is available from http://www.openrt.de/Publications/.

14.5.3 Estimation cost

One of the most obvious questions to quantify is the cost for the estimation step, which however is hard to determine: One obvious cost factor is the additional number of rays used for estimation. This, however, is very small compared to the bulk of the rays that is spent on shadow computations.

Unfortunately however the rays used for the estimate are much less coherent than the shadow rays for the instant radiosity step and can be significantly more costly. (up to two to three times as costly depending on scene, view, and parameters)[6]. Additionally, the estimation step requires other costly operations like sampling light sources or path directions.

Additionally to shooting the estimation paths, there is also an additional cost for sending the estimates across the network. This cost however is not significant. Similarly, the cost for combining the separate estimates on the server is negligible.

Adding all these cost factors for our estimation step together, the runtime of a single frame increases by about 20 to 50 percent compared to the unmodified algorithm when using the same number of VPLs during rendering[7]. However, because we drastically reduce the number of shadow rays (the main cost of the lighting simulation) that are required to reach a certain image quality we still obtain a significant net improvement. For example, if we would spend an additional 50% of the original time on the estimate and only manage to save 90% of all shadow rays, our method will still result in a net speedup by a factor of roughly five. The impact in practice is typically even higher and increases for more complex and occluded scenes. Because the estimation parameter and thus its cost can be chosen interactively the user can fine-tuned this tradeoff.

14.5.4 Overall Performance

Due to the discussed problems in exactly quantifying the impact of our method in detail, the best way of judging the improvements of our methods is to compare both methods side by side at the same frame rate. Therefore, we have taken the new method and have modified the quality parameters to

[6] Also note that unlike the bulk of the rays during rendering these sampling rays are not coherent enough to be shot with the fast SSE code, and have to be shot with the single-ray C code that is usually much slower (also see Table 7.6).

[7] Apart from the estimation overhead, this is also influenced by the fact that occluded shadow rays tend to be cheaper than non-occluded ones. As our method concentrates samples to non-occluded light sources, the shadow rays get slightly more costly. Even so, the biggest factor certainly is the estimation overhead.

achieve a reasonably good tradeoff for image quality versus rendering performance as it would typically be used with the original system in less complex scenes. For the comparison, we have then taken the original IGI system and adjusted its quality settings until the same frame rate was obtained.

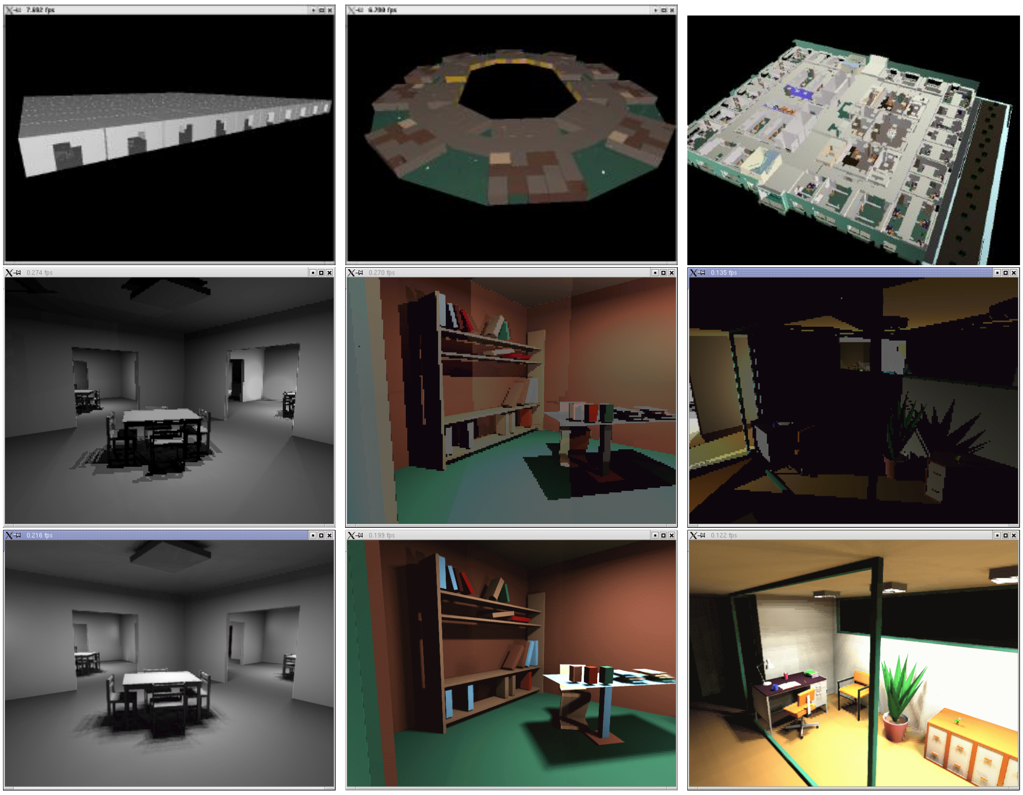


Figure 14.8: Comparison of the image quality of the original instant global illumination method (middle) versus our new importance sampling technique (bottom row) in the ERW10 (80k triangles, 100 lights), Ellipse (19k triangles, 1,164 lights), and Soda Hall scenes (2.2M triangles, 23,256 lights), respectively, while running at the same frame rate. Due to the cost for estimation, the lower row uses even less VPLs than the upper row (usually about 50%). While the images computed with the old method are hardly recognizable the new method produces images of reasonably quality at the same frame rate. Note that the lower row uses even fewer VPLs due to the cost for estimation.

The results of this side-by-side comparison can be seen in Figure 14.8: The image quality of the new method is generally much higher than with the original method. Whereas the original rendering quality is simply not tolerable at the given frame rate our method allows for image quality that is reasonably smooth. Although some artifacts are still visible, it nicely reproduces illumination features such as soft shadows that have not been possible with the original method.

14.6 Summary and Conclusions

In this chapter, we have presented an efficient importance sampling technique for computing global illumination in complex and highly occluded scenes such as entire buildings. Using a cheap and purely view-importance driven estimation step our method can efficiently avoid sampling most occluded light sources. Thus, the sampling effort is concentrated almost exclusively on light sources actually contributing to an image. At the same frame rate, this results in a significantly improved image quality.

Applying our importance sampling technique to the instant global illumination system for the first time allows for interactively and automatically exploring entire buildings illuminated with highly complex geometry and thousands of light sources. It is important to note that no expensive or manual preprocessing of the scenes has been necessary. However, some temporal artifacts remain and become visible as flickering.

We expect that future refinement of the importance estimate will be able to reduce these temporal artifacts. More samples due to high-performance realtime ray tracing, possible even with hardware support, would also help.

Even today our method allows for interactive walkthroughs under full global illumination with reasonably good quality that would be sufficient for most practical applications. It is important to note that this is possible even with models that many other rendering algorithms can hardly render at all even without computing global illumination.

Chapter 15

Instant Global Illumination – Conclusions and Future Work

As has been described in the previous chapters, realtime ray tracing can – at least if employed correctly – finally enable the computation of global illumination at interactive rates. After having outlined the issues and constraints of using realtime ray tracing for global illumination, we have presented the "Instant Global Illumination" method, a framework for global illumination that has been explicitly designed to run efficiently on a highly efficient, distributed ray tracing infrastructure, be it a software implementation or a hardware architecture. Instant Global Illumination can faithfully reproduce the most important lighting effects – smooth and hard shadows, smooth direct and indirect diffuse lighting, color bleeding, reflections and refractions – while still allowing for interactive and dynamic changes to all scene properties including geometry, materials and lights.

Instant global illumination scales to massively complex scenes of millions of polygons, and – at least with the proposed modifications – even to scenes with many light sources and high occlusion.

In the way proposed above, IGI scales linearly in the number of client PCs, and achieves frame rates of up to 40 frames per second at 640 × 480 pixels, even including procedural shaders, textures, tone mapping and antialising. Even so, much work remains to be done:

Higher performance: Obviously, it has to be investigated how the overall system performance can be furtherly increased. This includes both optimizations to the global illumination subsystem itself, as well as investigating methods to further accelerate the ray tracing core.

Once the overall performance gets furtherly increased, it also has to be evaluated how better networking technology could be used to remove the currently remaining network bottleneck of at most 40 frames per second at video resolutions. For practical applications, frame rates of 10–20 frames per second at full-screen resolutions would be desirable.

At such high resolutions, it would also make sense to re-investigate the optimal filter sizes for interleaved sampling and discontinuity buffering. While we currently use only 3 × 3, larger resolutions should also allow for filter sizes of 5 × 5 or even 7 × 7 without producing significantly visible artifacts. Similarly, as high resolutions might allow for larger packet sizes in the ray tracing core, such as packets of 4 × 4 rays instead of using only 4 rays per packet.

Hardware implementation: Apart from faster networking hardware, realtime performance at fullscreen frame rates could also be achieved by mapping the instant global illumination method to a hardware architecture such as the SaarCOR architecture (see Section 5.3). Due to its design, the instant global illumination method should map very well to such an architecture, and as such seems promising. Though this has not been finally simulated, is is currently being evaluated. Prelimirary results are promising.

Hierarchical evaluation model: In the algorithm itself, it has to be evaluated whether – and how – the instant global illumination algorithm can be modified to support a hierarchical evaluation model. Hierarchical methods and clustering have been highly successful for radiosity [Hanrahan91, Cohen93, Smits94, Sillion94, Bekaert99, Stamminger99], and might also be beneficial for instant global illumination. However, it is yet unclear how such an algorithm could be realized.

Glossyness and Caustics: Qualitatively, it is of major importance to seek for better ways of supporting highly glossy scenes. In its current form, instant global illumination is similar to radiosity methods in the sense that it performs best for scenes that are primarily diffuse. While IGI does not suffer from tesselation problems and can easily handle reflections and refractions, other specular effects – like glossiness and high-quality caustics – are still problematic. The latter effects are often less important for practical applications, but would nonetheless be interesting to support.

However, it is yet unclear how this can be realized. The most successful approach to caustics known today is photon mapping [Jensen96, Jensen97, Jensen01]. As discussed above, this unfortunately does not easily match a realtime ray tracing framework. Efficient alternatives for generating high-quality caustics so far are not yet known.

Essentially the same is true for highly glossy materials. While perfectly specular materials are already supported by computing reflections, glossy reflections are typically computed by sampling the glossy lobe with many secondary rays. Though this is theoretically simple, in practice it often leads to an unaffordable number of rays that have to be traced per pixel.

As such, much work still remains to be done for the ultimate goal of realtime, full-screen global illumination including all effects. Nonetheless, even in its current form, interactive global illumination offers a significant improvement in realism in particular during interactive sessions. This strongly indicates that interactive global illumination may eventually become a mainstream feature of 3D graphics.

Chapter 16

Final Summary, Conclusions, and Future Work

At the end of this thesis, we will first briefly summarize the content and new contributions that have been discussed in the previous chapters. Before concluding, we will also discuss and summarize all the potential areas of future research that have already been pointed out throughout this thesis.

Part I – Realtime Ray Tracing

In the first part of this thesis, we have first summarized the basics of ray tracing, and have discussed the advantages and disadvantages that ray tracing offers over triangle rasterization. We have shown that ray tracing is not only superior in offline rendering, but offers many advantages also – and especially – if used in an interactive context.

After having pointed out the advantages of ray tracing for future realtime graphics, we have briefly reviewed the different ongoing approaches towards realizing realtime ray tracing. These approaches include software-based systems (on shared-memory supercomputers as well as on commodity CPUs), approaches using the GPU as a ray tracing co-processor, and the design of specialized ray tracing hardware. All of these three approaches are currently being pursued both actively and successfully. While all these approaches have different advantages and disadvantages, we can conclude that the main remaining question no longer is *whether* realtime ray tracing will ever be commonly available on the desktop, but rather *in which form* this will come to happen.

Part II – The RTRT/OpenRT Realtime Ray Tracing Engine

In the second part of this thesis, we have taken a closer look at one of these approaches, by describing and discussing the RTRT/OpenRT realtime ray tracing engine. This software based ray tracing system achieves interactive performance by combining several techniques: First, an efficient implementation of the core ray tracing algorithms, namely fast triangle intersection, fast BSP traversal, and construction of high-quality BSPs. Second, optimizations towards the features and limitation of modern CPUs, including SIMD support and a special emphasis on caching and memory optimizations. Third, a sophisticated parallelization framework, which allows for efficiently and linearly scaling the performance of the ray tracing engine by adding more CPUs, even if those are located on different PCs.

Based on these core algorithms, we have then discussed some of the advanced issues of ray tracing whose importance is not limited to the previously described software implementation, but that have to be addressed no matter what kind of ray tracing architecture will eventually prove to be the most successful, be it software, hardware, or GPUs: First, we have discussed methods that allow for efficiently handling dynamic scenes in a realtime ray tracing system. We have then discussed API issues for realtime ray tracing, and have proposed the OpenRT interactive ray tracing API, an API that is as similar to OpenGL as possible, while still allowing for arbitrarily programmable shading as possible in RenderMan.

Taking all these measures together, the RTRT/OpenRT engine is a complete, extremely powerful rendering engine that offers arbitrarily programmable shading, support for highly complex geometry, efficient handling of dynamic scenes, and high performance, all available through a simple yet flexible and powerful API. This engine enables a set of uniquely new applications that cannot easily be realized with different approaches. Several practical examples of such applications have been demonstrated.

Part III – Instant Global Illumination

In the third – and final – part of this thesis we have then described the "Instant Global Illumination" method, a global illumination framework that has been explicitly designed to meet the constraints and restrictions of a realtime ray tracing system. Running on top of the above-mentioned RTRT/OpenRT engine, instant global illumination allows for finally achieving interactive global illumination – the physically correct simulation of light propagation in a scene at interactive rates. After describing the basic idea of the Instant Global Illumination method, we have discussed several improvements and optimizations of this method, including a faster and scalable implementation, support for programmable shading, and efficient anti-aliasing.

Finally, we have shown how – using an efficient importance estimation and importance sampling scheme – the Instant Global Illumination method can be extended to also handle massively complex and highly occluded scenes with millions of triangles and thousands of light sources.

Future Research

Being a relatively new area of research, realtime ray tracing has received few attention by researchers so far. While this thesis has discussed the most important issues of realtime ray tracing and interactive global illumination, most of these previously discussed techniques allow for extensions and future research.

Though many of these issues have already been mentioned in their respective chapters earlier in this thesis, we will briefly repeat them here, in order to present all these open issues side by side.

Faster Ray Tracing Implementations

Obviously, realtime ray tracing can be further improved my making it even faster. While we have spent significant effort in achieving high ray tracing performance, we are nonetheless convinced that further improvements are possible. As the biggest potential for furtherly optimizing ray tracing probably lies in the better exploitation of coherence, future research in accelerating ray tracing may also require to find new algorithms that concentrate on shooting rays in a more coherent way.

Efficient Shading

In order to improving the *effective* ray tracing performance one also has to reduce the performance impact of shading. As already discussed in Section 7, even simple shading has now become the bottleneck of the RTRT/OpenRT system, and for many scenes is more costly than tracing and intersecting the rays. However, how to best accelerate the shading process is still an open question.

The streaming-like shading in the Instant Global Illumination (see Section 13.3) has already shown up first successful steps into that direction, and has thereby shown the potential of this approach. However, it is yet unclear how exactly to solve that problem for arbitrarily programmable, unrestricted kinds of shaders. Most likely, research on faster shading will eventually also require to look out for more coherent rendering algorithms, design new shading APIs, and probably also on shading language compilers.

Better Support for Dynamic Scenes

Apart from higher performance in ray tracing and shading, establishing ray tracing as an alternative to triangle rasterization also requires the investigation and design of new methods for better handling dynamic scenes. Especially for highly interactive applications such as computer games, efficient support for dynamic scenes will be a prime prerequisite. This will especially require methods that can efficiently be used in parallel settings, and that can also be realized on potential future hardware architectures such as SaarCOR.

Construction of Ray Tracing Hardware

Commodity CPUs get increasingly faster every year, and in a few years will eventually be fast enough to deliver full-screen realtime ray tracing performance on a single desktop PC. On the other hand, once more compute power is available, it is likely to be used immediately for even further increasing the rendering quality, e.g. by antialiasing, even higher resolutions and framerates, even more complex shading, global illumination, etc.

As such, in order to reach the scenario of realtime ray tracing being cheaply available on everybodys desktop, it might be necessary to spend more effort on designing – and eventually building and marketing – ray tracing hardware. Eventually, such a different hardware platform may also require to re-investigate the previously discussed isses of dynamic scenes, proper API support, and efficient shading also from a hardware perspective.

Support for Non-Polygonal Primitive Types

In order to make ray tracing useful for a wide range of applications, we also have to investigate means of efficiently ray tracing non-polygonal geometry such as lines, text, or points. While these are often used by practical applications, they are surprisingly hard to (efficiently) support in ray tracing. Except for these kinds of primitives, many applications could benefit from the direct, built-in support for higher-order primitives such as Bezier- or NURBS-patches, or subdivision surfaces. Similarly, built-in support for interactively ray tracing point clouds, volumetric data, or ISO-surfaces would enable completely new kinds of applications, especially if all these different kinds of rendering primitives can be supported in one single framework.

While the respective algorithms for ray tracing these kinds of primitives are already well-known, the performance of these algorithms has to be significantly improved in order to meet the performance requirements of a realtime ray tracing system. This especially includes the question how to be able to support these primitives *without* thereby reducing the ray tracing performance for polygonal scenes.

Better API Support

Furthermore, more work has to be investigated into better API support for ray tracing. The most crucial issues of such future ray tracing APIs will be the support for dynamic scenes, the support for advanced primitive types, as well as the way that shading is being expressed (i.e. the shading language and shader API). However, as long as it is not clear how these respective issues themselves are best solved in a ray tracer (see above), it is not appropriate to address them already at the API level.

In the end, designing new APIs that are suitable for ray tracing may also lead to the convergence of rasterization and ray tracing APIs. Eventually, it would be advantageous to have one single, common API that equally supports triangle rasterization, point-based rendering, and ray tracing.

Extensions to Instant Global Illumination

Currently, the main limitation of the Instant Global Illumination technique is its dependence on mostly diffuse scenes (plus perfectly specular reflections), and its missing support for glossy materials and caustics. For many practical applications (especially in the VR industry), high frame rates, good scalability and a high-quality simulation of diffuse inter-reflection are actually much more important than support for caustics. On the other hand, many applications would benefit from the accurate caustics, at least if this does not corrupt the achievable performance and the quality of the diffuse lighting simulation. As photon mapping is currently the best known method for producing convincing shadows, support for high-quality caustics will eventually also require to design methods and techniques for realtime photon mapping.

Even more important however might be the support for “typical” real-world effects like glossy reflections, motion blur, or depth of field. These techniques are standard features for offline rendering, but are currently too expensive to be simulated at realtime rates.

Building Practical Applications

Finally, it has to be evaluated how all this vast potential that has now become available – massively complex scenes, high-quality and physically correct shading, and even global illumination – can be used for real-world, practical applications. Eventually, this is likely to further encourage the transition from purely "good-looking" images to "predictive" rendering being based on the accurate simulation of real-world phenomena. This however will also require to take a closer look at more realistic lighting and material models than are currently used in mainstream computer graphics.

Conclusions

Though the previous discussion has shown that there are enough open questions left to be answered, realtime ray tracing in recent time has made tremendous progress towards offering a practical alternative to triangle rasterization. Already today, it enables completely new applications that are currently impossible to realize with any other technique.

With all these uniquely new application that are already possible today, realtime ray tracing is likely to play a larger role in future 3D graphics. Similar to the introduction of texture mapping a few decades ago, it is likely to provide a new level of realism that is likely to soon be a standard feature of mainstream graphical applications.

Zusammenfassung

Die im Rahmen dieser Arbeit bearbeiteten Themen gliedern sich in drei Teile: Erstens, in Grundlagen, Zusammenfassung, und aktueller Stand der Dinge des Echtzeit Ray-Tracing; zweitens, in die Beschreibung des RTRT/OpenRT Echtzeit Ray-Tracing Systems; sowie drittens, die Benutzung des Echtzeit Ray-Tracings für die Interaktive Globale Beleuchtungsberechnung mittels "Instant Global Illumination".

Teil I – Echtzeit Ray-Tracing

Im ersten Teil dieser Arbeit wurden sowohl die Grundlagen des Ray-Tracing dargestellt, als auch die Vor- und Nachteile des Ray-Tracing im Vergleich zur Dreiecksrasterisierung aufgezeigt. Darauf basierend wurde gezeigt, daß das Ray-Tracing Verfahren nicht nur bei der nicht-interaktiven Bildsynthese vorteilhafter ist als Dreiecksrasterisierung, sondern dass es auch – und inbesondere – im interaktiven Kontext Vorteile bietet.

Nachdem diese Vorteile des Ray-Tracings fuer künftige Echtzeit-Graphik aufgezeigt wurden, folgte eine kurze Übersicht über die zur Zeit von unterschiedlichen Gruppen verfolgten Ansätze zur Verwirklichung dieser Echtzeit Ray-Tracing Technologie. Diese Ansätze kann man in drei Kategorien klassifizieren: Software-basierte Systemen (sowohl auf Mehrprozessor-Supercomputern als auch auf Standard-PC-Prozessoren), Ansätze die Graphikchips als Ray-Tracing Koprozessoren verwenden, sowie letztendlich das Design auf Ray-Tracing spezialisierter Hardware. Alle diese Ansätze werden heutzutage aktiv und zeitgleich mit großen Fortschritten weiterverfolgt. Während alle diese unterschiedlichen Ansätze unterschiedliche Stärken und Schwächen aufweisen, so stellt sich doch heute weniger die Frage, *ob* Echtzeit Ray-Tracing jemals eine weitverbreitete Graphiktechnologie sein wird, sondern nur noch, *in welcher Form* diese Entwicklung vonstatten gehen wird.

Teil II – Das RTRT/OpenRT Echtzeit Ray-Tracing System

Im zweiten Teil dieser Arbeit wurde dann einer dieser Ansätze genauer untersucht, indem die RTRT/OpenRT Echtzeit Ray-Tracing Architektur genauer beschrieben und diskutiert wurde. Dieses Software-basierte System erreicht interaktive Performanz durch die Kombination mehrerer Techniken: Erstens, eine effizient, hochgradig optimierte Implementierung der Kernalgorithmen des Ray-Tracing, insbesondere schnelle Strahl-Dreiecks-Schnittberechnungen, schnelles Traversieren von BSP-Bäumen, sowie die Generierung hochqualitativer BSPs. Zweitens, Optimierungen, die speziell auf die Eigenschaften moderner Prozessoren abgestimmt sind, insbesondere die Konzentration auf Caching-Effekte und die effiziente Unterstützung von SIMD-Erweiterungen. Drittens, eine ausgereifte Parallelisierungs-Architektur, die es erlaubt, die Performanz der Ray-Tracing Architektur linear und transparent dadurch zu steigern, dass die Rechenkapazitäten mehrer Prozessoren (bzw. PCs) kombiniert werden.

Basierend auf diesen Algorithmen wurden dann einige der fortgeschritteneren Themen des Echtzeit Ray-Tracing angesprochen, welche nicht auf diese Software-basierte Architektur beschränkt sind, sondern auch für alternative Ansätze des Echtzeit Ray-Tracing (wie GPUs und spezialisierte Ray-Tracing Hardware)in gleicher oder ähnlicher Form behandelt werden müssen: Dazu wurde zuerst eine Methode diskutiert, die es erlauben, auch dynamisch änderbare Szenen effizient in einem Echtzeit Ray-Tracer zu unterstützen. Danach wurden dann Anforderungen an Anwendungsschnittstellen (API's) für das Echtzeit Ray-Tracing diskutiert, wobei auch das OpenRT API vorgestellt wurde. Dieses OpenRT API wurde dabei speziell daraufhin entworfen, so ähnlich wie moeglich zu existierenden APIs fuer Echtzeit Graphik (OpenGL) zu erscheinen, dabei aber auch die Stärken des Ray-Tracing zu unterstützen, und dabei auch frei programmierbares, RenderMan-ähnliches Shading zu unterstützen.

Die Integration aller dieser Komponenten liefert dann die RTRT/OpenRT Echtzeit Ray-Tracing Architektur, eine vollständige, extrem mächtige Graphikarchitektur, welche sich auszeichnet durch frei programmierbares Shading, die Unterstützung auch massivst komplexer Geometrien, effiziente Handhabung dynamisch änderbarer Szenen, sowie hohe, interaktive Performanz; alle die für Applikationen zugänglich durch eine einfache aber nichtsdestotrotz mächtige Anwendungsschnittstelle. Diese Graphikarchitektur ermöglicht eine Anzahl von grundlegend neuen Anwendungen, welche nur schwer mit anderen Ansätzen realisierbar wären. Mehrere praktische Beispiele solcher Anwendungen wurden vorgestellt.

Teil III – Interaktive Beleuchtungssimulation durch “Instant Global Illumination”

Im dritten – und abschließenden – Teil dieser Arbeit wurde dann diskutiert, welche Implikationen sich durch die jetzt verfügbar gewordene Echtzeit Ray-Tracing Technologie auf dem Feld der Globalen Beleuchtungsberechnung ergeben. Dazu wurde die “Instant Global Illumination” Methode vorgestellt, welche explizit dahingehend entworfen wurde, die Voraussetzungen und Beschränkungen eines Echtzeit Ray-Tracing Systems zu erfüllen. In Kombination mit der vorgehend besprochenen RTRT/OpenRT Architektur ermöglicht diese Instant Global Illumination Methode nun endlich auch die Globale Belechtung – das heißt, die physikalisch korrekte Berechnung des Lichttransports – zu interaktiven Bildwiederholraten. Nach der Beschreibung der Grundlagen dieser Methode wurden dann noch einige Erweiterungen dieser Methode vorgestellt, insbesondere sowohl Verbesserungen und Optimierungen der Performanz und Skalierbarkeit der Methode, als auch Unterstützung für programmierbares Shading und effizientes Anti-Aliasing, sowie die Unterstützung massiv komplexer und hochgradig durch Verdeckung geprägter Szenen.

Schlußfolgerung

Obwohl die Entwicklung und die Fortschritte des Echtzeit Ray-Tracing heutzutage sicherlich noch nicht abgeschlossen sind, so erlauben doch auch die im Rahmen dieser Arbeit vorgestellten Möglichkeiten schon vollkommen neue Anwendungen, welche vorher in dieser Form nicht denkbar waren. Ähnlich wie die Einführung der Texturierung vor wenigen Jahrzehnten erlaubt diese neue Technologie einen neue Ebene des Realismus in graphischen Anwendungen, der vermutlich schon bald zur Standardausstattung auch von Alltags-Graphikanwendungen gehören wird.

Appendix A
List of Related Papers

Parts of this PhD thesis have already been published in previous publications. The following is a list of papers that have contributed to this thesis, and that might contain additional information.

Note however that some of these publications by now are significantly older than this thesis, and thus may contain information (such as performance data) that is already outdated by now. If in doubt, the information and data given in this thesis should precede all information and data given in any of these papers.

2001

Interactive Rendering using Coherent Ray Tracing
Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek
Computer Graphics Forum, 20(3), 2001, pages 153–164, by A. Chalmers and T.–M. Rhyne (editors), Blackwell Publishers, Oxford, (Proceedings of Eurographics 2001), Manchester, UK [Wald01a]

Interactive Distributed Ray Tracing of Highly Complex Models
Ingo Wald, Philipp Slusallek, and Carsten Benthin
Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering), by Steven J. Gortler and Karol Myszkowski (editors), pages 274–285, pages 2001, London, UK [Wald01c]

State of the Art in Interactive Ray Tracing
Ingo Wald and Philipp Slusallek
Eurographics 2001 State-of-the-Art Reports, Manchester, UK
[Wald01b]

2002

Interactive Global Illumination using Fast Ray Tracing
Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, Philipp Slusallek, *Rendering Techniques 2002*, by P. Debevec and S. Gibson (editors) pages 15–24, 2002, Pisa, Italy, (Proceedings of the 13th Eurographics Workshop on Rendering) [Wald02b]

Interactive Headlight Visualization –A Case Study of Interactive Distributed Ray Tracing–
Carsten Benthin, Ingo Wald, Tim Dahmen and Philipp Slusallek, Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization (PVG), pages 81–88, Blaubeuren, Germany, 2002 [Benthin02]

SaarCOR – A Hardware Architecture for Ray Tracing
Jörg Schmittler, Ingo Wald, and Philipp Slusallek, Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware, 2002, pages 27–36, 2002, Saarbrücken, Germany [Schmittler02]

OpenRT – A Flexible and Scalable Rendering Engine for Interactive 3D Graphics
Ingo Wald, Carsten Benthin, and Philipp Slusallek, Technical Report 2002, Saarland University Saarbrücken, Germany [Wald02a]

2003

Towards Realtime Ray Tracing – Issues and Potential
Ingo Wald, Carsten Benthin, and Philipp Slusallek, Technical Report 2003, Saarland University Saarbrücken, Germany [Wald03d]

The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing –
Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek, OpenSG Symposium 2003, Darmstadt, Germany [Dietrich03]

Interactive Global Illumination in Complex and Highly Occluded Scenes
Ingo Wald, Carsten Benthin, and Philipp Slusallek, Proceedings of the 14th Eurographics Symposium on Rendering, by P. H. Christensen and D. Cohen-Or (editors) pages 74–81, 2003, Leuven, Belgium [Wald03c]

Interactive Distributed Ray Tracing on Commodity PC Clusters –State of the Art and Practical Applications –
Ingo Wald, Carsten Benthin, and Philipp Slusallek, in Harald Kosch, Laszlo Böszörmenyi, and Hermann Hellwagner, editors, *Euro-Par 2003*, Klagenfurt, Austria, volume 2790 of *Lecture Notes in Computer Science*, Springer. [Wald03a]

A Scalable Approach to Interactive Global Illumination
Carsten Benthin, Ingo Wald, and Philipp Slusallek, in *Computer Graphics Forum*, 22(3), 2003, pages, 621–630, (Proceedings of Eurographics 2003), Granada, Spain [Benthin03]

Distributed Interactive Ray Tracing in Dynamic Environments
Ingo Wald, Carsten Benthin, and Philipp Slusallek, SIGGRAPH/Eurographics Workshop on Parallel Graphics and Visualization (PVG) 2003, Seattle, WA, USA, pages 77-86 [Wald03b]

Realtime Ray Tracing and its use for Interactive Global Illumination
Ingo Wald, Timothy J.Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek, Eurographics 2003 State-of-the-Art Reports, Granada, Spain [Wald03e]

Streaming Video Textures for Mixed Reality Applications in Interactive Ray Tracing Environments
Andreas Pomi, Gerd Marmitt, Ingo Wald, and Philipp Slusallek, Virtual Reality, Modelling and Visualization (VMV) 2003, Munich, Germany [Pomi03]

VRML Scene Graphs on an Interactive Ray Tracing Engine
Andreas Dietrich, Ingo Wald, Markus Wagner and Philipp Slusallek, IEEE VR 2004 (to appear) [Dietrich04]

Bibliography

[Agelet97] *Fernando Aguado Agelet, Fernando Perez Fontan, and Arno Formella.* Fast Ray Tracing for Microcellular and Indoor Environments. IEEE Transactions on Magnetics (to appear), March 1997.

[Akenine-Möller02] *Tomas Akenine-Möller and Eric Haines. Realtime Rendering (2nd edition).* A K Peters Ltd, July 2002. ISBN: 1568811829.

[Aliaga99] *Daniel G. Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kennth E. Hoff III, Tom Hudson, Wolfgang Stürzlinger, Rui Bastos, Mary C. Whitton, Frederick P. Brooks Jr., and Dinesh Manocha.* MMR: An Interactive Massive Model Rendering System using Geometric and Image-Based Acceleration. In *ACM Symposium on Interactive 3D Graphics*, pages 199–206, Atlanta, USA, April 1999.

[AltiVec] Motorola Inc. *AltiVec Technology Facts.* Available at http://www.motorola.com/AltiVec/facts.html.

[Amanatides87] *John Amanatides and Andrew Woo.* A Fast Voxel Traversal Algorithm for Ray Tracing. In G. Marechal, editor, *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.

[AMDa] Advanced Micro Devices. *Inside 3DNow![tm] Technology.* http://www.amd.com/products/cpg/k623d/-inside3d.html.

[AMDb] *Advanced Micro Devices.* Software Optimization Guide for AMD Athlon(tm) 64 and AMD Opteron(tm) Processors. Available from http://www.amd.com/us-en/Processors/TechnicalResources/.

[AMD03a] *Advanced Micro Devices.* AMD Athlon XP Processor Model 8 Data Sheet. Available from http://www.amd.com/us-en/Processors/, March 2003. Publication number 25175.

[AMD03b] *Advanced Micro Devices.* AMD Opteron Processor Model 8 Data Sheet. http://www.amd.com/us-en/-Processors, 2003.

[Anido02] *Manuel Anido, Nozar Tabrizi, Haitao Du, Marcos Sanchez-Elez, and Nader Bagherzadeh.* Interactive Ray Tracing using a SIMD Reconfigurable Architecture. In *Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing*, pages 20–28. IEEE Computer Soc, 2002.

[Apodaca90] *A. Apodaca and M. Mantle.* RenderMan: Pursuing the Future of Graphics. *IEEE Computer Graphics & Applications*, 10(4):44–49, July 1990.

[Apodaka00] *Anthony Apodaka and Larry Gritz. Advanced RenderMan: Creating CGI for Motion Pictures.* Morgan Kaufmann, 2000. ISBN: 1558606181.

[Appel68] *Arthur Appel.* Some Techniques for Shading Machine Renderings of Solids. *SJCC*, pages 27–45, 1968.

[ART] *Advanced Rendering Technologies.* http://www.art.-eco.uk/.

[Artusi03] *Alessandro Artusi, Jiri Bittner, Michael Wimmer, and Alexander Wilkie.* Delivering Interactivity to Complex Tone Mapping Operators. In Per H. Christensen and Daniel Cohen-Or, editors, *Proceedings of the 2003 Eurographics Symposium on Rendering*, pages 38–51, 2003.

[Arvo87] *James Arvo and David Kirk.* Fast Ray Tracing by Ray Classification. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):55–64, July 1987.

[Arvo88] *J. Arvo.* Linear-time Voxel Walking for Octrees. *Ray Tracing News (available at htpp://www.acm.org/tog/-resources/RTNews/html/rtnews2d.html*, 1(5), March 1988.

[Arvo90a] *James Arvo.* Ray Tracing with Meta-Hierarchies. In *SIGGRAPH '90 Advanced Topics in Ray Tracing course notes.* ACM Press, August 1990.

[Arvo90b] *James Arvo and David B. Kirk.* Particle Transport and Image Synthesis. *Computer Graphics (Proceedings of SIGGRAPH '90)*, 24(4):63–66, August 1990.

[ATI02] *ATI.* Radeon 9700 Pro Product Web Site, 2002. http://mirror.ati.com/products/pc/radeon9700pro/-index.html.

[Aupperle93] *Larry Aupperle and Pat Hanrahan.* Importance and Discrete Three Point Transport. In *Proceedings of the Fourth Eurographics Workshop on Rendering*, pages 85–94, June 1993.

[Badouel92] *Didier Badouel.* An Efficient Ray Polygon Intersection. In David Kirk, editor, *Graphics Gems III*, pages 390–393. Academic Press, 1992. ISBN: 0124096735.

[Bala99] *Kavita Bala, Julie Dorsey, and Seth Teller.* Radiance Interpolants for Accelerated Bounded-Error Ray Tracing. *ACM Transactions on Graphics*, 18(3):213–256, August 1999.

[Bala03] *Kavita Bala, Bruce Walter, and Donald Greenberg.* Combining Edges and Points for Interactive High-Quality Rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, pages 631–640, July 2003. ISSN:0730-0301.

[Baxter III02] *William V. Baxter III, Avneesh Sud, Naga K Govindaraju, and Dinesh Manocha.* GigaWalk: Interactive Walkthrough of Complex Environments. In *Rendering Techniques 2002 (Proceedings of the 13th Eurographics Workshop on Rendering)*, pages 203 – 214, June 2002.

[Bekaert99] *Philippe Bekaert.* Hierarchical and Stochastic Algorithms for Radiosity. PhD thesis, Katholieke Universitiet Leuven, Belgium, 1999.

[Bekaert01] *Philippe Bekaert.* Extensible Scene Graph Manager, August 2001. http://www.cs.kuleuven.ac.be/∼graphics-/XRML/.

[Benthin02] *Carsten Benthin, Ingo Wald, Tim Dahmen, and Philipp Slusallek.* Interactive Headlight Simulation – A Case Study of Distributed Interactive Ray Tracing. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization (PGV)*, pages 81–88. Eurographics Association, 2002. ISBN: 1-58113-579-3.

[Benthin03] *Carsten Benthin, Ingo Wald, and Philipp Slusallek.* A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum*, 22(3):621–630, 2003. (Proceedings of Eurographics).

[Bentley75] *Jon Louis Bentley.* Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Bentley79] *Jon Louis Bentley and Jerome H. Friedman.* Data structures for range searching. *Computing surveys*, 11(4):397–409, 1979.

[Bittner99] *Jiri Bittner.* Hierarchical Techniques for Visibility Determination. Technical Report DS-005, Department of Computer Science and Engineering, Czech Technical University in Prague, March 1999. Also available as http:/www.cgg.cvut.cz/∼bittner/publications/-minimum.ps.gz.

[Bolz03] *Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroeder.* Sparse Matrix Solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3):917–924, 2003. ISSN: 0730-0301.

[Carey97] *Rikk Carey, Gavin Bell, and Chris Marrin.* ISO/IEC 14772-1:1997 Virtual Reality Modelling Language (VRML97), April 1997. http://www.vrml.org/Specifications/VRML97.

[Carr02] *Nathan A. Carr, Jesse D. Hall, and John C. Hart.* The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association, 2002. ISBN: 1-58113-580-7.

[Cassen95] *T. Cassen, K. R. Subramanian, and Z. Michalewicz.* Near-Optimal Construction of Partitioning Trees by Evolutionary Techniques. In *Proceedings of Graphics Interface '95*, pages 263–271, Canada, June 1995.

[Chalmers02] Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical Parallel Rendering.* AK Peters, 2002. ISBN 1-56881-179-9.

[Chase00] *Ronald J. Chase, Thomas M. Kendall, and Steven R. Thompson.* Parametric Radar Cross Section Study of a Ground Combat Vehicle. Technical report, Army Research Laboratory and Raytheon corp., 2000.

[Chen91] *Shenchang Eric Chen, Holly E. Rushmeier, Gavin Miller, and Douglass Turner.* A progressive multi-pass method for global illumination. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 165–174. ACM Press, 1991. ISBN: 0-89791-436-8.

[Cleary86] *John G. Cleary, Brian M. Wyvill, Graham M. Birtwistle, and Reddy Vatti.* Multiprocessor Ray Tracing. In *Computer Graphics Forum*, volume 5, pages 3–12. North-Holland, Amsterdam, 1986.

[Cohen93] *Micheal F. Cohen and John R. Wallace. Radiosity and Realistic Image Synthesis.* Morgan Kaufmann Publishers, 1993. ISBN 0-12178-270-0.

[Cohen94] *Daniel Cohen.* Voxel Traversal along a 3D Line. In Paul Heckbert, editor, *Graphics Gems IV*, pages 366–369. Academic Press, 1994. ISBN: 0123361559.

[Cook84a] *Robert Cook, Thomas Porter, and Loren Carpenter.* Distributed Ray Tracing. *Computer Graphics (Proceeding of SIGGRAPH 84)*, 18(3):137–144, 1984.

[Cook84b] *Robert L. Cook.* Shade trees. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 18(3):223–231, July 1984.

[Cook87] *Robert L. Cook, Loren Carpenter, and Edwin Catmull.* The REYES Image Rendering Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 1987)*, pages 95–102, July 1987.

[Crow77] *F.C. Crow.* Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH 77 Proceedings)*. ACM Press, July 1977.

[Damez03] *Cyrille Damez, Kirill Dmitriev, and Karol Myszkowski.* State of the Art in Global Illumination for Interactive Applications and High-Quality Animations. *Computer Graphics Forum*, 22(1):55–77, March 2003.

[Dandekar99] *Kapil R. Dandekar, Alberto Arredondo, Guanghan Xu, and Hao Ling.* Using Ray Tracing to Evaluate Smart Antenna System Performance in Outdoor Wireless Communications. In *1999 SPIE 13th Annual International Symposium on Aerospace / Defense Sensing, Simulation, and Controls*, April 1999.

[Debevec97] *Paul E. Debevec and Jitendra Malik.* Recovering High Dynamic Range Radiance Maps from Photographs. *Computer Graphics*, 31(Annual Conference Series):369–378, 1997.

[Debevec98] *Paul Debevec.* Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-based Graphics with Global Illumination and High Dynamic Range Photography. *Computer Graphics (Proceedings of SIGGRAPH 98)*, 32(4):189–198, 1998.

[DeMarle03] *David E. DeMarle, Steve Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen.* Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 87–94, 2003.

[Diefenbach96] *Paul J. Diefenbach.* Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering. PhD thesis, University of Pennyslvania, 1996.

[Dietrich03] *Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek.* The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, pages 23–31, Darmstadt, Germany, 2003. Eurographics Association.

[Dietrich04] *Andreas Dietrich, Ingo Wald, Markus Wagner, and Philipp Slusallek.* VRML Scene Graphs on an Interactive Ray Tracing Engine. In *Proceedings of IEEE VR 2004*, pages 109–116, 2004.

[DirectX] Microsoft DirectX 8.0. http://www.microsoft.com/-windows/directx/.

[Dmitriev02] *Kirill Dmitriev, Stefan Brabec, Karol Myszkowski, and Hans-Peter Seidel.* Interactive Global Illumination using Selective Photon Tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 21–34, 2002.

[Drettakis97] *George Drettakis and François Sillion.* Interactive Update of Global Illumination using a Line-Space Hierarchy. In *Computer Graphics (Proceedings of SIGGRAPH 1997)*, pages 57–64, Aug 1997.

[Du03] *Haitao Du, Marcos Sanchez-Elez, Nozar Tabrizi, Nader Bagherzadeh, Manuel Anido, and Milangros Fernandez.* Interactive Ray Tracing on Reconfigurable SIMD MorphoSys. In *Design Automation and Test in Europa Conference and Exhibition*, pages 144–149, March 2003.

[Durand97] *Fredo Durand, George Drettakis, and Claude Puech.* The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool. In *SIGGRAPH 97 Conference Proceedings*, pages 89–100, 1997.

[Durand99] *Fredo Durand.* 3D Visibility: Analytical Study and Applications. PhD thesis, Universite Grenoble I – Joseph Fourier Sciences et Geographie, July 1999.

[Durgin97] *Greg D. Durgin, Neal Patwari, and Theodore S. Rappaport.* An Advanced 3D Ray Launching Method for Wireless Propagation Prediction. In *IEEE 47th Vehicular Technology Conference*, volume 2, May 1997.

[Dutre01] *Philip Dutre, Kavita Bala, and Philippe Bekaert.* Advanced Global Illumination, 2001. SIGGRAPH 2001 Course Notes, Course 20.

[Ebert02] *David Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. Texturing & Modeling: A Procedural Approach.* Morgan Kaufmann, Third edition, 2002. ISBN 1-55860-848-6.

[Eldridge00] *Matthew Eldridge, Homan Igehy, and Pat Hanrahan.* Pomegranate: A Fully Scalable Graphics Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 443–454, July 2000.

[Erickson97] *Jeff Erickson.* Pluecker Coordinates. *Ray Tracing News*, 1997. http://www.acm.org/tog/resources/-RTNews/html/rtnv10n3.html#art11.

[Fernandez02] *Sebastian Fernandez, Kavita Bala, and Donald Greenberg.* Local Illumination Environments for Direct Lighting Acceleration. In *Proceedings of 13th Eurographics Workshop on Rendering*, pages 7–14, Pisa, Italy, June 2002.

[Fernando03] *Randima Fernando and Mark J. Kilgard. The Cg Tutorial – The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley, 2003.

[Foley97] *Foley, van Dam, Feiner, and Hughes. Computer Graphics – Principles and Practice, 2nd edition in C.* Addison Wesley, 1997.

[Formella94] *Arno Formella, Christian Gill, and V. Hofmeyer.* Fast Ray Tracing of Sequences by Ray History Evaluation. In *Proceedings of Computer Animation '94*, pages 184–191. IEEE Computer Society Press, May 1994.

[Foruma] *MPI Forum.* MPI – The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi.

[Forumb] *Myrinet Forum.* Myrinet. http://www.myri.com/-myrinet/overview/.

[Fujimoto86] *Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata.* ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[Futral01] *William T. Futral. Infiniband Architecture: Development and Deployment – A Strategic Guide to Server I/O Solutions.* Intel Press, 2001.

[Gamma94] *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994. ISBN 0-201-63361-2.

[Geist94] *Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, 1994.

[Gigante88] *Michael Gigante.* Accelerated Ray Tracing using Non-Uniform Grids. In *Proceedings of Ausgraph '90*, pages 157–163, 1988.

[Glassner84] *Andrew S. Glassner.* Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[Glassner88] *Andrew Glassner.* Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, 1988.

[Glassner89] *Andrew Glassner. An Introduction to Ray Tracing.* Morgan Kaufmann, 1989. ISBN 0-12286-160-4.

[Glassner94] *Andrew S. Glassner. 3D Computer Graphics, Second Edition.* The Lyons Press, July 1994. ISBN: 1558213058.

[GNU] The GNU gcc compiler, version 3.3.1. http://www.gnu.org.

[Goldsmith87] *Jeffrey Goldsmith and John Salmon.* Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[Goodnight03] *Nolan Goodnight, Rui Wang, Cliff Woolley, and Greg Humphreys.* Interactive Time-Dependent Tone Mapping using Programmable Graphics Hardware. In Per H. Christensen and Daniel Cohen-Or, editors, *Proceedings of the 2003 Eurographics Symposium on Rendering*, pages 26–37, 2003.

[Govindaraju03] *Naga K. Govindaraju, Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha.* Interactive Shadow Generation in Complex Environments. *ACM Transaction on Graphics (Proceedings of ACM SIGGRAPH)*, 22(3):501–510, 2003.

[Granier01] *Xavier Granier, George Drettakis, and Bruce Walter.* Fast Global Illumination Including Specular Effects. In *Proceedings of the 2001 Eurographics Workshop on Rendering*, pages 47–58, 2001.

[Gritz96] *Larry Gritz and James K. Hahn.* BMRT: A Global Illumination Implementation of the RenderMan Standard. *Journal of Graphics Tools*, 1(3):29–47, 1996.

[Gröller91] *Eduard Gröller and Werner Purgathofer.* Using temporal and spatial coherence for accelerating the calculation of animation sequences. In *Proceedings of Eurographics '91*, pages 103–113. Elsevier Science Publishers, 1991.

[Haber01] *Jörg Haber, Karol Myszkowski, Hitoshi Yamauchi, and Hans-Peter Seidel.* Perceptually Guided Corrective Splatting. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):142–152, September 2001.

[Haines86] *Eric A. Haines and D. P. Greenberg.* The Light Buffer: A Ray Tracer Shadow Testing Accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, September 1986.

[Haines87] *Eric A. Haines.* A Proposal for Standard Graphics Environments. *IEEE Computer Graphics and*

Applications, 7(11):3–5, November 1987. Available from http://www.acm.org/pubs/tog/resources/-SPD/overview.html.

[Haines91a] *Eric Haines.* Efficiency Improvements for Hierarchy Traversal in Ray Tracing. In James Arvo, editor, *Graphics Gems II*, pages 267–272. Academic Press, 1991.

[Haines91b] *Eric A. Haines.* Fast Ray-Convex Polyhedron Intersection. In James Arvo, editor, *Graphics Gems II*, pages 247–250. Academic Press, San Diego, 1991. includes code.

[Hall01] *D. Hall.* The AR350: Today's ray trace rendering processor. In *Proceedings of the Eurographics/SIGGRAPH workshop on Graphics hardware - Hot 3D Session 1*, 2001.

[Hanrahan90] *Pat Hanrahan and Jim Lawson.* A language for shading and lighting calculations. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 24(4):289–298, August 1990. ISBN: 0-201-50933-4.

[Hanrahan91] *Pat Hanrahan, David Salzman, and Larry Aupperle.* A Rapid Hierarchical Radiosity Algorithm. In *Computer Graphics (SIGGRAPH 91 Conference Proceedings)*, pages 197–206, 1991.

[Harris02] *Mark Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra.* Physically-based visual simulation on graphics hardware. In *Graphics Hardware*, pages 109–118, 2002.

[Havran97] *Vlastimil Havran.* Cache Sensitive Representation for the BSP Tree. In *Compugraphics'97*, pages 369–376. GRASP – Graphics Science Promotions & Publications, December 1997.

[Havran98] *Vlastimil Havran, Jimí Bittner, and Jimi Sára.* Ray Tracing with Rope Trees. In László Szirmay Kalos, editor, *14th Spring Conference on Computer Graphics*, pages 130–140, 1998.

[Havran99] *Vlastimil Havran.* Analysis of Cache Sensitive Representation for Binary Space Partitioning Trees. *Informatica*, 23(3):203–210, May 1999. ISSN: 0350-5596.

[Havran00] *Vlastimil Havran, Jan Prikryl, and Werner Purgathofer.* Statistical Comparison of Ray-Shooting Efficiency Schemes. Technical Report TR-186-2-00-14, Department of Computer Science, Czech Technical University; Vienna University of Technology, July 2000.

[Havran01] *Vlastimil Havran.* Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Heckbert87] *Paul S. Heckbert.* Ray Tracing JELL-O (R) Brand Gelatin. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):73–4, July 1987. revision appears in CACM, Vol. 31, #2, Feb. 1988, p. 130-134.

[Heckbert92] *P. Heckbert.* Discontinuity meshing for radiosity. In *Eurographics Rendering Workshop 92 proceedings*, pages 203–226, May 1992.

[Heidrich99] *Wolfgang Heidrich.* High-quality Shading and Lighting for Hardware-accelerated Rendering. PhD thesis, Universität Erlangen-Nürnberg, 1999.

[Held97] *Martin Held.* ERIT: A Collection of Efficient and Reliable Intersection Tests. *Journal of Graphics Tools: JGT*, 2(4):25–44, 1997.

[Hennessy96] *John L. Hennessy and David A. Patterson. Computer Architecture – A Quantitative Approach, 2nd edition.* Morgan Kaufmann, 1996.

[Hoppe99] *Reiner Hoppe, Gerd Wölfle, and Friedlich M. Landstorfer.* A Fast and Enhanced Ray Optical Propagation Model for Indoor and Urban Scenarios based on an Intelligent Preprocessing of the Database. In *10th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Osaka, Japan F5-3, September 1999.

[Hsiung92] *Ping-Kang Hsiung and Robert H. Thibadeau.* Accelerating ARTS. *The Visual Computer*, 8(3):181–190, March 1992.

[Humphreys96] *Greg Humphreys and C. Scott Ananian.* TigerSHARK: A Hardware Accelerated Ray-tracing Engine. Technical report, Princeton University, 1996.

[Humphreys02] *Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski.* Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):693–702, July 2002.

[Hurley02] *James T. Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov.* Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of Graphicon*, 2002. Available from http://www.graphicon.ru/-2002/papers.html.

[Intel] Intel Corp. *Intel Computer Based Tutorial.* http://-developer.intel.com/vtune/cbts/cbts.htm.

[Intel01] Intel Corp. *IA-32 Intel Architecture Optimization – Reference Manual*, 2001.

[Intel02a] Intel Corp. *Intel C/C++ Compilers*, 2002. http://www.intel.com/software/products/compilers.

[Intel02b] *Intel Corp.* Intel Pentium III Streaming SIMD Extensions. http://developer.intel.com/vtune/cbts/simd.htm, 2002.

[Intel02c] *Intel Corp.* Introduction to Hyper-Threading Technology. http://developer.intel.com/technology/-hyperthread, 2002.

[Jensen95] *Henrik Wann Jensen and Niels Jorgen Christensen.* Efficiently Rendering Shadows using the Photon Map. In H. Santo, editor, *Edugraphics + Compugraphics Proceedings*, pages 285–291. GRASP- Graphic Science Promotions & Publications, 1995.

[Jensen96] *Henrik Wann Jensen.* Global Illumination using Photon Maps. *Rendering Techniques 1996*, pages 21–30, 1996. (Proceedings of the 7th Eurographics Workshop on Rendering).

[Jensen97] *Henrik Wann Jensen.* Rendering Caustics on Non-Lambertian Surfaces. *Computer Graphics Forum*, 16(1):57–64, 1997.

[Jensen01] *Henrik Wann Jensen. Realistic Image Synthesis Using Photon Mapping.* A K Peters Ltd, 2001. ISBN 1-56881-147-0.

[Jevans89] *David Jevans and Brian Wyvill.* Adaptive Voxel Subdivision for Ray Tracing. *Proceedings of Graphics Interface '89*, pages 164–172, June 1989.

[Kajiya86] *James T. Kajiya.* The Rendering Equation. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 143–150, 1986.

[Kautz03] *Jan Kautz.* Realistic, Real-Time Shading and Rendering of Objects with Complex Materials. PhD thesis, Saarland University (Max-Planck-Institut fuer Informatik), Saarbrücken, Germany, 2003.

[Kay86] *Timothy L. Kay and James T. Kajiya.* Ray Tracing Complex Scenes. *Computer Graphics (Proceedings of SIGGRAPH 86)*, 20(4):269–278, June 1986. Held in Dallas, Texas.

[Keates95] *Martin J. Keates and Roger J. Hubbold.* Interactive Ray Tracing on a Virtual Shared-Memory Parallel Computer. *Computer Graphics Forum*, 14(4):189–202, October 1995.

[Kedem84] *G. Kedem and J. L. Ellis.* The Raycasting Machine. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers ICCD '84*, pages 533–538. IEEE Computer Society Pess, 1984.

[Keller97] *Alexander Keller.* Instant Radiosity. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 49–56, 1997.

[Keller98] *Alexander Keller.* Quasi-Monte Carlo Methods for Realistic Image Synthesis. PhD thesis, University of Kaiserslautern, 1998.

[Keller00] *Alexander Keller and Ingo Wald.* Efficient Importance Sampling Techniques for the Photon Map. In *Vision, Modelling, and Visualization (VMV) 2000*, pages 271–279, November 2000.

[Keller01] *Alexander Keller and Wolfgang Heidrich.* Interleaved Sampling. *Rendering Techniques 2001*, pages 269–276, 2001. (Proceedings of the 12th Eurographics Workshop on Rendering).

[Keller03] *Alexander Keller.* Monte Carlo & Beyond - Course Material. Technical Report 320/02, University of Kaiserslautern, 2003. Published in Eurographics 2003 Tutorial Notes.

[Kessenich02] *John Kessenich, Dave Baldwin, and Randi Rost.* The OpenGL Shading Language, Version 1.051, February 2002. Available from http://www.3dlabs.com/support/-developer/ogl2/downloads/ShaderSpecV1.051.pdf.

[Kirk91] *David Kirk and James Arvo.* Improved Ray Tagging For Voxel-Based Ray Tracing. In James Arvo, editor, *Graphics Gems II*, pages 264–266. Academic Press, 1991.

[Kirk92] David Kirk, editor. *Graphics Gems III.* Academic Press, 1992. ISBN: 0124096735.

[Klimaszewski97] *Kryzsztof S. Klimaszewski and Thomas W. Sederberg.* Faster Ray Tracing using Adaptive Grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, January/ February 1997.

[Kolb95] *Craig Kolb, Don Mitchell, and Pat Hanrahan.* A Realistic Camera Model for Computer Graphics. *Computer Graphics*, 29(Annual Conference Series):317–324, November 1995.

[Kollig02] *Thomas Kollig and Alexander Keller.* Efficient Multidimensional Sampling. *Computer Graphics Forum*, 21(3):557–563, 2002. (Proceedings of Eurographics 2002).

[Krüger03] *Jens Krüger and Rüdiger Westermann.* Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 22(3), July 2003.

[Lafortune93] *Eric Lafortune and Yves Willems.* Bidirectional Path Tracing. In *Proc. 3rd International Conference on Computational Graphics and Visualization Techniques (Compugraphics)*, pages 145–153, 1993.

[Lafortune96] *Eric Lafortune.* Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering. PhD thesis, Katholieke Universitiet Leuven, Belgium, 1996.

[Larsen01] *E. Scott Larsen and David McAllister.* Fast Matrix Multiplies using Graphics Hardware. In *Supercomputing 2001*, pages 55–60, 2001.

[Larson98] *Greg Ward Larson.* The Holodeck: A Parallel Ray-Caching Rendering System. *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, 1998.

[Leray87] *Pascal Leray.* Towards a Z-Buffer and Ray-Tracing Multimode System Based on Parallel Architecture and VLSI Chips. In Wolfgang Straßer, editor, *Advances in Computer Graphics Hardware I*, pages 141–145. Springer Verlag, 1987.

[Levoy96] *Marc Levoy and Pat Hanrahan.* Light Field Rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH)*, pages 31–42. ACM Press, 1996.

[Lext00] *Jonas Lext, Ulf Assarsson, and Tomas Möller.* BART: A Benchmark for Animated Ray Tracing. Technical report, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, May 2000. Available at http://www.ce.chalmers.se/BART/.

[Lext01] *Jonas Lext and Tomas Akenine-Möller.* Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations*, pages 311–318, 2001.

[Lischinski92] *Dani Lischinski, Filippo Tampieri, and Donald P. GreenBerg.* Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics and Applications*, 12(6):25–39, November 1992.

[Lischinski93] *Dani Lischinski, Filippo Tampieri, and Donald P. Greenberg.* Combining Hierarchical Radiosity and Discontinuity Meshing. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH)*, pages 199–208. ACM Press, 1993.

[MacDonald89] *J. David MacDonald and Kellogg S. Booth.* Heuristics for Ray Tracing using Space Subdivision. In *Proceedings of Graphics Interface '89*, pages 152–63, Toronto, Ontario, June 1989. Canadian Information Processing Society.

[MacDonald90] *J. David MacDonald and Kellogg S. Booth.* Heuristics for Ray Tracing using Space Subdivision. *Visual Computer*, 6(6):153–65, 1990.

[Mai00] *Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz.* Smart Memories: A Modular Recongurable Architecture. *IEEE International Symposium on Computer Architecture*, pages 161–171, 2000.

[Mark01] *William Mark.* Shading System Immediate-Mode API, v2.1. In *SIGGRAPH 2001 Course 24 Notes – Real-Time Shading*, August 2001.

[Mark03] *William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard.* Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 22(3):896–907, 2003.

[McKown91] *John W. McKown and R.Lee Hamilton Jr.* Ray Tracing as a Design Tool for Radio Networks. *IEEE Network Magazine*, November 1991.

[Meissner98] *M. Meissner, U. Kanus, and W. Strasser.* VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *Eurographics/ACM SIGGRAPH Workshop on Graphics Hardware*, 1998.

[Meißner02] *M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa.* VIZARD II: A Reconfigurable Interactive Volume Rendering System. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 137–146. Eurographics Association, 2002.

[Meneveaux03] *Daniel Meneveaux, Kadi Bouatouch, Gilles Subrenat, and Philippe Blasi.* Efficient Clustering and Visibility Calculation for Global Illumination. *AFRIGRAPH 2003*, pages 87–94, 2003.

[Möller] *Tomas Möller.* Practical Analysis of Optimized Ray-Triangle Intersection. http://www.ce.chalmers.se/staff/-tomasm/raytri/.

[Möller97] *Tomas Möller and Ben Trumbore.* Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[Muuss95a] *Michael J. Muuss.* Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium '95*, June 1995.

[Muuss95b] *Michael J. Muuss and Maximo Lorenzo.* High-Resolution Interactive Multispectral Missile Sensor Simulation for ATR and DIS. In *Proceedings of BRL-CAD Symposium '95*, June 1995.

[Myszkowski01] *Karol Myszkowski, Takehiro Tawara, Hiroyuki Akamine, and Hans-Peter Seidel.* Perception-Guided Global Illumination Solution for Animation Rendering. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques (ACM SIGGRAPH)*, pages 221–230. ACM Press, 2001.

[Neider93] *Jackie Neider, Tom Davis, and Mason Woo. OpenGL Programming Guide.* Addison-Wesley, 1993. ISBN 0-20163-274-8.

[NVidia01] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, March 2001. Available from http://www.nvidia.com.

[Olano98] *Marc Olano and Anselmo Lastra.* A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques (ACM SIGGRAPH)*, pages 159–168. ACM Press, July 1998.

[OpenSG01] *OpenSG-Forum.* http://www.opensg.org, 2001.

[OSG] OpenSceneGraph. http://www.openscenegraph.org.

[Parker98a] *Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan.* Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98*, pages 233–238, October 1998.

[Parker98b] *Steven Parker, Peter Shirley, and Brian Smits.* Single Sample Soft Shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, October 1998. Available at http://www.cs.-utah.edu/∼bes/papers/coneShadow.

[Parker99a] *Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Chuck Hansen, and Peter Shirley.* Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Computer Graphics and Visualization*, 5(3):238–250, July-September 1999.

[Parker99b] *Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan.* Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics (I3D)*, pages 119–126, April 1999.

[Paul02] *Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig.* Real PRAM-Programming. In *Proceedings of EuroPar 2002*, 2002. Jörg Fischer is the same person as Jörg Schmittler.

[PCI Express] PCI Express Specifications. http://www.pcisig.com/-specifications/pciexpress/.

[PCI-X] PCI-X 2.0 Specifications. http://www.pcisig.com/-pcix_20/#overview.

[Peercy00] *Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive Multi-Pass Programmable Shading.* ACM Press/Addison-Wesley Publishing Co., New Orleans, USA, July 2000.

[Pfister99] *Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler.* The VolumePro real-time raycasting system. *Computer Graphics*, 33(Annual Conference Series):251–260, 1999.

[Pfister01] *Hanspeter Pfister.* RAYA – A Ray Tracing Architecture for Volumes and Polygons. SIGGRAPH course on Interactive Ray Tracing, 2001.

[Pharr97] *Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan.* Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, August 1997.

[Pixar89] *Pixar. The RenderMan Interface.* San Rafael, September 1989.

[Plasi97] *Philippe Plasi, Betrant Le Saëc, and Gèrad Vignoles.* Application of Rendering Techniques to Monte-Carlo Physical Simulation of Gas Diffusion. In Julie Dorsey and Philipp Slusallek, editors, *Rendering Techniques '97*, pages 297–308. Springer, 1997.

[Pomi03] *Andreas Pomi, Gerd Marmitt, Ingo Wald, and Philipp Slusallek.* Streaming Video Textures for Mixed Reality Applications in Interactive Ray Tracing Environments. In *Proceedings of Virtual Reality, Modelling and Visualization (VMV)*, 2003.

[Proudfoot01] *Kekoa Proudfoot, William Mark, Svetoslav Tzvetkov, and Pat Hanrahan.* A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of the 28th annual conference on Computer* *graphics and interactive techniques (ACM SIGGRAPH)*, pages 159–170. ACM Press, August 2001.

[Pulleyblank87] *Ron Pulleyblank and John Kapenga.* The Feasibility of a VLSI Chip for Ray Tracing Bicubic Patches. *IEEE Computer Graphics and Applications*, 7(3):33–44, March 1987.

[Purcell02] *Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan.* Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002. (Proceedings of SIGGRAPH 2002).

[Purcell03] *Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan.* Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[Reinhard95] *Erik Reinhard.* Scheduling and Data Management for Parallel Ray Tracing. PhD thesis, University of East Anglia, 1995.

[Reinhard00] *Erik Reinhard, Brian Smits, and Chuck Hansen.* Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering*, pages 299–306, Brno, Czech Republic, June 2000.

[Rohlf94] *John Rohlf and James Helman.* IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics*, 28(Annual Conference Series):381–394, July 1994.

[Rubin80] *Steve M. Rubin and Turner Whitted.* A Three-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[Samet89] *Hanan Samet.* Implementing Ray Tracing with Octrees and Neighbor Finding. *Computers and Graphics*, 13(4):445–60, 1989.

[Sbert97] *Mateu Sbert.* The Use of Global Random Directions to Compute Radiosity. Global Monte Carlo Techniques. PhD thesis, University of Catalonia, Spain, 1997.

[Schachter83] *Bruce Schachter. Computer Image Generation.* John Wiley and Sons, New York, 1983.

[Schmittler02] *Jörg Schmittler, Ingo Wald, and Philipp Slusallek.* SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 27–36, 2002.

[Schmittler03] *Jörg Schmittler, Alexander Leidinger, and Philipp Slusallek.* A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *Computer and Graphics, Volume 27, Graphics Hardware*, pages 693–699, 2003.

[Shirley96] *Peter Shirley, Changyaw Wang, and Kurt Zimmerman.* Monte Carlo Techniques for Direct Lighting Calculations. *ACM Transactions on Graphics*, 15(1):1–36, 1996.

[Shirley02] *Peter Shirley. Fundamentals of Computer Graphics.* A K Peters Ltd, 2002. ISBN 1-56881-124-1.

[Shirley03] *Peter Shirley and R. Keith Morley. Realistic Ray Tracing.* A K Peters Ltd, Second edition, 2003. ISBN 1-56881-198-5.

[Shoemake98] *Ken Shoemake.* Pluecker Coordinate Tutorial. *Ray Tracing News*, 1998. http://www.acm.org/tog/resources/-RTNews/html/rtnv11n1.html#art3.

[Sillion94] *François X. Sillion and Claude Puech. Radiosity and Global Illumination.* Morgan Kaufmann Publishers, 1994. ISBN: 1558602771.

[Simiakakis95] *George Simiakakis.* Accelerating Ray Tracing with Directional Subdivision and Parallel Processing. PhD thesis, University of East Anglia, 1995.

[Simmons00] *Maryann Simmons and Carlo H. Sequin.* Tapestry: A Dynamic Mesh-based Display Representation for Interactive Rendering, June 2000. ISBN 3-211-83535-0.

[Slusallek95] *Philipp Slusallek, Thomas Pflaum, and Hans-Peter Seidel.* Using Procedural RenderMan Shaders for Global Illumination. In *Computer Graphics Forum (Proc. of Eurographics '95*, pages 311–324, 1995.

[Smits94] *Brian Smits, James Arvo, and Donald Greenberg.* A Clustering Algorithm for Radiosity in Complex Environments. In *Proceedings of the 21st annual Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH)*, pages 435–442. ACM Press, 1994.

[Smits98] *Brian Smits.* Efficiency Issues for Ray Tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998.

[Spackman91] *John Spackman and Philip Willis.* The SMART Navigation of a Ray Through an Oct-Tree. *Computers and Graphics*, 15(2):185–194, 1991.

[Stamminger99] *Marc Stamminger. Finite Element Methods for Global Illumination Computations.* Herbert Utz Verlag, 1999. ISBN: 3896756613.

[Stamminger00] *Marc Stamminger, Jörg Haber, Hartmut Schirmacher, and Hans-Peter Seidel.* Walkthroughs with Corrective Textures. In *Proceedings of the 11th Eurographics Workshop on Rendering*, pages 377–388, 2000.

[Stevens98] *W. Richard Stevens. Unix Network Programming Volume 1.* Prentice Hall, 1998.

[Subramanian90a] *K. R. Subramanian.* A Search Structure based on K-d Trees for Efficient Ray Tracing. PhD thesis, The University of Texas at Austin, December 1990.

[Subramanian90b] *K. R. Subramanian and Donald S. Fussel.* Factors Affecting Performance of Ray Tracing Hierarchies. Technical Report Tx 78712, The University of Texas at Austin, July 1990.

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

[Sung92] *Kelvin Sung and Peter Shirley.* Ray Tracing with the BSP Tree. In David Kirk, editor, *Graphics Gems III*, pages 271—274. Academic Press, 1992. ISBN: 0124096735.

[Szirmay-Kalos98] *Laszlo Szirmay-Kalos and Gabor Márton.* Worst-Case versus Average Case Complexity of Ray-Shooting. *Computing*, 61(2):103–131, 1998.

[Teller98] *Seth Teller and John Alex.* Frustum Casting for Progressive, Interactive Rendering. Technical Report MIT LCS TR-740, MIT, January 1998.

[Tole02] *Parag Tole, Fabio Pellacini, Bruce Walter, and Donald P. Greenberg.* Interactive Global Illumination in Dynamic Scenes. *ACM Transactions on Graphics*, 21(3):537–546, 2002. (Proceedings of ACM SIGGRAPH 2002).

[Ullman03] *Thomas Ullman, Thomas Preidel, and Beat Brüderlin.* Efficient Sampling of Textured Scenes in Vertex Tracing. In *Proceedings of Eurographics Conference (Short Presentations)*, 2003.

[Ullmann01a] *Thomas Ullmann, Alexander Schmidt, Daniel Beier, and Beat Brüderlin.* Adaptive Progressive Vertex Tracing for Interactive Reflections. *Computer Graphics Forum (Proceedings of Eurographics)*, 20(3), September 2001.

[Ullmann01b] *Thomas Ullmann, Alexander Schmidt, Daniel Beier, and Beat Brüderlin.* Adaptive Progressive Vertex Tracing in Distributed Environments. In *Proceedings of the Ninth Pacific Conference on Computer Graphics and Applications (Pacific Graphics 2001)*, pages 285–294. IEEE, October 2001.

[Upstill90] *Steve Upstill. The RenderMan Companion.* Addison-Wesley, 1990.

[Veach94] *Eric Veach and Leonid Guibas.* Bidirectional Estimators for Light Transport. In *Proceedings of the 5th Eurographics Worshop on Rendering*, pages 147 – 161, Darmstadt, Germany, June 1994.

[Veach97] *Eric Veach.* Robust Monte Carlo Methods for Light Transport Simulation. PhD thesis, Stanford University, 1997.

[Wagner02] *Markus Wagner.* Development of a Ray-Tracing-Based VRML Browser and Editor. Master's thesis, Computer Graphics Group, Saarland University, Saarbrücken, Germany, 2002.

[Wald] *Ingo Wald and Tim Dahmen.* *OpenRT User Manual.* Computer Graphics Group, Saarland University. http://www.openrt.de.

[Wald01a] *Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek.* Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics 2001).

[Wald01b] *Ingo Wald and Philipp Slusallek.* State-of-the-Art in Interactive Ray-Tracing. In *State of the Art Reports, Eurographics 2001*, pages 21–42, 2001.

[Wald01c] *Ingo Wald, Philipp Slusallek, and Carsten Benthin.* Interactive Distributed Ray Tracing of Highly Complex Models. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques*, Proceedings of the 12th Eurographics Workshop on Rendering Techniques, London, UK, June 25-27, 2001, pages 274–285. Springer, 2001.

[Wald02a] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications.

[Wald02b] *Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek.* Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques 2002*, pages 15–24, 2002. (Proceedings of the 13th Eurographics Workshop on Rendering).

[Wald03a] *Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek.* Interactive Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, Klagenfurt, Austria, August 2003. Springer.

[Wald03b] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[Wald03c] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* Interactive Global Illumination in Complex and Highly Occluded Environments. In Per H Christensen and Daniel Cohen-Or, editors, *Proceedings of the 2003 Eurographics Symposium on Rendering*, pages 74–81, Leuven, Belgium, June 2003.

[Wald03d] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* Towards Realtime Ray Tracing – Issues and Potential. Technical report, Saarland University, 2003.

[Wald03e] *Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek.* Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*. Eurographics, 2003.

[Wallace87] *John R. Wallace, Michael F. Cohen, and Donald P. Greenberg.* A Two-pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods. *Computer Graphics*, 21(4):311–320, 1987.

[Walter99] *Bruce Walter, George Drettakis, and Steven Parker.* Interactive Rendering using the Render Cache. In *Rendering Techniques 1999 (Proceedings of Eurographics Workshop on Rendering)*, 1999.

[Walter02] *Bruce Walter, George Drettakis, and Donald P. Greenberg.* Enhancing and Optimizing the Render Cache. In *Rendering Techniques 2002 (Proceedings of Eurographics Workshop on Rendering)*, 2002.

[Ward91] *Gregory Ward.* Adaptive Shadow Testing for Ray Tracing. In *2nd Eurographics Workshop on Rendering*, 1991.

[Ward92] *Gregory J. Ward and Paul Heckbert.* Irradiance Gradients. In *Third Eurographics Workshop on Rendering*, pages 85–98, Bristol, UK, 1992.

[Ward99] *Gregory Ward and Maryann Simmons.* The Holodeck Ray Cache: An Interactive Rendering System for Global Illumination in Nondiffuse Environments. *ACM Transactions on Graphics*, 18(4):361–398, October 1999.

[Wernecke94] *Josie Wernecke. The Inventor Mentor.* Addison-Wesley, 1994. ISBN 0-20162-495-8.

[Whang95] *K. Y. Whang, J. W. Song, J. W. Chang, J. Y. Kim, W. S. Cho, C. M. Park, and I. Y. Song.* Octree-R: An Adaptive Octree for efficient Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, December 1995. ISSN 1077-2626.

[Whitted80] *Turner Whitted.* An Improved Illumination Model for Shaded Display. *CACM*, 23(6):343–349, June 1980.

[Wilson93] *Tom Wilson.* Ray Tracing Abstracts Survey, 1993. available from ftp://nic.funet.fi/pub/sci/papers/graphics/-rtabs.4.93.shar.Z.

[Woo90] *Andrew Woo.* Fast Ray-Polygon Intersection. In Andrew S. Glassner, editor, *Graphics Gems*, page 394. Academic Press, San Diego, 1990.