Acceleration Methods for Ray Tracing based Global Illumination

Dissertation

zur Erlangung des Doktorgrades
Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

vorgelegt von

Holger Dammertz
aus München

Institut für Medieninformatik, 2011

Amtierender Dekan:	Prof. Dr.-Ing. Klaus Dietmayer, Ulm University, Germany
Gutachter:	Dr. rer. nat. Alexander Keller, mental images, Berlin
Gutachter:	Prof. Dr.-Ing. Hendrik P. A. Lensch, Ulm University, Germany
Externer Gutachter:	Peter Shirley, Adjunct Professor, Ph.D. University of Utah, Salt Lake City, UT, USA
Tag der Promotion:	19.04.2011

Abstract

The generation of photorealistic images is one of the major topics in computer graphics. By using the principles of physical light propagation, images that are indistinguishable from real photographs can be generated. This computation, however, is a very time-consuming task. When simulating the real behavior of light, individual images can take hours to be of sufficient quality. For this reason movie production has relied on artist driven methods to generate life-like images. Only recently there has been a convergence of techniques from physically based simulation and movie production that allowed the use of these techniques in a production environment. In this thesis we advocate this convergence and develop novel algorithms to solve the problems of computing high quality photo-realistic images for complex scenes. We investigate and extend the algorithms that are used to generate such images on a computer and we contribute novel techniques that allow to perform the necessary ray-tracing based computations faster. We consider the whole image generation pipeline starting with the low level fundamentals of fast ray tracing on modern computer architectures up to algorithmic improvements on the highest level of full photo-realistic image generation systems.

In particular, we develop a novel multi-branching acceleration structure for high performance ray tracing and extend the underlying data structures by software caching to further accelerate the result. We also provide a detailed analysis on the factors that influence ray tracing speed and derive strategies for significant improvements. These create the foundations for the development of a production quality global illumination rendering system. We present the system and develop several techniques that make it usable for realistic applications. The key aspect in this system is a path space partitioning method to allow for efficient handling of highly complex illumination situations as well as the handling of complex material properties. We further develop methods to improve the illumination computation from environment maps and provide a filtering technique for preview applications.

Acknowledgments

Many thanks to

- Alexander Keller for his guidance, support, and his passion for computer graphics
- Johannes Hanika for a fun and highly productive research cooperation
- Hendrik P. A. Lensch for his feedback, support and for providing the environment to finish this Ph.D. thesis
- the old and new computer graphics group
- mental images for the financial support
- Carsten Wächter, Matthias Raab, Leonhard Grünschloss, and Daniel Seibert for proofreading and many rendering related discussions
- my wife and family

Contents

1 Introduction

One of the strongest driving forces in computer graphics has been the strive to reproduce reality. Starting with the beginning of graphical computer displays, scientists and artists alike pushed the boundaries of what is possible. Over the years, computer generated images were able to approximate the appearance and phenomena of the real world better and better. While in the beginning only primitive shapes, a limited color space, and simple algorithms could be used to approximate reality, the huge advancements in computing power over the last 20 years as well as significant improvements in the fundamental algorithms allow us to now generate images and movies entirely in the computer. The results are almost impossible to distinguish from real photographs or live footage.

Even though high quality computer generated images are now a reality there is a large area of unsolved problems and a lot of room for improvement. The simulation of the phenomena from nature is a highly complex and compute-intensive task. Especially for the production of movies, where thousands of individual images are needed, speed as well as quality is of paramount importance. The goal of increased realism is tightly coupled with increasing complexity. While in the beginning of computer graphics the realistic rendering of a single object was an enormous task, nowadays whole cities and forests are modeled in a computer and rendered into realistic images.

1.1 Physically Based Rendering and Visual Effects

Physically based rendering is a sub-class of graphics algorithms that use real physical quantities and models to not only produce visually pleasing but also quantitatively correct images. The correctness of the image is of course only valid within the constraints of the physical model that is used but nevertheless allows to produce results that can be measured in the real world. This approach is in stark contrast to artist driven methods that are used for real time computer graphics as well as for movie production. While the visual effects in movies look strikingly realistic since over a decade now, their rendering methods do not rely on realistic physical simulations but on handcrafted models and artist tweaked parameters to achieve the desired results. Only recently this has begun to change and the visual effects industry as well as the real time graphics community have realized the advantage realistic models can provide over manual crafting. Even though one can always argue that artistic freedom is important for visual effects and real time graphics, the advantages of being able to simulate light and surfaces as they behave in the real world cannot be neglected. In this thesis we advocate this convergence and present novel algorithms that improve the robustness as well as the speed of physical light transport simulation.

1.2 Thesis Overview

One very general way to simulate the distribution of light is by tracing individual light rays through the scene and modeling their behavior like reflection or refraction using physical models. This basic operation is realized by an algorithm called ray tracing and is one of the fundamental building blocks that we use in this thesis. Based on this foundation, we will develop novel acceleration methods for fast incoherent ray tracing and several new techniques and improvements for physically based rendering algorithms. Our ray tracing algorithm acceleration methods directly benefit all simulation algorithms developed in this thesis and we show their effectiveness in a complete rendering system that uses all our results. Some of the algorithms and results presented in this thesis have already been published at conferences and in different journals [Damm 06, Damm 08a, Damm 08b, Bend 08, Damm 09a, Damm 09b, Damm 10a, Damm 10b]. We present this research in a unified framework and give additional results and discussions. We will also extend the work at several points and present new techniques and insights.

The remainder of this thesis is organized as follows. In Chapter 2 we will review the fundamentals of image generation and discuss the basic framework of a rendering system. We also discuss some fundamentals of processor architectures that are required for high performance rendering applications. Chapter 3 develops our novel data structure for accelerated ray tracing specifically optimized for physically based rendering problems and Chapter 4 further improves this data structure by considering the representation of the 3d scenes and the hierarchy construction algorithm. In Chapter 5 we will consider software caching for improving the ray tracing performance in light transport simulations.

Chapter 6 then introduces a full-featured, general global illumination system for production quality image generation. This system is built on top of the data structures and acceleration techniques developed in the previous chapters and addresses many of the problems encountered in image generation for visual effects with physically based rendering systems. In this chapter we will contribute several techniques for global illumination computation like an environment map sampling technique and an image-space filter for interactive preview. Finally, in Chapter 7 we will summarize the developed techniques and discuss their relationships.

1.2.1 Summary of Contributions

To summarize, the main contributions in this thesis are

- a novel bounding volume hierarchy for acceleration of incoherent rays. This acceleration structure is targeted towards modern hardware architectures, performs significantly faster than previously used structures, and requires less memory than previous methods.
- an efficient triangle splitting heuristic that improves the quality of ray tracing acceleration structures in many situations.
- an iterative ray-triangle intersection algorithm that guarantees that no intersections are missed. This is often needed in physical simulations where accuracy is paramount.
- a discussion of software caching for ray tracing acceleration with a branch-free parallel triangle intersection algorithm and a novel shadow cache that can greatly improve the efficiency of visibility queries.
- a global iterative stopping criterion for Monte Carlo light transport simulation based on integro-approximation.
- a path space partitioning method that, contrary to previous algorithms, solves the full light transport equation by several distinct algorithms. With this method we develop a production quality global illumination renderer that has less restrictions than previous methods (for example allowing for non-physical BRDFs).
- an environment map sampling technique that allows to start light paths with correct Monte Carlo weights.
- an image-space filtering technique for noisy Monte Carlo images that allows for fully interactive real-time preview of illumination and material changes.
- a discussion of on-demand spectral rendering with application examples in fluorescent concentrator simulation and a spectral camera lens simulation.

2
Image Generation Fundamentals

In this Chapter we will provide the fundamentals of the image generation pipeline and basic architecture considerations that are needed to understand the rest of this thesis.

2.1 Image Generation by Camera Simulation

It is our goal to create an efficient computer program to generate synthetic images that are indistinguishable from real images. One point of view to explain this process is by considering how real images are acquired into a computer processable format by a camera. The computer program now tries to replicate this process as closely as possible to generate a final image.

Raster Images

A computer generated image is stored as an array of color values. Each color value represents usually a square in the image region called a pixel (short for picture element) but the square layout is not necessary [Damm 09c]. Most computer generated images are displayed on computer monitors that have three color components per pixel called RGB. They correspond to the red, green and blue channel, each of which has a resolution of an 8 bit integer (values between 0 and 255). There exist other standards and, especially for print, other representations but in this thesis we will use the 8 bit per channel RGB representation with square pixels as final output. Figure 2.1 illustrates the data layout and representation.

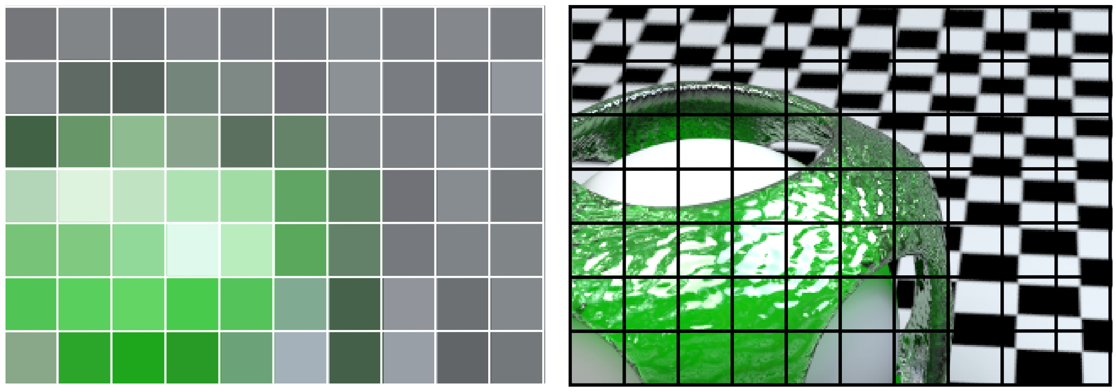


Figure 2.1: Illustration of a pixel raster and an image stored in the raster. The left figure shows a pixel raster of the size 10×7. The right figure shows the same image on a pixel raster with the resolution 320×224 pixels. As an overlay the raster of the left image is shown.

Dynamic Range vs. Tone Mapping

In nature, brightness differences have a very high dynamic range. Most commonly used display technologies cannot reproduce this. They provide usually only 8 bits per channel for the final output. Nevertheless, internal representations during image synthesis need a higher resolution and range. Throughout this thesis we will use the RGB format, but with 32 bit floating point numbers per channel if not otherwise noted. A short overview of floating point numbers can be found in Section 2.4. This allows to perform all computations directly inside the image buffer without losing too much accuracy. This is necessary for high quality photo-realistic image synthesis. The representation using floating point numbers is sometimes called high dynamic range representation (HDR) as it allows for huge value differences using the exponent of the float representation. Even though there exist approaches to directly display images with a higher dynamic range [Seet 04], for most display devices the dynamic range of the image has to be mapped to a smaller displayable range. This is performed with tone mapping operations. In Chapter 22 of [Shir 02] an overview of tone reproduction for computer graphics can be found. A commonly used tone-mapping operator is described by Reinhard in [Rein 02]. A comparison of five recent operators can be found in [Ashi 06].

The simplest operator to use is by normalizing the luminance of the color value:

$$L_{new} = \frac{L}{L+1}$$

The luminance is computed by transforming the RGB value into a color-space with a luminance channel like HSV.

For combining computer generated images with real world images in movie production a high dynamic range in the stored output image is also very important. To achieve realistic results the response curve of the film has to be modeled and used to adjust the computer generated image. The closer the simulation is to real world parameters the easier it becomes to adjust the computer generated images.

Ray Tracing

Ray tracing is a fundamental technique to create computer generated images. It was first developed in 1968 by Appel in [Appe 68] as a method to add shadows to a computer generated image. Since then a lot of research started on using ray tracing for computer graphics and image generation.

In its basic form ray tracing computes computer generated images by computing a color value for each pixel independently of all the other pixels. To generate an image several components are needed. First is a camera that defines the viewer (or eye) position in the scene and also the viewing direction. Coupled with the camera is the image plane that contains the pixel elements. A ray tracer now provides a function $trace(x, d)$

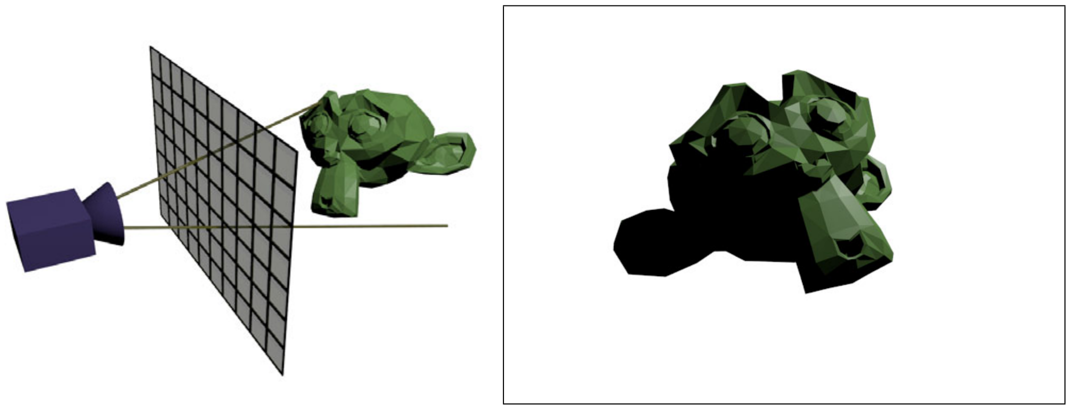


Figure 2.2: This figure illustrates how ray tracing is used to generated an image from a 3d scene. Rays are generated from the camera position through each pixel of the image plane and intersected with the 3d scene. This is shown in the left figure. Rays that intersect the object give the pixel the according color. The final output image is shown on the right.

with a 3d position x as origin and a direction d and returns the closest intersection of this ray with the scene. To generate a simple image with this function for each pixel a ray is generated that originates at the camera origin (eye position) and has a direction through each pixel. The pixel is then shaded with the object color at the intersection point. This process is illustrated in Figure 2.2. These rays that compute the nearest intersection point seen through each pixel are called primary rays or eye rays. Of course computing such a simple image with ray tracing is very inefficient as much faster methods exist to compute primary intersections. The most prominent algorithm is rasterization [Fole 94]. The power of ray tracing lies in its generality. The $trace$ function is defined for arbitrary x and d and thus allows for complex computations using only this simple operation. The application to general light transport simulation is discussed in Section 2.2 and we will use it in Chapter 6 as a fundamental building block in our rendering system.

A 3d scene is modeled by using geometric primitives. For a generic ray tracer one can use any primitive that allows to compute the intersection of a ray segment with this object. Primitives that have a simple implicit representation are the ones that result in the most simple intersection algorithms. A common example is a sphere where the intersection computation can be performed by inserting the ray equation $r(t) = \mathbf{o} + t\mathbf{d}$ into the implicit sphere equation $S : |(x - p)| = r$ and solving for t. The closest of the two possible solutions is the result of the $trace$ function. Other simple primitives are for example an infinite plane or an infinite cylinder. More complex objects may need more complex intersection algorithms. In [Shir 00] many ray object intersection algorithms can be found. In Section 2.3.3 we will discuss the most common object representation used for high performance ray tracing.

2.2 Light Transport Simulation by Path Tracing

The basic idea of light transport simulation with the ray tracing algorithm is to identify light transport paths that connect a light source with the image sensor. The contribution of each path is summed up in each pixel element of the sensor. In [Whit 80], Whitted introduced a simple ray tracing based image synthesis algorithm that is able to generate complex effects using a very simple recursive algorithm. It is one of the most intuitive variants of ray tracing based light transpo ne algorithm and shows a typical example image. In or

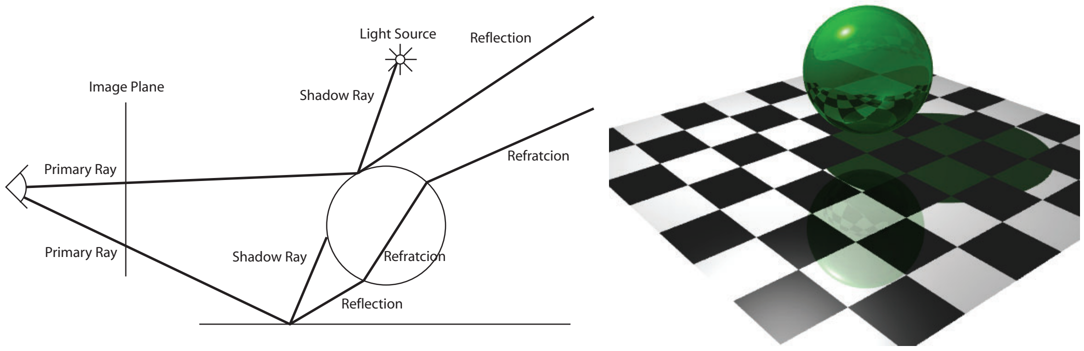


Figure 2.3: Whitted style or recursive ray tracing. The left image schematically shows the recursive extension of camera rays to simulate various effects. The right image shows a ray traced image with a reflective floor and a glass sphere.

only with the information of the hit point from the primary ray as in Figure 2.2, recursive ray tracing generates a path through the scene by recursively extending the ray at each hit point until a termination criterion is met. This allows to use for example the laws of reflection and refraction from optics to compute the appearance of a perfect glass sphere or mirror. Additionally, one can compute hard shadows from point light sources by connecting each hit point with a point light source and using the $trace$ function to compute the visibility. The intersection point is now only illuminated when it can receive light from the source.

Later in Section 6.1 we will more closely investigate the algorithms that are used for ray tracing based physical light transport simulation.

2.3 Rendering Software Architecture Design

From a software point of view implementing a renderer requires several important supporting components and algorithms to enable the generation of images from realistic scenes. Several choices can be made on the overall system structure and some are tightly related to the scene representation. The common surface representation is described in Section 2.3.3. In [Phar 04], the authors describe a full physically based rendering system in detail and develop a very flexible and extensible rendering system. This allows the authors to integrate many of the research results on physically based rendering systems. While this system is very flexible it sacrifices speed. In this thesis we instead follow a different approach. Instead of supporting all different kinds of geometry and possible algorithm combinations we identify the minimum needed functionality and implement it in a way that tightly couples with the fundamental ray tracing developed in Chapter 3. This allows to fully exploit the enhancements made to the ray tracer in all aspects of the rendering system and reduces the amount of glue code and additional data structures. In this section we shortly highlight some of the key design decisions. Many more details can be found in the later sections that implement our novel algorithms.

2.3.1 Enabling Physically Based Global Illumination in Movie Production

The use of computer generated images in movie production has a long history. It is used both for fully computer generated movies and for the combination with life action footage. A whole industry has built around the required techniques often summarized under the acronym CG VFX (computer graphics visual effects). This industry is highly artist driven and for a long time used a phenomenological approach to image generation. Widely used commercial products are Pixar's Renderman [Pixa 05] and mental images' mental ray [Drie 05]. Both of these products are in their fundamental design oriented towards artist driven workflow and have only limited support for physically based light transport. Surface appearance is described by huge programs called shaders that compute the final color locally without or with limited access to global information. Illumination is approximated by placing many invisible, colored light sources manually in the scene. To handle the complexity of images that need to be combined with real live footage the final image is not produced in a single rendering pass but computed in individual layers. This layered approach is difficult to handle with realistic global illumination algorithms as the information from one layer can't influence the other layers in a straight forward way. Another problem with full global illumination algorithms is the use of manually crafted shader networks for surface behavior description. Since they are crafted by artists using a purely visual approach they do not adhere to any mathematical theory or have physically plausible properties. This inhibits their use in a physically based rendering system. In Section 2.3.4 we will present in more detail the material model we use in this thesis and in Section 6.3 we will address the problem of complex shaders by developing a global illumination renderer that still converges to a unique and reasonable solution even if physically non-plausible shaders are used. Using a physically based renderer has the advantage of simplifying the lighting design done by artists as the behavior is like in the real world. Still it is usually necessary to provide at least a fall back to old shaders since the production pipeline of a full movie is rather complex and a complete change is difficult. Our system allows for a smoother transition in such a production environment.

2.3.2 Scene Description

The first design decision is how the scene data is transferred from a modeling application into the renderer. This description is tightly coupled with the abilities of the rendering system. An industry standard for general rendering (not physically based) is the RenderMan Interface [Pixa 05]. This document describes a postscript inspired language to describe 3d scenes and the parameters to produce movie quality images from them. It is an artist driven format that allows to tune and specify every parameter. In contrast to this a physically based system can use a simpler format since most of the complexity is in the rendering algorithms.

In our system a scene for the renderer consists of at most 4 files.

scene.ra2 the raw triangle data describing the surfaces in 3d

scene.[ran|ruv] additional information per triangle that stores vertex normals and uv coordinates. These files are optional and only created when needed.

scene.srs a text file containing all additional information needed to create an image. This are the materials, the cameras and additional environment information.

An important design decision in our system is that we support only one primitive to represent surfaces. Many rendering systems allow for different input primitives like spheres, subdivision surfaces, Bézier patches and arbitrary polygons that are either converted in the rendering system or directly used for rendering. This significantly increases the implementation complexity and inhibits many low level optimizations important for high performance. The primitive we use is the triangle. It is stored as a raw binary float file (with the extensions .ra2) where 9 floats represent a single triangle. This file format is also used in Chapter 3. It has the advantage of being loadable with a single line of code and it contains the data already in the format needed for the optimized ray tracer. We argue that all other surfaces that need to be supported in the rendering system can be converted already in the scene export process instead of in the rendering system. Our system supports for example subdivision surfaces but the code to handle them is in the exporter framework instead of the renderer. This, of course, increases the scene representation data on disk but this is not a real problem as the interchange format should not be the renderer representation but the original file format from the modeling application. Another restriction of this method is that now the exporter has to be executed again when parameters of the scene export, like subdivision level, change. All additional information needed for rendering are stored in separate files. Optional information that is often needed per vertex are texture coordinates and vertex normals. Both are stored in separate files in the same raw floating point data layout.

The scene.srs file contains all additional information to fully describe the scene to generate a final image. It is a complete specification of the global illumination problem. It is stored as a plain text file containing the following information:

Environment describes the background color or an optional HDR image for distant illumination. It can also contain the parameters to use an analytical sky model instead. We use the model described in [Pree 99].

Geometry is just the file-names of the .ra2 .ran and .ruv files.

Cameras contains a list of cameras with the parameters like lens setup, size, and also image output resolution.

Materials is a list of named materials with the necessary parameters and texture file names. A material may also emit. In this case it produces light sources. The material model that is used is described in detail in Section 2.3.4.

Material Associations is a simple list where material names are assigned to ranges of triangles.

Parameters for the renderer like algorithms to use and other global settings are assigned via command line parameters.

2.3.3 Boundary Representation

While an implementation of the $trace$ function could support many different object primitives that lend themselves nicely to object oriented design this comes at the cost of speed and a more complex scene description language. To achieve high speed in ray tracing only one primitive should be chosen. This allows one to develop highly optimized algorithms. In recent years it turned out that the triangle representation is a very good choice for a general ray tracer. Triangles need only 3 points in space to be described completely. They are always guaranteed to be flat (in contrast to arbitrary polygons) and can be used to approximate any surface. Many efficient ray intersection algorithms

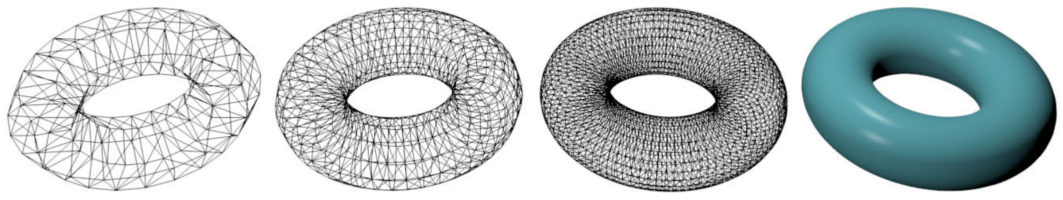


Figure 2.4: Illustration of the approximation of objects using triangles. The quality of the approximation depends on the number of triangles. From left to right the number of triangles in the approximating mesh increases.

exist for triangles and they have a simple to use surface parametrization. When an object is approximated with triangles the resulting object is often called a mesh. This approximation is shown for different numbers of triangles in Figure 2.4. Storing a full triangle mesh can be done in many different formats. Depending on application several additional information can be stored with the triangle information as is for example in the winged edge data structure the case [Fole 94]. In a full rendering system it must also be considered that many different algorithms may need to access the triangle data in different ways and storing multiple copies might not be an option for highly detailed scenes because of memory consumption. This is mostly important in real time applications where also physics simulation and sound simulation needs to access the scene data. For offline rendering applications there are usually less constraints.

Ray Triangle Intersection

The goal of ray triangle intersection computation is to first determine if a ray intersects the triangle and if this is true to compute the intersection point. Two basic approaches for ray triangle intersection tests exist. The first uses a direct 3d test using for example Plücker coordinates (see for example [Kin 97]) and computes ratios of volumes. In [Lofs 05] an automatic optimization is performed for these kind of intersection tests. The other method is to compute the intersection point of the ray with the plane the triangle lies in and then uses a test to determine if this point is inside the triangle or outside. [Lofs 05] also provides a comparison of various intersection tests. We use a variant of the triangle intersection test described by Möller et. al in [Moll 97]. This test falls into the second category. Our implementation that extends this test to intersect 4 triangles at once is described in Section 5.2.

Geometry Data Attributes and Texture Mapping

Besides the raw triangle information, additional data is required to compute a high quality computer generated image. Some of this data can be stored per object but often it is required to define varying data over a surface. This data can be stored in the vertices of the mesh and is then interpolated across the triangle using barycentric coordinates.

Examples for such information are vertex normal data to better approximate the smooth appearance of objects during shading computations or vertex colors to change the color of triangles. This data is often called vertex attributes.

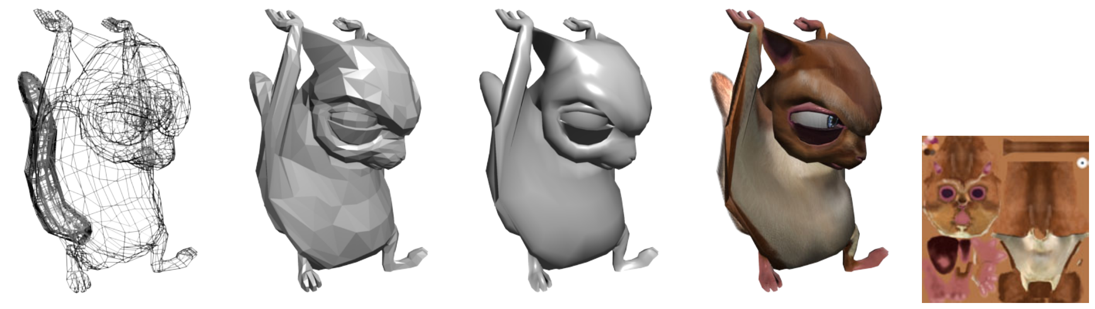


Figure 2.5: Illustration of interpolating information across the surface. The leftmost image shows the mesh of the object. Next a flat shaded version is shown. The third image is computed by using the same geometry but normals are stored per vertex and interpolated across each triangle. Finally the object is also colored by interpolating uv coordinates and accessing the texture map shown in the rightmost image. The model that is shown is from the Yo Frankie! project (c) copyright Blender Foundation.

Storing information per vertex allows only to generate smooth data variation across a single triangle. Texture mapping is a method of storing information on the surface each (discretized) surface point of an object. Usually a texture is a rectangular image array. The texture is mapped onto a surface using a surface parametrization. This parametrization maps parts of the texture onto the surface allowing to query the information at each surface point. An overview of texture mapping can be found in [Shir 02]. For texture mapping on a triangle mesh additional information per vertex is stored. This additional data is often called uv coordinates. The coordinates are a 2d point defining a position on the texture map. To get a uv coordinate of an arbitrary point x inside the triangle the barycentric coordinates of x are used to interpolate the uv coordinates from the three triangle vertices. Texture mapping and smooth normal interpolation is illustrated in Figure 2.5. As said before texture mapping is used to store arbitrary (but discrete) data on surfaces of objects. This data can be used to define color, reflectivity and any other information that might be needed by the rendering algorithm. It usually is used a a simple way of enhancing the detail of otherwise flat (triangle-) geometry.

Clip Mapping

Clip Mapping is a well known technique to efficiently increase detail on models without a huge increase in geometry and memory. It uses textures to define holes on a surface. In the simplest case the clip map texture just contains a 1 where the surface should be used and 0 where no surface should be. Figure 2.6 illustrates the use of a clip map on a trash can.

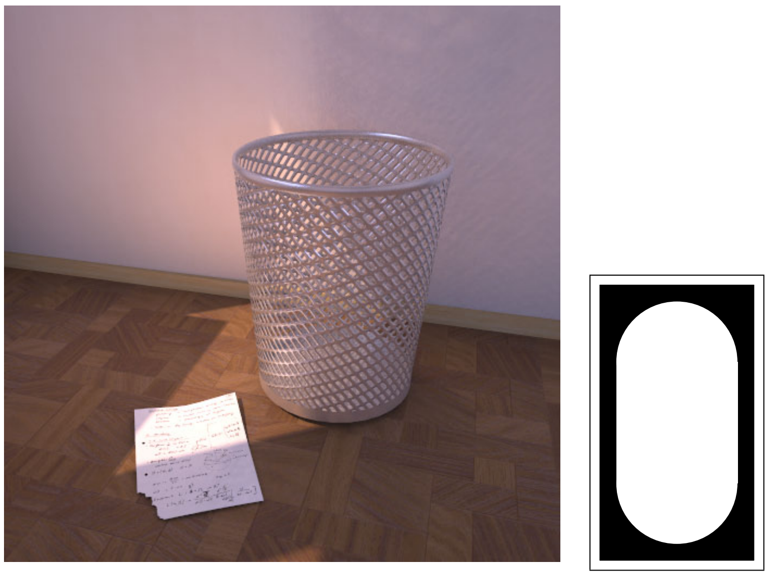


Figure 2.6: Clip Mapping on a trash can. The trash can was modeled from a cone and a clip map was used to create the mesh structure. The clip-map is shown on the right. Note that the torn out corner of the paper on the floor was also modeled with a clip map.

These kind of textures are special because they can be interpreted to not modify the shader but instead to modify the underlying geometry. They could, of course, be implemented as a shader. This but then the shader is responsible for creating a new ray when the surface is clipped away. Considering high performance ray tracing this is inefficient since a lot of overhead is involved in creating a new ray and starting a new traversal at the root node. Instead it is much more efficient to integrate clip maps directly into the ray tracing core. This has to be considered when designing a ray tracer for a full rendering system. To enable a ray tracing core to directly use clip mapping the information has to be accessible during the triangle intersection computation. This adds an additional conditional in the triangle intersection computation to check if a clip map is attached to the triangle and then check if the intersection is clipped away. This direct integration is not as trivial as it may seem as, depending on the ray tracing system, the texture coordinate computation and access can be quite complicated. This leads to a design choice on how tightly coupled the ray tracer should be with the rendering system. For our ray tracing system developed in this thesis that was used for example to create the image in Figure 2.6 we added support for simple 1-bit clip maps directly to the ray tracing core and gave the ray tracer access to the uv-coordinate array.

2.3.4 Material Representation

When talking about materials for rendering systems there are two concepts that need to be distinguished. The first is the bi-directional reflectance distribution function (*BRDF*) which is usually a mathematical model for describing the reflection behavior of a surface. An alternative to a mathematical model is to use measured data ([Matu 03]). The other concept is usually called a *shader*. A shader can be a complex program where the behavior is fully programmable by the artist. While in a general artistic rendering system this freedom poses no major problem for the rendering algorithms it has to be more restricted in the context of physically based rendering. Here we use the notion of a *shader* as a group of BRDF with the parameters for this models included. The BRDF are fixed in our context but the parameters (for example the diffuse color) may be programed or read from a texture. An important fact is that the material model used in a rendering system has a significant impact on the correctness and on the rendering speed. There have been a lot of BRDF models proposed that fulfill the requirements of a Monte Carlo rendering system. For correct results the BRDF has to be energy conserving and needs a representation that allows for efficient evaluation as well as for the ability to importance sample it effectively. This is in contrast to a generic shader that is used in a more general rendering application. Usually a BRDF model is part of such a shader but does not need to obey the energy conservation. In our system we can use any of the physically based BRDF models that were developed like the one by Schlick [Schl 94] or Ward [Ward 92a].

A physically based BRDF model is defined as a function

$$f(x, \omega_i, \omega_o) := \frac{dL_r(x, \omega_o)}{dE_i(x, \omega_i)}$$

that depends on the quotient of the reflected radiance L_r at a given surface-point x and the outgoing direction ω_o and the incident irradiance E_i at the same surface point from incoming direction ω_i. It is used in the integral of the rendering equation 6.1.

In Section 2.3.4 we develop a small extension to the reflection model described in [Blin 77] that is designed to provide all the needed functionality for a flexible material system and is perfectly suited for the rendering algorithm we describe.

Each BRDF model can have many different parameters to tune it's appearance. These parameters can be set for each object but more commonly they are set using texture maps on objects. To support different parameters like diffuse color, different kinds of exponents, specular color etc. the system has to provide the ability to use multiple texture maps per object. This is easy to implement but has to be considered in the data format as it is often needed to provide different parametrizations (called uv sets) for each texture.

A Simplified Practical Layered Material Model

Here we will describe shortly the material model we use in our rendering system. It is especially developed for the use in the path space partitioning method in Section 6.3

but is general and we use it for all of our algorithms.

While general production renderers for the movie industry can use arbitrary and complex programs to define surface color and reflection behavior [Drie 05, Kess 08], physically-based systems are more restricted in their choice of reflection functions [Phar 04]. Many practical algorithms exploit the fact that BRDFs are not only energy-conserving, but also provide an efficient sampling function.

A wealth of surface reflection models have been developed in computer graphics that provide most of the required functionality [Blin 77, Ward 92a, Ashi 00]. These reflection models allow one to describe a wide variety of basic surface types. For more complex surface behavior usually several reflection models are combined to a single layered material [Phar 04]. Physically based rendering systems require the BRDF model to satisfy the Helmholtz reciprocity condition in order to obtain consistent results. The renderer we develop in Section 6.3 partitions path space and uses only one technique on each partition, therefore this condition no longer needs to be fulfilled, allowing us to use more convenient approaches like e.g. the halfway vector disc model [Edwa 06] or even arbitrary programs for surface shading.

In order to determine the partition of the path space, we need to decide when a reflection is almost diffuse. In principle our system can use any physically-based layered BRDF model for which a parameter κ can be assigned to each layer expressing how diffuse it is. This parameter is assumed to be normalized such that for $\kappa = 0$ the layer is perfect Lambertian and for $\kappa = 1$ the surface is a perfect mirror. During rendering each sample evaluates only one material layer that is selected by random sampling. We use the threshold of $\kappa = 0.2$ for classifying a material layer as diffuse or specular.

For example, in all renderings we use the Blinn-Phong model [Blin 77] with Phong-exponent $k = 1024\kappa + 1$. Layered materials such as simple metal or glass (one reflecting, one transmitting layer) and more complex materials like coated plastic can be constructed easily using the Fresnel term or one of the approximations [Schl 94] for weighted sampling of material layers. For transmissive materials the direction is selected according to Snell's law but the width of the lobes is controlled as before, as this allows for diffuse and imperfect glass.

The common approach in a physically based rendering system is to support multiple different material models that are all combined using a standardized sampling interface. This approach allows the artist using the rendering system to freely choose appearance and also allows for plug-in materials. While this increases the flexibility of the rendering system it comes at the cost of more complex scene construction and has a high impact on the rendering speed. Additionally supporting plug-in materials is problematic in the context of physically based rendering because it requires the plug-in author to implement a mathematically correct BRDF model. The speed impact results mostly from the need for a general interface inhibiting pre-computation of parameters and making SIMD processing of the material model difficult.

In contrast to the common method of supporting multiple different material models we propose a simple physically correct material model with layering. Multi layered materials are a natural choice for progressive rendering. In a progressive renderer not all layers need to be evaluated at once but can be sampled according to their weight. For example the Ashikmin-Shirley BRDF model ([Ashi 00]) is an implicit multi-layered material model with two layers, a diffuse and a specular. The obvious advantages of a single material model are that each surface behaves the same (at least within the parameters of the shading model) and a single optimized code path can be used. It also simplifies SIMD optimization of the code.

In our model we use multiple layers of the energy corrected Blinn-Phong model that was presented in [Blin 77]. Each layer is a weighted instance of this basic model. During sampling a random variable is used to choose between these layers and potential absorption. This variable is also used to sample whether the transmissive part of the material should be sampled (using the Fresnel term of this layer). Each layer has two color channels (called C_0 and C_{90}) that are interpolated based on the Fresnel term. This is an artistic extension to the classical wavelength independent Fresnel term that always approaches a value of 1 at gracing angles. This behavior is a special case of our method by setting each channel in C_{90} equal to 1. The effect of this parameter is illustrated in Figure 2.7. With the use of the color interpolation reciprocity is of course lost for this BRDF model, but with our path space partitioning described in Section 6.3 this is not a problem. To control the specular behavior each layer has a parameter called hardness (κ) that varies from 0.0 (Lambertian) to 1.0 for a perfect mirroring layer.

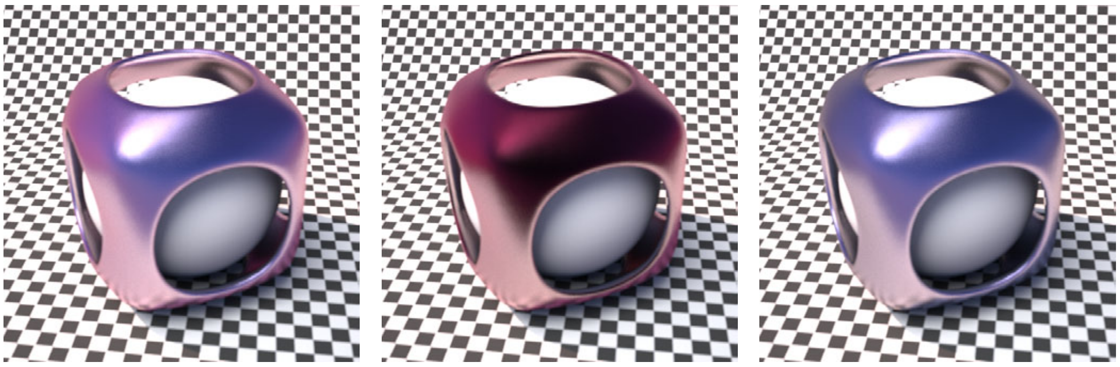


Figure 2.7: Visualization of using the Fresnel term to interpolate between two colors. The left image shows the interpolated results. The middle image has $C_0 = 0$ and the right image has $C_{90} = 0$. Note the difference in the highlights from the sun and the slightly different position.

In the following we describe the material model used for each layer in our system. As already mentioned above we use the Blinn micro-facet distribution with the additional normalization needed for physically based rendering.

$$f_r(x, \omega_i, \omega_o) = \frac{\langle n, h \rangle^k \cdot \frac{k+2}{2}}{4\pi \langle n, \omega_o \rangle \langle h, \omega_o \rangle} \tag{2.1}$$

The exponent k is computed from the hardness value κ specified for each material

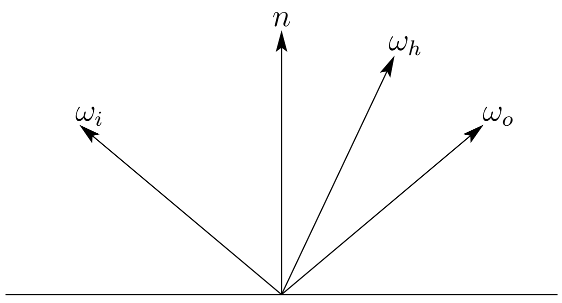


Figure 2.8: The directions and vectors used for the BRDF computation of a single layer.

layer. A practical value to map the interval of $\kappa \in (0, 1)$ to the exponent k is given by $k = 1024\kappa + 1$. There are two special cases of this BRDF for $\kappa = 0$ and $\kappa = 1$. In the case of $\kappa = 0$ a Lambertian layer is constructed and for $\kappa = 1$ a perfect mirror is assumed. Note that equation 2.1 does not produce itself a Lambertian BRDF for $\kappa \to 0$. Figure 2.9 illustrates the effects of this parameter for a single layer material model.

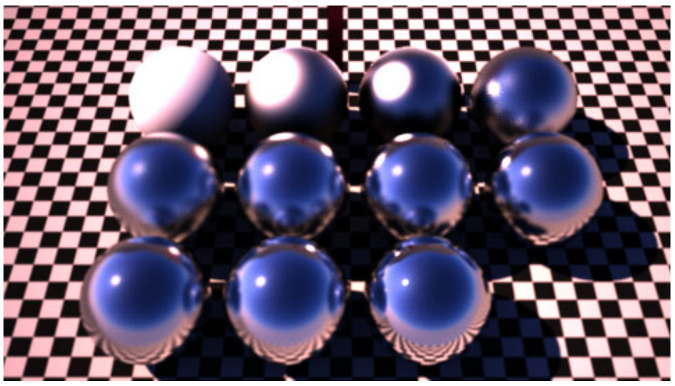


Figure 2.9: Varying the hardness from 0.0 to 1.0 for a single layer material.

To edit these materials we implemented an easy to use material editor that integrated the full renderer to preview the parameters. Figure 2.10 shows two screenshots of this editor. Since the material model is very simple the functionality depicted in these two screenshots is already enough to create a wide variety of realist materials that work with our rendering system. Two materials are illustrated in Figure 2.11.

2.3.5 Sampling Light Sources

For almost all global illumination algorithms, the rendering system has to be able to generate correctly weighted sample points on light sources. As we consider only phys-

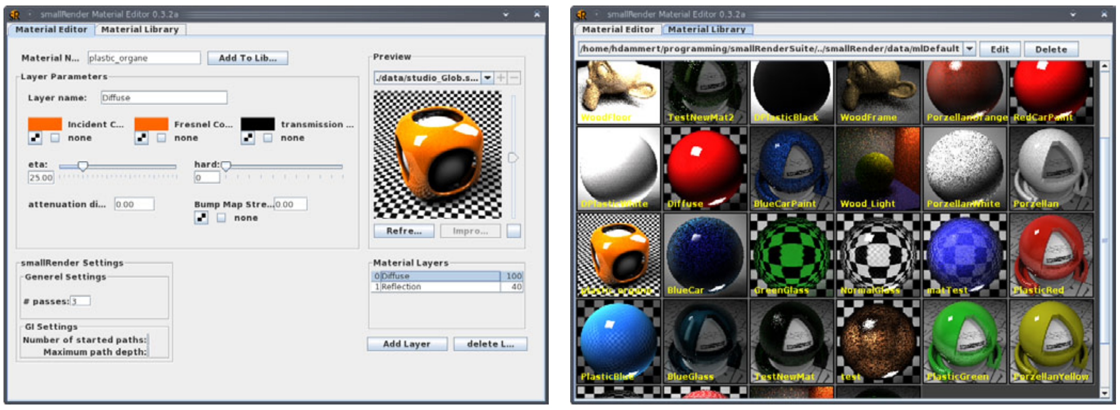


Figure 2.10: The Material Editor developed for the layered BRDF described above shows the simplicity of the model. The two screen-shots already describe the full functionality needed.

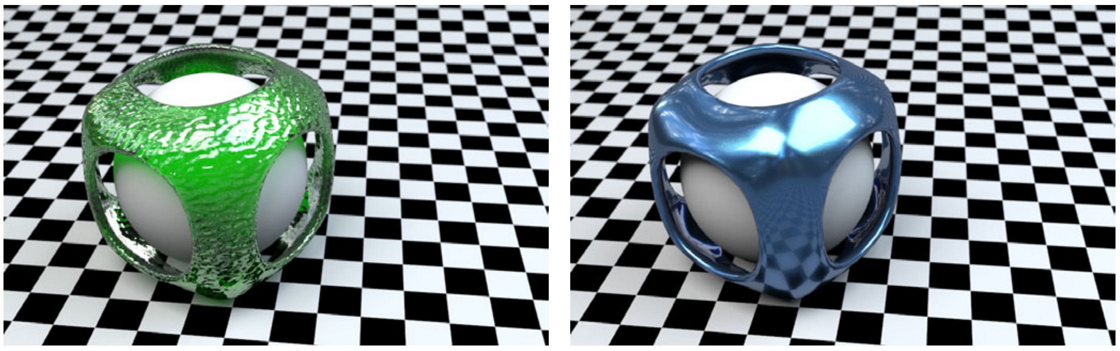


Figure 2.11: Two example renderings illustrating realistic materials created with our material model. The left shows a glass material and the right a multilayer car paint material.

ically plausible scenes in our rendering system, light sources (with the exception of environment maps) are part of the scene and modeled as triangles. These triangles are marked as emitters and have the light source information like spectral properties and emittance behavior. The usual way to generate sample points on light sources is to build a cumulative density function (CDF) using the emittance power of each triangle. Now three random or quasi-random numbers are needed to select a triangle and a point on this triangle. This approach works well if all light sources in the scene contribute to the final image. Since the CDF is build based on the power, bright light sources are sampled more often than dark light sources. The problem with this approach is a dramatic decrease in efficiency when larger scenes are rendered. For example a complete modeled house has many light sources but the camera is only in a single room. An approach in unbiased rendering algorithms is to build the CDF based on the light source contribution to the final image [Sego 06].

This can work well but needs many light samples until a good CDF is acquired. This many light samples are not available when directly using only point light sources. Another problem with this method is, that it can't be used when the scene is illuminated by a sky model ([Pree 99]) or when image based lighting ([Debe 98]) is used.

We develop an approach completely based on the usage of point light sources that works also well when image based lighting and other environment illumination is used by discarding unimportant light sources stochastically. This approach is described in Section 6.3.2. A full rendering system also needs to support image based lighting using environment maps. In Section 6.4 we will present a physically correct solution on how to integrate infinitely far away illumination with a point light based rendering system.

2.4 Processor Architecture

In light transport simulation there are a few core algorithms that will be executed a million times or more and will amount to over 90% of the computation time. For this reason even small optimizations to these core algorithms are beneficial for the final rendering speed. So after considering all algorithmic optimizations it is important to directly optimize for the target processor architecture. Additionally, the shift in processor architecture has to be also recognized by algorithm design. Simply using classical algorithms on modern architectures does not give the performance the new hardware promises.

This shift in processor architecture consists mainly of two trends. The first one is an increase in the number of processing elements and the number of instructions that can be executed in parallel. Multi-core and many-core are almost ubiquitous and additionally the increasing availability of chip area allows to pack multiple computation pipelines and execution units into the CPU. This results in different parallelizations that are available to the programmer that have to be recognized when designing algorithms. At one end are multiple full processor cores that are able to execute independent streams of instructions. On a lower level there are SIMD (single instruction multiple data) units that allow to execute a single instruction on several data elements. SIMD is explained in more detail below. The most common size for SIMD units is currently 4 but in the future much wider SIMD-width are to be expected [Seil 08]. Further forms of parallel execution can be found in multiple execution units that allow for example to execute a memory fetch and a computational instruction in parallel at the same time. Usually on this low level a programmer expects the compiler to know the underlying architecture and optimize for it. But a knowledge of this behavior aids algorithm design. Knowing for example that a computation between memory fetches is virtually free may change the design significantly. The second trend in processor architecture can be explained by the divergence of computation speed and memory access time. This leads to either deeper and large caches or different memory models that require coherent memory access.

For maximum performance each new architecture and the trade-offs have to be reconsidered. Thus partitioning the ray tracing algorithm into several distinct parts makes it easier to understand the problems and to find fitting solutions for the chosen hardware platform.

It is likely that the upcoming computer architecture for commodity hardware become even more heterogeneous than it is today. Several different *computational engines* will be found in the usual desktop computer. Aside the graphics processing unit with it's ever increasing power there exist affordable acceleration cards for Physics Simulation or more general acceleration cards like an additional CELL processor on an PCI-Express card. Even inside a single architecture like the CELL-Processor a heterogeneous collection of two different computational engines can be found.

This heterogeneous hardware allows one to spread the computation over different processors and a first approach to use it would be to use the computational engine that is best suited for the given task. Another goal could be to use all different available computational engines to solve a given task as quickly as possible. In that case it may be an enormous task for the application developer to efficiently use all different engines at the same time without stalls.

In the following we shortly review some important architectural topics that later in this thesis guided the design decisions and may be required to understand some of the optimizations. More details are given when needed in the specific sections. In this thesis we are mainly concerned with accelerating ray tracing on general purpose processors that can be found in any desktop computer or notebook. In the last few years an increasing amount of research has been done to bring ray tracing to graphics processing units (GPUs) and their unique architecture. While these are quite successful and provide sometimes much higher speed there are still some limitations that restrict the use of these ray tracers in a fast full global illumination renderer. One problem is for example the limited amount of memory these graphics cards have in many cases. Another problem is that complex shading operations are solved mostly in a hybrid approach and can not fully utilize the advantages of GPUs. There is still a lot of ongoing research to create systems that efficiently use the graphics cards. Most of the results we present in the next section can be employed to any ray tracer regardless on which hardware it runs on. But some design considerations especially for the novel acceleration structure we present in Section 3.2 is specifically targeted towards CPU architectures. We want to note here again that we are mostly concerned with accelerating incoherent rays that occur in global illumination computations. These are very troublesome for all GPU ray tracers that usually slow down to speeds comparable to CPU ray tracers. Only recently some hardware modifications to allow GPUs to trace faster incoherent rays have been presented in [Aila 10] but these are still only theoretical.

Parallel Processing

There exist a lot of different parallel architectures and several programming models that can be used to parallelize a given problem. A very good introduction to parallel programming can be found in [Quin 03]. On the usual multi-core (CPU) machines a ray tracing based image generation algorithm is rather easy to parallelize once the acceleration structure has been build. The parallel construction of acceleration structures is an interesting problem and has for example been investigated for *k*d-trees in [Ize 06] or for bounding volume hierarchies in [Laut 09]. Each core may have it's own caches but they all access the same main memory. Since each ray is at first independent of each other ray they can be traced in parallel and only the final combination in the frame buffer has to be synchronized. On NUMA (Non-Uniform Memory Architecture) machines building a scalable solution is more difficult. A possible solution strategy for the visualization of larger models is described in [Step 06].

Intrinsics. To really utilize a given hardware architecture the written code needs low level access to all available functions. Many complex operations are often available as a single CPU instruction but it may not be possible to express this directly in a high level language. Still, for developing high performance algorithms it is not necessary to directly program in the assembly language of the processor. High level language compilers like many C and C++ compilers provide a mechanism called *intrinsics* that allow to access specific instructions directly from within high level code. These intrinsics are implemented by many compilers as extensions to the language standard. For example the usage of intrinsics for the Intel compiler is described in [Scho 03] and the GCC is compatible to the intrinsic usage for Intel platforms. Intrinsics not only allow to perform specialized instructions on registers and memory regions but also can access for example hardware counters or control the cache access and pre-fetching. But they are of course specific to a given hardware. This has the disadvantage of reduced portability. Each different hardware platform from another vendor has its own set of intrinsics. Hardware from a single vendor is usually backwards compatible but introduces new functions with each release. Still the benefit of using intrinsics is too large to be ignored, especially for SIMD processing described in the next Section. To support multiple platforms we will develop and use a simple abstraction layer described also in the next Section.

SIMD Processing. SIMD is an acronym that stands for Single Instruction Multiple Data. It is a form of parallelism where multiple streams of data are processed by a single stream of instructions. A processing width of 4 then means that 4 data elements are processed by a single instruction in parallel. This is illustrated in Figure 2.12 for two simple operations.

Often the number of data elements is not directly fixed but the width of a single processing element. A common size are 128-bit registers and all SIMD instructions operate on this width. In a 128-bit register fit for example four 32 bit floating point numbers or integers or 16 8-bit chars.

Almost all modern hardware architectures provide hardware support for SIMD processing each with its unique marketing brand name. The companies *Intel* and *AMD*

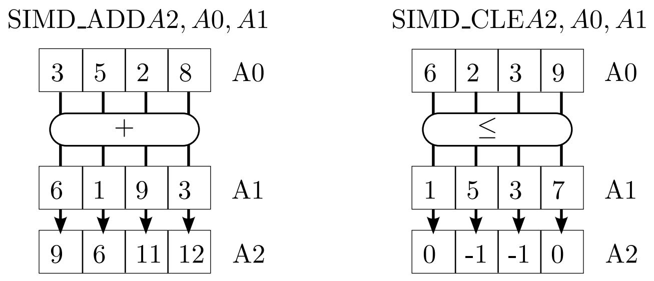


Figure 2.12: Illustrating the principle of SIMD computation. The first instruction is a simple add. Adding two SIMD registers results in a component wise addition. The second instruction is a compare for less or equal. This is again done component wise. The result of a comparison is 0 when it is false. When the comparison results in true each of the bits in the component are set to 1. This is the binary integer representation of -1. The reason for setting all bits to 1 is that the result can now be easily used in masking operations by using bit-wise *and* and *or* operations.

now support the SSE instruction set (where various different versions exist). The *Intel* SSE instruction set is described in detail for example in [Inte 09]. The older SIMD instructions sets that are still supported are *Intel* MMX and *AMD* 3dnow. *Sun* has the MAJC architecture, *IBM* has Altivec for the PowerPC and CELL SPE [Cell 09] architecture, *ARM* names it NEON [Cort 09], and for the SPARC architecture it is called VIS.

All these processing units are vendor-specific and provide different functionality with no unified or standardized way to program them. The usual way to develop for them in a high level language is to use intrinsics but they are also platform dependent. Luckily even though the used instructions may differ in details, many of the standard operations have equivalents on most other architectures. So, for example, a SIMD width of 4 (currently the standard on almost all CPUs) and the basic operations like addition, multiplication and bit wise operations are available everywhere. This allows one to use a simple and basic abstraction layer building a wrapper around the intrinsics. For the use in this thesis we developed a simple collection of C pre-processor macros that has a back-end for PowerPC Altivec, Intel SSE and a software implementation for computers without SIMD instruction set and for debugging purposes. Instead of pre-processor macros inline functions could also be used. This abstraction can easily extended to support other SIMD vector units of the same size. It is important to note that the abstraction should be as close as possible to the underlying hardware as else many of the low level optimizations could not be implemented.

In the following we present the basic ideas by presenting the implementation of some selected functions. This also serves as an introduction to SIMD programming. The first step is to define a type for the basic most commonly used data types. In the case of a ray tracing based image generation software the most basic types are a 32-bit integer

```
#ifdef USE_ALTIVEC
  typedef vector float SIMD_float;
  typedef vector int SIMD_int;
#elif definded(USE_SSE)
  typedef __m128 SIMD_float;
  typedef __m128i SIMD_int;
#else
  typedef union ALIGN16 SIMD_float {
    float ALIGN16 f[4];
    unsigned int ALIGN16 i[4];
  } ALIGN16 SIMD_float;
#endif
```

Listing 2.1: Data structure definition for the SIMD abstraction code.

and 32-bit floating point numbers. For the software implementation a union is used because of the ability of all vector units to interpret the contents of a register without conversion. On PowerPC there exist a language extensions for vector units that uses overloaded operations. So for example for an addition no direct intrinsic is needed and the add operator $+$ can be used.

Now implementing the basic arithmetic and logic functions is straightforward. Listing 2.2 shows as an example the implementation of two arithmetic instructions for the SIMD abstraction layer. The Altivec version can use the overloaded operator, the SSE version uses intrinsics and the non vectorized version modifies each component individually. Note that the compiler automatically unrolls the `for` loops in this case.

The last example shown in Listing 2.3 is more complex and illustrates the possibility to support instructions that are not equal over different architectures efficiently. Here we chose the SSE shuffle instruction that is needed for example for the SIMD sorting used in Section 3.3.3. Luckily the permutation function of Altivec units is able to perform the operations of an SSE shuffle instruction. The shuffle instruction is best understood by looking at the software implementation in Listing 2.3. The goal is to reorder the 4 elements in an SIMD register by providing its new positions `a, b, c, d`. For the SSE version there exist a macro in the SSE API `_MM_SHUFFLE(...)` that constructs the required bit mask for the `_mm_shuffle_ps` instruction. The Altivec `vec_perm` instruction is more powerful than required but with a simple look-up table the workings of the shuffle instruction can be mapped. This is of course non-optimal considering the Altivec architecture as more instructions are needed but in our application the shuffle instruction is only a small part of the algorithm and it does not hurt the performance much compared to the fact that the same code can be used on all platforms. Some of the modern compilers have the ability for auto-vectorization. This can, in theory, automatically generate SIMD code from the program code but works only in very few cases. So manual written code is still far superior to compiler generated SIMD code for complex algorithms.

```
#ifdef USE_ALTIVEC
  #define SIMD_ADD(a, b) ((a)+(b))
  #define SIMD_CLE(a, b) ((vector float)vec_cmple(a, b))
#elif definded(USE_SSE)
  #define SIMD_ADD(a, b) (_mm_add_ps(a, b))
  #define SIMD_CLE(a, b) (_mm_cmple_ps(a, b))
#else
  static inline SIMD_float SIMD_ADD(SIMD_float a, SIMD_float b) {
    SIMD_float r;
    for (int i = 0; i < 4; i++) {
      r.f[i] = (a.f[i] + b.f[i]);
    }
    return r;
  }
  static inline SIMD_float SIMD_CLE(SIMD_float a, SIMD_float b) {
    SIMD_float r;
    for (int i = 0; i < 4; i++) {
      r.i[i] = (a.f[i] <= b.f[i]) ? -1 : 0;
    }
    return r;
  }
#endif
```

Listing 2.2: Basic arithmetic instruction macros for the SIMD abstraction layer. The instructions are an addition (ADD) and a comparison for less or equal and are explained in Figure 2.12.

Memory

Increasing the number of cores not only poses new problems for the algorithm design but also increases the well known problem of the divergence of memory access speed versus compute power. While in the last years the compute power has almost increased exponentially according to Moore's law[1] the memory access speed has grown only linearly. This gap is well known and one of the common solutions to alleviate this is by the introduction of cache hierarchies. The basic idea here is to introduce additional memory blocks that are smaller but are located closer to the CPU and thus can be accessed faster. A very good and complete treatment of the memory subsystem and caches in modern commodity hardware can be found in the technical report by Drepper [Drep 07]. In the following we just review the aspects that are directly important to the optimization considerations developed in this thesis.

At first, a programmer has to be aware of the existence of caches or their absence. For example on the CELL architecture or on GPUs there may be no hardware caching for some of the memory accesses but local memory that is programmer controlled. In these cases a software managed cache is often a viable alternative. Directly consid-

[1] Actually Moore's low only states that the number of transistors roughly doubles every two years, but due to ever improving hardware designs and on-chip parallel processing like SIMD this directly relates to the raw compute power of a CPU.

```
#ifdef USE_ALTIVEC
  static const unsigned int _av_shufmask[] = {0x00010203, 0x04050607,
                                              0x08090A0B, 0x0C0D0E0F,
                                              0x10111213, 0x14151617,
                                              0x18191A1B, 0x1C1D1E1F};
  // Shuffle Semantic: c,d,e,f in [0,3] (SSE-Style)
  //                   e,f are lower; e,f select from a
  //                   c,d are upper; c,d select from b
#define SIMD_SHUFFLE(a, b, c, d, e, f) (
  vec_perm(a, b, (vector unsigned int){_av_shufmask[f], _av_shufmask[e],
                                       _av_shufmask[d+4], _av_shufmask[c+4]})
                                      )
#elif definded(USE_SSE)
#define SIMD_SHUFFLE(a, b, c, d, e, f) (
   _mm_shuffle_ps(a, b, _MM_SHUFFLE(c,d,e,f))
   )
#else
  static inline SIMD_float SIMD_SHUFFLE(SIMD_float f1, SIMD_float f2, int a,
                                        int b, int c, int d) {
    SIMD_float r;
    r.f[0] = f1.f[d]; r.f[1] = f1.f[c]; r.f[2] = f2.f[b]; r.f[3] = f2.f[a];
    return r;
  }
#endif
```

Listing 2.3: Implementation of the SIMD shuffle instruction for the abstraction layer. Shuffle reorders the 4 elements of an SIMD register. The Altivec `vec_perm` instruction is more powerful than the SSE counterpart as it allows reordering and replication of individual bytes. This explains the lookup-table `_av_shufmask`.

ering a cache hierarchy in an implementation means to fit the data types and memory accesses to the cache line width and the access structure of the main memory. The next thing to consider is the latency of a memory access. This latency of course varies greatly on where the requested data is currently stored in the cache hierarchy. Knowing this latency may allow an algorithm to run faster by reordering for example the instructions in a way that after a memory request other independent computations can be performed. Usually a modern compiler tries to optimize given code in a way fitting to the hardware but it can only use information locally available. A programmer has application domain knowledge and information about the actual execution behavior of an algorithm that allows even better optimization[2].

A rather old technique to hide some of the memory access latency is called simultaneous multi-threading [Tull 95]. Intel implemented the technique [Marr 02] in hardware and called it hyper threading. This technique basically allows to switch executing an-

[2]Modern compilers support profile guided optimization that allows to run an instrumented version of a program to automatically gather run-time information that is used in a second compilation step for further optimization

```
int value;
int* p = &value;
p = p | 2; // store some information in the last two bits
...
if (p & 3 == 2) {... // do sth. with the information stored
...
int number = *(p & 0xFFFFFFF7); // access the original value
```

Listing 2.4: Bit Operations to use the two unused bits in a memory address to store and retrieve information. This is for example useful when storing child pointer in acceleration structures.

other stream of instructions when a high latency memory access occurs. Simultaneous multi-threading can also be implemented in software and is a common approach to hide DMA latency for example on CELL architectures [Cell 09].

Instead of manually optimizing algorithms to a specific memory hierarchy and caching structure there exists another approach called cache oblivious algorithms. An overview can be found in chapter 38 of the "Handbook Of Data Structures and Applications" [Meht 04]. While not achieving peek performance they can make some algorithms more portable across multiple platforms and still provide high performance. But of course they can not adapt to situations like the CELL architecture where no caches are available at all. Nevertheless these methods can be directly applied to computer graphics related problems like mesh layout [Yoon 05] and for bounding volume hierarchies [Yoon 06].

Low Level Operations

Programming close to the underlying hardware is important in getting the last bit of performance out of a given algorithm. Additionally knowing the exact size and bit layout of data types allows to efficiently use all available bits. In this section we review the important data types and standard approaches that are used during optimization. A very good and helpful book about optimizing computations using the benefits of the binary number representation in a computer is [Warr 02].

Bit Operations. When optimizing algorithms it is often beneficial to use every last bit in a memory region. Especially when memory access is optimized collecting all information in a small space reduces the number of memory accesses and cache thrashing. For example the standard size of an integer or floating point number is currently 4 bytes. For alignment reasons a memory location for such a number is divisible by 4. So when a pointer to this memory address is stored that last two bits are always 0. Knowing this, it is easy to store up to 4 different additional states by using a bit-wise `or`. Using this method it is necessary to mask these two bits before using the variable for memory access. The small code section in Listing 2.4 shows an example usage on 32bit architectures. This is a common approach to compactly store the nodes in an acceleration

1	8	23
sign	exponent	mantissa

Figure 2.13: Bit layout of an IEEE 32-bit floating point number.

structure for ray tracing where for example the split axis of a *k*d-tree node (3 possible choices in 3d) can be stored in a memory pointer. The last state can be used to indicate a leaf node.

Floating Point Representation. The IEEE 754 standard [IEEE 85] for floating point numbers (floats) is widely adopted now by processor manufacturers. Nevertheless many vendors only implement a subset of the full standard for performance and die area reasons. For example it is common to omit the support for denormalized numbers or only provide a subset of the standardized rounding modes. Sometimes the accuracy may also vary but the general representation of a floating point number is still the same across a wide range of platforms. An introduction to the use of floating point numbers in scientific applications is given by Goldberg in [Gold 91]. The general layout of a 32-bit floating point number is illustrated in Figure 2.13. As 32-bit floats provide in many cases enough accuracy for graphics application and they are the only ones considered here. Looking at the bit representation of a floating point number we can use some of the bits for other purposes when we know something about the stored number. For example when we know that the number is always positive we can use the first bit in the floating point number. When we know that an absolute value of a number is always smaller or equal to 1, a single bit in the exponent is free (This is for example the case when storing normal vectors or RGB-float colors). When not the full accuracy of the 23-bit mantissa is needed an elegant way to gain some additional bits is to use some of the least significant bits of the mantissa. This has also the advantage that depending on the computation no masking has to be performed. These tricks are necessary to allow for compact storage of acceleration structure for ray tracing but still be able to access the data without a complex unpacking operation. Masking single bits in a register is very fast and is usually a lot faster than an additional memory access to request further information. In later sections when we use additional bits we will discuss it in detail how the masking and access is handled in each case.

There exist of course other ways to compress data but these are always either slower due to decompression or reduce the precision. The above techniques are actually no compression but just try to use the available space as efficiently as possible.

Hardware Dependent Differences. When talking about low level operations it is important to recall the most significant differences in hardware architectures a programmer should always be aware of to avoid incompatibilities. When non-portable code is written these differences can be ignored but when it is likely that some of the code

```
if (__builtin_expect((b), TRUE)) {
...
} else {
...
}
```

Listing 2.5: Branch Hints can be given to the compiler using intrinsics.

should be executable on different architectures it is helpful to keep them in mind. The first important difference is the natural size of data types and pointer sizes. For example the switch from 32-bit architectures to 64-bit architecture changed the natural size of a pointer. So arrays of pointers suddenly needed twice as much memory and a pointer did not fit anymore inside an integer. The memory alignment requirements also differ. For example many SIMD units require 16-byte alignment of the memory address and either fail completely when it is not met or suffer a severe performance penalty. Another difference may be the native width of integer computations. For example many current GPUs only have a 24-bit integer unit and slow down significantly when 32-bit integer arithmetic is used. Little endian architectures and big endian architectures also may cause trouble when binary data is loaded from disk or bit operations are performed. Other performance related differences are for example the cache line size or the number of bytes that should be accessed together for optimal performance of the memory subsystem.

Keeping these differences in mind it is possible to write portable code even when low level operations are used but the programmer may have to write more code and be more careful about choosing the implementation strategy. Finally we want to stress that the techniques sketched in this section should be only used where they are really needed. Properly used the may provide the additional performance needed to allow the computation to finish in the required time but should only be used in the most critical code sections.

Branch Hints. To achieve high clock rates, current processors have deep execution pipelines and perform aggressive out of order execution of instructions. A branch (or conditional jump) in the code can thus significantly impact the execution performance when the execution has to be changed and the pipeline needs to be filled from the beginning. The performance impact of branches is a well known problem and one solution strategy is to use branch prediction [Smit 81]. Modern processors have very sophisticated branch prediction mechanisms.

A compiler usually has knowledge about the processors branch predictor and can use static analysis to provide the necessary hints during execution [Ball 93] and can deliver an accuracy of up to 70%. Dynamic profiling analysis uses a special benchmarking build to collect information about the execution behavior and later use this information in a recompilation to reorder the given branches. Nevertheless the programmer can use domain specific knowledge to gain additional performance. The use of branch hints is build into many compilers and can give performance benefits when the execution behavior is known. For example when the programmer expects that a branch based on a Boolean expression `b` is likely to be taken this can be hinted to the compiler by the intrinsic shown in Listing 2.5.

3 Efficient Tracing of Incoherent Rays

In this chapter we will introduce a novel acceleration data structure for ray tracing. Parts of this have been published in [Damm 08a]. The basic task of a ray tracing algorithm is to provide an efficient method to intersect a given ray quickly with a given scene database. As already discussed in Section 2.3.2 our scene representation is based on triangles. The necessary ray triangle intersection was mentioned in Section 2.3.3. The task of an acceleration data structure is now to allow for an efficient ray traversal (traversing the data structure until a leaf or node with triangles is found) and to quickly prune triangles that are not in the path of the ray. Section 3.1 will give an overview of the commonly used acceleration data structures. In Section 3.2 we will introduce our novel acceleration data structure and in Section 3.3 we will more closely investigate the ray traversal. Section 3.4 provides an introduction to on demand hierarchy construction and the results are presented and discussed in Section 3.5

When performing light transport simulation in realistic scenes we already noted in the previous chapter that ray tracing is often the only choice to robustly simulate all possible light paths. The core ray traversal algorithm is then the major bottleneck consuming about 80% to 90% of the total computation time. While there exist many acceleration strategies exploiting coherence (see Section 3.5.3 for a short overview) this coherence is almost never available in full global illumination algorithms.

The basic idea is instead of searching coherence in the rays, coherence is created directly in the acceleration structure and used implicitly. In the following Chapter 4 we will investigate how this acceleration structure can be constructed efficiently and in Chapter 5 we will improve the performance even further by introducing software caching. Bounding volume hierarchies (BVH) belong to the simplest and most efficient acceleration schemes for ray tracing. They are usually constructed as a binary tree of bounding volumes but in this chapter we will introduce and investigate higher arity bounding volume hierarchies. Memory requirements [Wach 07] and memory latency [Wald 07a] have been recognized as bottlenecks and fast hierarchy construction has been investigated [Wach 06, Wald 06a, Shev 07]. Section 2.4 introduced SIMD instructions that are used extensively in this Chapter to describe our optimized acceleration structure.

As already mentioned, algorithms that perform physically correct simulations of light transport tend to shoot rays as wide-spread as possible in order to increase efficiency, e.g. by employing quasi-Monte Carlo methods. As a result, most rays that account for global illumination effects are incoherent. Because of that, major parts of the simulation cannot benefit from tracing packets of rays. This has been recognized in professional rendering products [Chri 06, Sec. 6.4], too. They use SIMD instructions by intersecting multiple objects at a time. Compared to tracing packets this has the major disadvantages that hierarchy traversal is not accelerated and that memory bandwidth is not reduced.

We address the bandwidth problem in Section 3.2 by reducing the memory footprint of the acceleration data structure. In contrast to reduced-precision methods [Maho 05, Hubo 06], which work best when the overhead of transforming the packed data to world space is amortized over a bundle of rays, we increase the arity of the acceleration hierarchy, which saves memory for nodes and pointers. This is similar to a lightweight BVH [Clin 06], but in the context of efficient SIMD processing and without the drawbacks of limited precision and median split hierarchy construction. The resulting trees are flat compared to standard BVHs. Smits [Smit 98] naturally recognizes the possibility of BVHs with higher arity (without regard to parallel processing), but does not indicate how to construct the acceleration structure in this case while still minimizing a construction cost metric like the surface area heuristic (SAH). We will investigate this in Chapter 4. Also, the suggested data structure does not support sorted traversal, which we found to be crucial for good performance on current CPU architectures, especially for intersection rays. Sorted traversal is presented in Section 3.3. He also points out the opportunity of speeding up shadow rays using a cache. As this cache only stores one single primitive with the previously found intersection and is not hashed per light source, he states that the method only works well with large triangles and very coherent rays. We investigate the use of software caching in Chapter 5 and show that our method works very well with many small triangles and highly incoherent rays. Recently there has been further investigation of the advantages of multi-branching bounding volume hierarchies in [Erns 08, Tsak 09].

3.1 Overview of Acceleration Data Structures

In this section we will give an overview of the commonly used data structures used to accelerate general ray tracing. Realistic triangle scenes contain at least millions of triangles to approximate the geometry. These scenes can not be rendered in a reasonable amount of time by intersecting a given ray with all primitives. Such an algorithm is $O(nm)$ with n being the number of triangles and m the number of rays. Note that also several million rays are needed for the computation of an image. Tracing only the primary rays with one ray per pixel of a normal full HD image (resolution 1920×1080) requires 2 million rays without any secondary effects. So already early in the history of ray tracing spatial and hierarchical acceleration structures have been developed to speed up the scene intersection process by reducing the number of search steps per ray for example to $O(\log(n))$. A summary of acceleration structures can be found in [Glas 89]. A thorough comparison of many acceleration structures can be found in [Havr 01]. Using acceleration structures significantly speed up the rendering process by reducing the number of ray triangle tests needed but they still guarantee the correct result. Building acceleration structures of course costs computation time but for static scenes it has to be done only once and can then be reused for all rays amortizing the build time quickly. In the following sections we will describe and discuss shortly the three major acceleration structure types for triangle scenes.

3.1.1 Bounding Volume Hierarchies

Bounding volume hierarchies (BVH) are an object partitioning method. (in contrast to space partitioning described in Section 3.1.2). They have a long tradition in the context of ray tracing and many variants exist [Gold 87, Wach 06, Geim 06, Wald 07b]. The basic idea is to divide recursively the array of all objects into disjoint subsets. For each subset a bounding volume is computed that fully encloses all contained objects. Starting with the list of all objects this process is repeated until a termination criterion is met. The simplest criterion is to stop when only a single object is left in the list. The number of subsets n (with $n > 1$) is arbitrary but the common choice in ray tracing is to build binary hierarchies. The choice of bounding volume is also arbitrary but has a significant impact on speed and construction complexity. The chosen bounding volume should be easy to compute from a list of objects and should be fast to intersect with a given ray. Additionally it should enclose the list of objects as tightly as possible.

Basic traversal of a BVH is straightforward. A given ray is intersected with the hierarchy starting at the root node and traversing only children that are intersected by the ray. It is important to note that in a basic implementation all nodes that are intersected by the ray need to be checked since the bounding volumes can overlap. So no early termination during traversal can be done as in a *k*d-tree (Section 3.1.2) or a grid (Section 3.1.3). Several optimizations can be introduced for the traversal that significantly can increase the performance. One is to use ray coherence that will be discussed in Section 3.5.3, the others are described in more detail in Section 3.3 in the context of our new acceleration structure.

For the construction of bounding volume hierarchies the choice of where to split the list of objects and how to efficiently sort them is the most important. This choice is usually achieved by employing a heuristic. Simple center-, and median-split heuristics are easy to implement but are much slower in the ray tracing performance. Currently the best known method in the general case for hierarchy construction is to use the surface area heuristic (SAH). In [Wald 07a] an efficient construction based on the SAH for bounding volume hierarchies is described. We give an overview of these construction heuristics in Section 4.1 and analyze the impact of these algorithms for our novel acceleration structure in Section 4.2.

3.1.2 *k*d-Trees

The *k*d-tree is a space partitioning method. In contrast to object partitioning like a BVH described in Section 3.1.1 the space is subdivided into disjoint volumes. There exist several methods that fall into the category of space partitioning. Octrees (see for example [Same 89]) are one popular choice but they can be seen as a special case of a general *k*d-tree. Grids are a non-hierarchical space partitioning method and are described in Section 3.1.3. In [Szir 02] an average case analysis for space partitioning methods with simulation results can be found.

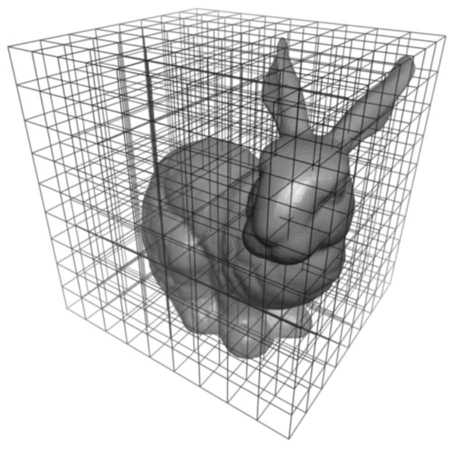
A *k*d-tree is a binary hierarchy and each node contains only an axis-aligned split plane and references to the children. It is one of the most popular acceleration structures for ray tracing [Havr 02, Shev 07]. The k stands for the dimension of the hierarchy and $k = 3$ for typical 3d scenes. The split plane divides the space into a positive and a negative half-space sometimes also called left and right. The children then recursively subdivide the space further. The construction of such a *k*d-tree significantly influences the performance and also the memory requirements. A very good overview and analysis of different construction and traversal methods can be found in [Havr 01]. In [Wald 06a] the construction of *k*d-trees is analyzed and a method to build them in $O(n \log(n))$.

One important advantage of a *k*d-tree over a BVH is that the ray automatically traverses the hierarchy in order of increasing distance and distinct nodes never overlap. This allows for an efficient termination of the ray after the first intersection which is called “early out”. On the other hand *k*d-trees have a significant disadvantage that is inherent to all space subdivision methods. Individual objects may be in more than one node at the same time as it is almost always impossible to find a split plane that does not intersect some objects. This leads to a duplication of references onto these objects that increases the memory consumption and may result in multiple intersections with the same object during traversal. The multiple intersection problem can be solved with mailboxing [Kirk 91] but the memory consumption is in the average case always higher than BVH memory consumption. Nevertheless *k*d-trees are among the fastest ray tracing acceleration structures.

The generalization of a *k*d-tree is called binary space partitioning (BSP) tree and allows for arbitrary planes to be chosen [Ize 08]. This comes at the cost of a significant higher memory consumption because now arbitrary planes need to be stored instead of only 3 different axes and also the construction is much more complicated and needs more computation. In [Kamm 07] the authors analyze a hybrid called restricted BSP trees, where a fixed number $n > 3$ of different plane orientations is allowed that can still be stored compactly. In [Budg 08] the efficient construction and traversal of restricted BSP trees is discussed.

3.1.3 Grids

Grids are in their basic implementation a non-hierarchical acceleration structure. The axis aligned bounding box of the scene is subdivided uniformly into non-overlapping equally sized volume elements that are also axis aligned. An illustration of such a uniform grid can be found in the Figure on the right. The construction of a grid is very easy and fast. Each volume element of the grid has an assigned list. Each primitive of the scene is inserted in all the lists of the volume elements that it intersects. This is basically an $O(n)$ operation because each of the n primitives has to be touched only once. For that reason it is often used for interactive scenes where the acceleration structure has to be updated each frame [Rein 00]. In [Laga 08b] the authors describe two efficient methods for building grids and in [Kalo 09] a parallel construction is presented. The traversal with a ray is done using a modified DDA algorithm that is similar to line rendering (see for example [Aman 87] and also [Laga 08b]). The traversal has to guarantee that all voxels the ray intersects are visited. An advantage of grids is, similar to a *k*d-tree, that the intersection computation can be terminated as soon as the first hit is found (early out) as the grid is also traversed in ray direction with monotonic increasing distance. An important disadvantage of grids comes from the uniform subdivision. Due to this, grids are bad for adapting to locally fine detail with large empty space. For efficient ray tracing of fine detail (i.e. many small triangles) a small grid size needs to be chosen but then large empty space has to be traversed in many small steps significantly slowing down the ray tracing process. A solution to this problem is using adaptive grids [Klim 97] or multi-level grids [Park 99]. Another interesting approach to improve the adaptivity of grids is given in [Laga 08a]. There, the authors use constrained tetrahedralizations as an adaptive grid to ray trace triangle scenes.



3.2 n-ary Bounding Volume Hierarchies

In this section we will present a novel acceleration structures for accelerated ray tracing of incoherent rays. Bandwidth problems are an important issue [Wald 07a] for efficient streaming processing. We achieve a small memory consumption, reducing the overall bandwidth, by flattening the hierarchy and favoring larger leaves which also allow for efficient streaming intersection of the primitives. We use axis-aligned bounding boxes, stored as a minimum and a maximum corner point, for bounding volumes. A detailed description of the node memory layout is given in Listing 3.1.

As the SIMD width is $n = 4$ in most current processors, the discussed implementation focuses on this kind of architecture. Each node of the hierarchy has four children and the triangle intersection always intersects four triangles at once. Thus our new BVH implementation is called QBVH (Quad-BVH) throughout this thesis, although most of the concepts are not limited to $n = 4$.

Construction by Hierarchy Compaction

While the construction of binary hierarchies has got a lot of attention in current research, higher arity hierarchies have seldom been used for ray tracing. Here the splitting choice seems to be considerably more complex because instead of a simple bisection a quadrisection is needed. In this section we show that, even though it may seem more complex, the same principles as for the binary hierarchy construction can be used. Our described construction directly extends also to higher arity hierarchies.

Figure 3.1 illustrates the new construction of a 4-ary QBVH. The same split plane proposals as for the binary case are used but instead of creating nodes after each split the object sets are split again resulting in four disjoint object sets. Now each node does not contain a single bounding volume. Each node directly contains the four bounding volumes of its children.

The $n = 2^k$-ary BVH construction can be seen as collapsing a classical binary hierarchy as seen in Figure 3.1 for the case of $n = 4$. Each k-th level of the hierarchy is kept and the rest discarded. This results in 2^k bounding volumes per node. Figure 3.2 illustrates how a QBVH is constructed, for $k = 2$, approximately halving the memory requirements. This view allows to use the same construction principles used for binary hierarchies to construct 2^k-ary hierarchies.

While using four boxes per node would already be sufficient to enable SIMD processing of the above data structure in the manner of [Chri 06], it is important to store along which planes (perpendicular to the canonical axes x, y, or z) the objects were partitioned. The plane indices of the binary hierarchy are stored as three integers `axis0`, `axis1`, and `axis2`, respectively. This allows exploiting the spatial order of the boxes for a more efficient pruning using the sign of the ray direction during hierarchy traversal. This is described in detail in Section 3.3. Of course, as already mentioned previously storing three full integers for only three split plane positions is very inefficient memory wise. This can be easily packed into smaller data types like a single byte each or even using

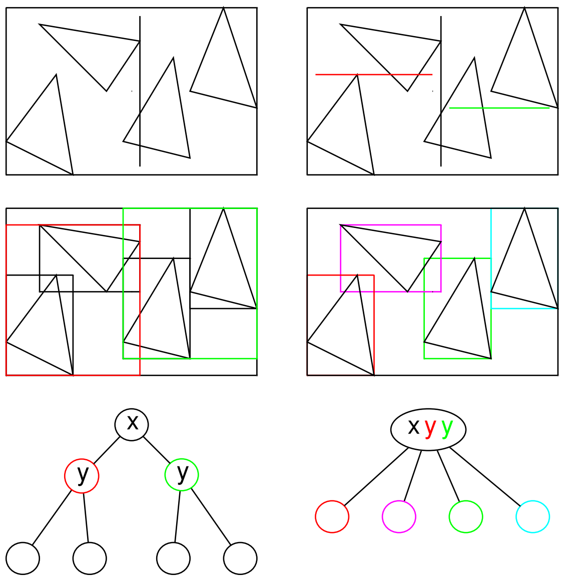


Figure 3.1: Construction of a binary BVH (left), compared to the construction of a 4-ary BVH (QBVH, right). Four children are created for each node. The top row indicates the split plane candidates common to both approaches.

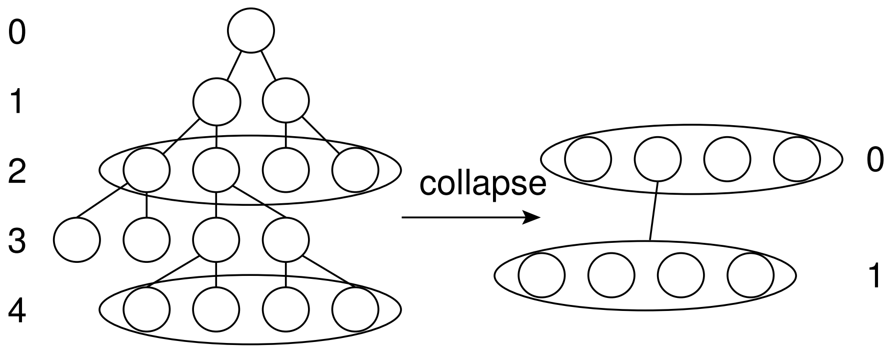


Figure 3.2: Collapsing a binary hierarchy to a QBVH. Classical build methods are used to create a binary hierarchy, which is then collapsed by leaving out intermediate levels.

bit operations and store only 2 bit per plane. But this comes a the cost of an additional unpacking operation. Depending on application this might still be a good choice but for the sake of simplicity we use here the basic layout with 3 full 32 bit integers.

Leaf Criterion

When building acceleration structure hierarchies for ray tracing the recursive process is stopped with a so called termination criterion. While the termination criterion for *k*d-trees is complex and usually involves several user parameters a bounding volume hierarchy can use the number of primitives as a simple termination criterion. For a *k*d-tree this does not work as there are situations where the number of primitives can not be reduced anymore and other criteria like the size of the volume needs to be used.

As mentioned at the beginning, bounding volume hierarchies were usually built down to a single triangle per leaf. Due to the availability of efficient SIMD processing this not only increases the memory of the hierarchy unnecessarily but also reduces the final performance. The decision to create a leaf instead of further constructing a hierarchy is based on several criteria. The simplest criterion is to use a fixed threshold number of primitives. When the number of primitives in the list during construction drops below this value, a leaf is created. We show that this is a very good choice for the QBVH. It is quick to evaluate, allowing for fast hierarchy construction and shows high performance. Using more than a single primitive per leaf has another very important benefit. It reduces the number of empty leaves significantly. Empty leaves are created when there are not enough primitives to perform a split into 4 children (or n in the general case). These empty nodes were considered a huge problem for previous attempts of n-ary BVHs in ray tracing.

3.3 Investigation of n-ary Hierarchy Traversal

The hierarchy traversal is the most performance critical code part in any ray tracing algorithm because it is the most executed path. It can be split into several parts: the leaf intersection, the intersection of a ray with the bounding primitive and the handling of the order of traversal in combination with the stack management needed for a recursive traversal. In this section we investigate the choices for the data structures of a ray and a hierarchy node and how to efficiently intersect a bounding primitive and traverse a given hierarchy. We will then present and discuss several different choices for sorted traversal of a $n-$ary bounding volume hierarchy. For the discussion we will use extensively the concept of SIMD processing that was described in Section 2.4. The intersection with the triangles contained in a leaf is discussed in Section 5.1.

3.3.1 Data Structures for Traversal

The data structure of a single node in a QBVH is illustrated in Listing 3.1. Contrary to the common notion of bounding volume hierarchies, where a node contains a single

```
struct ALIGN16 SIMD_BVH_Node {
  float bbox[2*4*3];
  int child[4];
  int axis0, axis1, axis2;
  int fill;
};
```

Listing 3.1: The full data structure of a QBVH node. It contains 4 axis aligned bounding boxes, 4 child pointer and additional information that may be changed depending on implementation or optimizations. The size of a node is exactly 128 bytes.

volume bounding all children, we directly store each child bounding volume in the parent node. In the case of 4-ary hierarchies this results in four bounding boxes and four child pointers. So already the top level BVH node consists of four bounding boxes. The minimum and maximum of the four bounding volumes are in a layout suitable for direct loading and processing in a SIMD register. The nodes are stored as a single array and the `child[4]` arrays contain the indices into this array. Additionally the split plane axis used during construction are stored. They are needed for a sorted traversal along the ray direction and are explained in Section 3.3. The integer `fill` is used to store additional data for optimizations later on. The data layout of these 4 integers is not fixed and may be changed for example to store the bits needed for the 3 axis in a single char but the total size of a single node should not be changed. The result is a large BVH node with a size of 128 bytes which perfectly matches the caches of modern computer hardware. The 4 bounding boxes are stored in structure-of-arrays (SoA) layout for direct processing in SIMD registers. This access is achieved in C code by a simple cast. The layout and size of a `SIMD_BVH_Node` guarantees that no memory alignment problems occur during an SIMD access when the total array of nodes is allocated with proper alignment. The node data is kept in a linear array of memory locations, so the child pointer can be stored as integer indices instead of using platform-dependent pointers. It is important to note that for a leaf no additional `SIMD_BVH_Node` is created. The leaf bounding box is already stored in the parent and the leaf data can be encoded directly into the corresponding `child` integer.

We use the sign of the child index to encode whether a node is a leaf or an inner node. There exist various other encoding possibilities and it depends on the processor architecture and compiler which one is the most efficient. The most memory efficient is to directly encode the leaf information into the remaining 31 bits. Generally we chose 4 bits for the number of triangles in the leaf and the remaining 27 bits as the start index in the triangle array. Since the triangle intersection is done using SIMD, the number is a multiple of 4. So up to 64 triangles per leaf can be stored and up to 2^{27} triangles can be indexed. Empty leaves are encoded in a special value (`INT_MIN`) so the 4 bits can be used for the full 16×4 triangles. Note that when using bounding volume hierarchies the number of triangles per leaf is easily bounded because forcing another split guarantees

```
typedef struct SIMD_Ray {
  SIMD4_Vec3 o4;   // origin
  SIMD4_Vec3 id4;  // inverse direction
  SIMD4_Vec3 d4;   // direction
  SIMD_float t4;   // t-far value
  int dirSign[3];  // the sign of the direction components (0 or 1)
}
```

Listing 3.2: The ray data structure that is used for the SIMD traversal of a QBVH. It contains a single ray replicated 4 times across SIMD registers to allow for simultaneous intersection of 4 bounding boxes and 4 triangles.

a reduction in the number of triangles.

If the upper bound of 2^{27} is not acceptable for certain applications, a slightly less memory efficient version is created. The full 31 bits are used to index an additional data structure containing the leaf information. Another memory efficient encoding strategy is to store only the start value of the triangle reference array and to mark the last triangle with a negative index in this array.

Traversal Framework

The first important step for optimizing the traversal is to avoid explicit recursive function calls and instead using a locally managed stack. This stack consists only of indices into the array of hierarchy nodes (in our case `SIMD_BVH_Node`). During the traversal loop the yet to be visited nodes are pushed onto this stack. The data structure for a ray used in an efficient SIMD traversal is defined in Listing 3.2. The ray is prepared into a `SIMD_Ray` prior to traversal by replicating the values for the maximum ray distance (`t-far`), the origin, and reciprocal of the direction across SIMD registers. Additionally the signs of the components of the ray direction are stored as integers to allow easy indexing of the near and far bounding box sides. The basic structure of the traversal is now shown in Listing 3.3. Given a ray, the stack-based traversal algorithm starts by simultaneously intersecting the ray with the four bounding boxes contained in the root `SIMD_BVH_Node` using SIMD instructions. The pointers of the children with a non-empty bounding box intersection are sorted and then pushed on the stack. The routine is repeated by popping the next element as long as there is one.

For the bounding box intersection we use a branch-free implementation of the slab test [Will 05]. The 4 boxes are intersected simultaneously with the replicated ray so that in the end four distinct intersection results are computed. The computation using our SIMD abstraction layer is shown in Listing 3.4. The return value of the bounding box intersection function is a SIMD mask containing all ones when the respective box was intersected and zeros otherwise. This allows one to use the result in the traversal below as increment for the stack pointer as shown in the next section.

```
SIMD_Ray ray = prepare_SIMD_Ray(...);
top = 0; stack[top] = root_node;
while (top >= 0) {
  if (stack[top] < 0) {
    intersectLeaf(stack[top], ray);
  } else {
    if (ray intersects at least one box)
      // push the intersected children
      // onto the stack
    }
  }
  top--;
}
```

Listing 3.3: Basic structure of a QBVH traversal. Given a ray the nodes of the hierarchy are intersected recursively. If a leaf is found (encoded by a node pointer that is < 0) the triangles stored in that leaf are intersected.

```
SIMD_int IntersectSIMD_Ray_4Box(ray, node) {
  SIMD_float t0, t1, min, max;
  min = SIMD_SET1(0.0f);
  max = ray->t4;
  for (int i = 0; i < 3; i++) {
    t0 = SIMD_SUB(node->bbox[ray->dirSign[i]], ray->o4[i]);
    t0 = SIMD_MUL(t0, ray->id4[i]);
    t1 = SIMD_SUB(node->bbox[3-ray->dirSign[i]], ray->o4[i]);
    t1 = SIMD_MUL(t1, ray->id4[i]);
    min = SIMD_MAX(min, t0);
    max = SIMD_MIN(max, t1);
  }
  return SIMD_CLE(min, max);
}
```

Listing 3.4: SIMD intersection of an AABB with a replicated ray. This code is an adjustment of the traditional slab test and returns 4 masks each encoding whether the replicated ray intersected the associated box.

3.3.2 Unsorted Traversal

After determining which child boxes should be intersected the associated indices are pushed onto the stack. Listing 3.5 shows the basic traversal of a QBVH using the operations explained so far. It uses an unordered stack push, i.e. the order of the intersected bounding boxes is determined by the hierarchy construction and stays always the same. This is not optimal for tracing nearest intersection rays as the performance is now largely view dependent but this simple and elegant traversal routine is easiest to understand. Later, unordered traversal is again useful when we discuss optimizations concerning occlusion rays. The only part that is left to explain in Listing 3.5 it the stack pushing. The variable `top` is the index to the top of the stack and at the beginning the

```
while (top >= 0) {
  int index = stack[top];
  if (index < 0) {
    intersectLeaf(index, ray);
  } else {
    SIMD_int boxResult = IntersectSIMD_Ray_4Box(ray, nodes[index]);
    stack[top] = pNodes[index].child[0]; top -= boxResult[0];
    stack[top] = pNodes[index].child[1]; top -= boxResult[1];
    stack[top] = pNodes[index].child[2]; top -= boxResult[2];
    stack[top] = pNodes[index].child[3]; top -= boxResult[3];
  }
  top--;
}
```

Listing 3.5: This code extends the traversal framework from Listing 3.3 by an unordered stack push. The result from the SIMD bounding box intersection that was shown in Listing 3.4 is interpreted as an integer to increase the stack pointer either by 1 or by 0.

top element is popped from the stack. After intersecting the associated four bounding boxes this element is no longer needed and can directly be replaced. The direct way for handling the stack push would be the use of a conditional to decide for each of the four components returned in an SIMD register by the bounding box intersection weather to store at `stack[top]` and increase the `top` variable. Instead we always write the child index to the stack at the top position and use the respective component inside the intersection result to increase `top`. This has to be written as a decrement because when an intersection with a bounding box is found the SIMD component is set to all ones which represents a -1 in twos complement. The result of these 4 operations is now a stack where only the intersected bounding boxes are pushed.

3.3.3 Sorted Traversal

As written above, the unordered traversal is not optimal for nearest intersection rays because some rays may traverse the hierarchy virtually from back to front. Especially in scenes with a high overdraw, i.e. many potential triangles can be hit by the ray, this may lead to a lot of unneeded leaf intersections and traversal steps. So to enable an early pruning of the traversal in a QBVH, the nodes are pushed on the stack according to the order determined by the ray direction.

t-far Sorted Traversal

The first possible approach for this sorting is to use the t-parameter of the bounding box intersections. The intersected children are pushed onto the stack in an order that the closest sub-part of the hierarchy is traversed first. Since this sorting is executed in the inner loop the tricky part is to efficiently implement it using SIMD instructions while

```
#define COMP(a, b) {SIMD_float t0, t1;
                    t0 = SIMD_MIN(a, b);
                    t1 = SIMD_MAX(a, b);
                    a = t0; b = t1;}
```

Listing 3.6: The SIMD comparison function that is used used for sorting a single register. It selects the minimum and maximum from the two given registers in place.

correctly handling all special cases. The first part of the algorithm uses branch free SIMD sorting. In [Furt 07] the authors describe a practical way for an implementation using masking and swizzling operations to perform SIMD parallel sorting of arrays. For a QBVH we only need to sort a single SIMD register but their approach directly applies. In the sorting algorithm a `COMP` function is heavily used that is defined as shown in Listing 3.6. Retrieving the t-far parameters from the bounding box intersection is no problem as these values are already computed by the intersection routine. But simply sorting the returned t-far SIMD register is not sufficient for a correct result. The first problem is that not the sorting of the t-parameters is needed but the sorting of the child indices according to the t-parameters. So after sorting the t-register the same reordering has to be done for the mask register and the 4 child pointers. This is not straightforward to do in a branch-free implementation but would be simple when conditionals were used. The second problem that needs to be considered is that there may be several t parameters that are the same. When using axis aligned bounding boxes in an acceleration hierarchy it is not uncommon that several planes coincide. So it often happens that the ray intersection point is the same not only mathematically but also numerically.

The following branch-free algorithm in Listing 3.7 solves both these problems by using knowledge about the floating point representation of numbers. What is shown there is only the inner part of the traversal algorithm. As before it is tried so present the basics of the algorithm as clearly as possible so there is still some room left for optimization by for example more efficient access to single components of an SIMD register. In the first three lines bit masks are defined as constants. The first mask (`amask`) has zeros in the last two significant bits and is used for clearing the last two bits in a given number by using a logical “and”. The second mask contains the numbers 0 to 3 each in one component of an SIMD register. They have to be in the correct order so that the memory access in the last four lines of the algorithm fetches the correct components. The last mask is the inverse of the first one (i.e. the two least significant bits set). The intersection of the ray with the bounding boxes is now done as before but the distance of a potential intersection along the ray is also returned in the register `x0`. If one of the four bounding boxes was not hit, the t-parameter that would be returned is 0. The next line is an important trick to avoid denormalized numbers in the following computations when 0 was returned. On floating point hardware that support the use of denormals they usually mean a huge performance penalty when they are actually used. In our case

```
CONST_INT32_PS(amask, 0xFFFFFFFC, 0xFFFFFFFC, 0xFFFFFFFC, 0xFFFFFFFC);
CONST_INT32_PS(omask, 0x3, 0x2, 0x1, 0x0);
CONST_INT32_PS(emask, 0x3, 0x3, 0x3, 0x3);

SIMD_float x0, x1, x2, x3; // stores the t-far values and are used for sorting
SIMD_int boxResult = IntersectSIMD_Ray_4Box_GetT(&x0, ray, nodes[index]);
x0 = SIMD_ADD(x0, SIMD_SET(1.0f)); // avoid denormals

x0 = SIMD_OR(SIMD_AND(x0, amask), omask); // encode index

x1 = SIMD_MOVE_LH(x1, x0);
COMP(x0, x1);
x0 = SIMD_MOVE_HL(x0, x1);
x1 = SIMD_SHUFFLE(x1, x0, 0x88);
COMP(x0, x1);
x2 = SIMD_MOVE_HL(x2, x1);
x3 = x0;
COMP(x2, x3);
x0 = SIMD_SHUFFLE(x0, x2, 0x13);
x1 = SIMD_MOVE_LH(x1, x3);
x1 = SIMD_SHUFFLE(x1, x0, 0x2d); // x1 is now sorted

SIMD_int sidx = SIMD_AND(x1, emask); // extract the stored indices

stack[top] = pNodes[index].child[sidx[0]]; top-=boxResult[sidx[0]];
stack[top] = pNodes[index].child[sidx[1]]; top-=boxResult[sidx[1]];
stack[top] = pNodes[index].child[sidx[2]]; top-=boxResult[sidx[2]];
stack[top] = pNodes[index].child[sidx[3]]; top-=boxResult[sidx[3]];
```

Listing 3.7: The branch-free implementation of the t-far sorted QBVH traversal using floating point bit operations. The trick is to efficiently retrieve the bounding box indices from the sorted t-far values.

here we only want to compare four floating point numbers so adding 1 does not change the order. The next line encoded the four numbers 0 to 3 into the least significant bits of the mantissa of the four floating point numbers. Here it becomes clear why we need the addition above. This encoding is the solution to both problems mentioned before. First it ensures the uniqueness of the four numbers even if the same t-far values are returned from the bounding box intersection and second it contains the information of the original position without changing the order in a relevant way. The next code block performs the sorting of an SIMD register that is based on the description in [Furt 07]. Afterward we have the sorted result in the register `x1`.

Finally the indices are extracted by using the mask with ones in the two least significant bits. The SIMD register `sidx` now contains in each component a number between 0 and 3 and is used below in the correct order to access the child indices and use the intersection result for the stack increment as before. The result is that we push the bounding boxes in the order of decreasing t-far values onto the stack. The increment of the top pointer is the same as in the unsorted traversal using the fact that -1 is the bit mask of a successful intersection. This guarantees that in the next iteration of the loop the bounding box closest to the ray origin with respect to the t-far value is intersected.

Split-Plane Sorted Traversal

An alternative to sorting along the ray bounding box intersections is to use the original split planes from the hierarchy construction. To generate the 4 children of a QBVH node 3 split planes are created and used during the construction. These same three split planes can then be used with a given ray to estimate in what order the children should be traversed. Figure 3.3 illustrated the ordering in 2d for a single node. To define the correct order it is sufficient to know only the split plane axis. The actual position of the plane is not needed. Having one of the axis (x, y, z) the respective sign of the ray direction determines the order. This is similar to the order a *k*d-tree would define using the same split plane proposals. This has the advantage of traversing the hierarchy the same way as it was originally build and the disadvantage that additional information is needed. But to store the axis only 2 bits are needed. The standard QBVH node from Listing 3.1 has already more than enough free space for this additional information due to of cache line alignment.

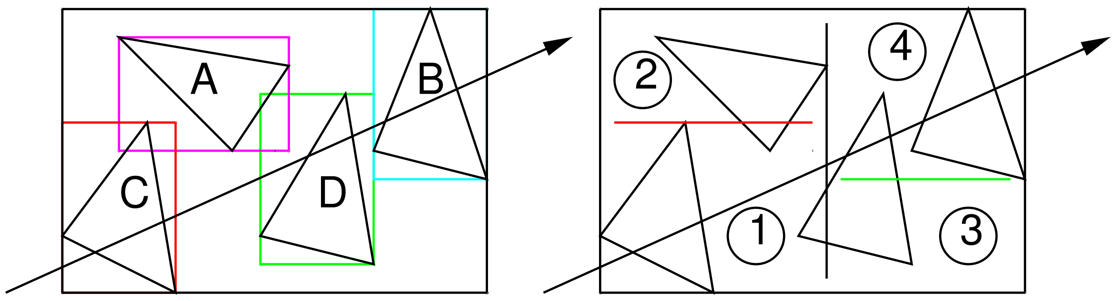


Figure 3.3: Split-Plane sorted QBVH traversal. The dimension of the split plane proposals are kept to sort the child nodes using the ray direction.

The branch free version of this ordering can be implemented in the following way. First the four child pointers are loaded into a SIMD register. Second, a SIMD mask is constructed for each of the three axes stored in the QBVH node based on the sign of the ray direction of that component. Third this mask is used to select either the original value or a shuffled value resulting in a correctly ordered SIMD register. Finally the contents of the SIMD child register are pushed onto the stack in a scalar way as before. The child pointer are copied at the top of the stack and then decrementing the stack top pointer by the corresponding entry of the reordered result mask from the bounding box intersection. For this purpose, this mask has to be ordered again in the same way as the child pointers to push the correct children onto the stack.

A bit surprisingly, an implementation using cascaded branches based on the ray direction signs and split axes is equally fast on an Intel®Core$^{TM}2$ processor in the gen-

```
if (ray.dirSign[pNodes[index].axis0]) {
  if (ray.dirSign[pNodes[index].axis1]) {
    if (boxResult[0]) stack[top++] = pNodes[index].child[0];
    if (boxResult[1]) stack[top++] = pNodes[index].child[1];
  } else {
    if (boxResult[1]) stack[top++] = pNodes[index].child[1];
    if (boxResult[0]) stack[top++] = pNodes[index].child[0];
  }
  if (ray.dirSign[pNodes[index].axis2]) {
    if (boxResult[2]) stack[top++] = pNodes[index].child[2];
    if (boxResult[3]) stack[top++] = pNodes[index].child[3];
  } else {
    if (boxResult[3]) stack[top++] = pNodes[index].child[3];
    if (boxResult[2]) stack[top++] = pNodes[index].child[2];
  }
} else {
  // Same as above but axis1 and
  // axis2 cases in inverse order
}
```

Listing 3.8: Split-Plane sorted QBVH traversal

eral case and even slightly faster when only primary rays are traced. This may be due to the sophisticated branch prediction of this particular processor. As the implementation is also much simpler than the SIMD sorting we show this algorithm here in Listing 3.8. The first decision based on `axis0` chooses which two of the four children should be intersected first. In Figure 3.3, this corresponds to the decision whether to visit (A, C) or (B, D) first. The second and third decision sort each of the child pairs again, resulting in (A, C) or (C, A) and (B, D) or (D, B).

An alternative to the branching implementation is to use a lookup table indexed by the sign bits. While this is more elegant, the shown implementation benefits from the learning capabilities of a branch predictor (at least on Intel CPUs) and the lookup-table implementation was always slightly slower in our experiments.

Comparison

We now compare the unsorted traversal with the split-plane and the t-far sorted traversal. We selected three representative scenes and count the number of bounding box intersections and the number of leaf intersections needed for the final image. As the actual performance of a sorting implementation is largely architecture dependent we do not provide actual timing results. The number of bounding box intersections is directly related to the number of traversal steps and thus a good general performance indicator. The rendering algorithm used for this evaluation is a simple path tracer. This is a good rendering algorithm to measure the sorting as every ray in this kind of algorithm needs to find the closest intersection. It also produces largely incoherent rays after several bounces that shows the problem of unsorted traversal.

Scene	no sorting	t-sorting	split-plane
Sponza	18.5M (3.10M)	16.2M (2.18M)	16.0M (2.18M)
Conference	13.8M (3.39M)	13.3M (3.32M)	12.9M (3.17M)
Interior	28.8M (4.48M)	21.7M (3.55M)	21.2M (3.25M)

Table 3.1: Comparing the three described traversal algorithms. Shown are the average number of million bounding box intersections needed per rendering pass at a resolution of 640×480. In brackets the average number of million leaf intersections is shown. For the statistics and the images 512 passes were rendered. *no sorting* pushes the intersected children in an arbitrary order onto the stack, `t-sorting` uses the algorithm that uses the distance of the bounding box intersections to determine the order and the last one uses the construction split-planes for the sorting.

Figure 3.4 shows the three test scenes and camera perspectives used for the evaluation. As expected from this simple illumination algorithm the resulting images are very noisy. Table 3.1 shows the results of the three traversal methods as the average number of bounding box intersections per pass at a resolution of 640×480.

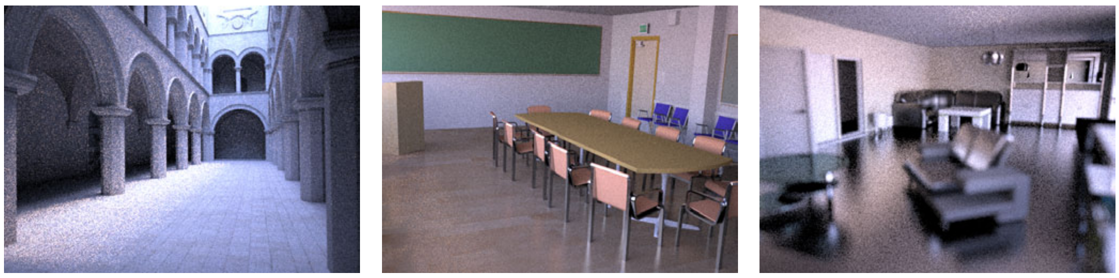


Figure 3.4: The three test scenes used for the comparison of different traversal algorithms. As a simple path tracer is used the resulting images are very noisy.

It is important to note that the number of bounding box intersections for the unsorted version depends on the order of the stack pushing. Even though the path tracing algorithm produces more or less incoherent rays after the first bounce, the primary rays all originate from the camera position. Additionally the secondary rays are not truly random as for example in the Sponza scene a lot more rays go upward instead of downward. It is also interesting that the t-far sorting is in all three cases slightly inferior to the split plane based sorting. The reason is that the split-plane based sorting uses the same order as originally intended by the SAH during construction. The sorting by distance can swap the order in which the ray would naturally pass through the geometry and, thus, intersections are potentially found later.

3.3.4 t-Far Stack for Early Out

The reason why the two sorting methods described previously are beneficial is that on the average traversing the closer sub-hierarchy first results in a close intersection. Since we have a BVH we cannot terminate the ray at the first intersection as in a *k*d-tree but the t-far value of the ray is still used in the bounding box intersection. So a shorter ray is less likely to intersect the farther away boxes and performance improves. When looking at this behavior more closely there is an inherent disadvantage of storing all children directly inside the node. During traversal the decision to traverse a child (i.e. the ray - child bounding box intersection) is done before the closer sub-hierarchy is intersected. This results in a late early-out (the pruning of a sub-hierarchy is delayed and done one traversal step deeper) and more bounding boxes are read from memory and intersected then needed. This problem increases even more with higher arity hierarchies.

A solution to this problem is to introduce a second stack for the t-far values of the bounding box intersections. Depending on the architecture it is better to extend the previous stack to hold two values instead of using a second separate stack. Now, instead of storing only the child pointer of the intersected boxes the t-far value of the bounding box intersection is stored too. Before fetching the memory of a full QBVH node and intersecting the four bounding boxes the current t-far value of the ray is compared to the value on the stack. Only if it is greater or equal, the sub-hierarchy needs to be traversed. This completely avoids unnecessary intersection tests similar to a classic binary BVH.

Scene	no t-stack (tsort)	t-stack (tsort)	no t-stack (ssort)	t-stack (ssort)
Sponza	16.2M (2.18M)	15.1M (1.90M)	16.0M (2.18M)	15.1M (1.94M)
Conference	13.3M (3.32M)	12.9M (3.17M)	12.9M (3.17M)	12.6M (3.04M)
Interior	21.7M (3.55M)	20.0M (3.33M)	21.2M (3.25M)	19.8M (3.09M)

Table 3.2: Comparing the average number of million bounding box intersections per pass for the t-far sorted traversal without and with additional t stack. In brackets are the average number of million leaf intersections. The test setting is the same as in the previous section shown in Figure 3.4.

As can be seen in Table 3.2 this additionally reduces the number of bounding box intersections but on current architectures this does not result in a rendering speed up. Even though the number of bounding box intersections is reduced the final rendering speed is in all cases almost equal on Intel and PowerPC processors (variance smaller than 1%) than just using the simple sorting. The reason is that the additional memory access inside the most executed loop and the additional branch out-weight the decreased number of bounding box intersections.

The implementation of the t-far stack is a nice example of the importance of considering the order of branches with respect to branch prediction. In a straight forward implementation without a given branch hint the first pass of ray tracing the images in Figure 3.4 is about 25% slower then the subsequent passes. Using a branch hint removes this problem completely and additionally results in a slightly faster overall execution time.

To summarize, on current Intel processors and the Cell processor the traversal is faster without this “optimization". But it may pay off for wider SIMD operations as for example for the Larrabee processor [Seil 08]. The wider the SIMD width the more child nodes are pushed onto the stack and thus have to be accessed when no early out can be performed. Additionally the SIMD width also defines the amount of memory that has to be accessed for each node.

3.4 On Demand Construction

On demand hierarchy construction is a simple way of delaying the construction until a ray actually passes through the respective node [Wach 06]. When the actual ray tracing only touches a small part of the scene this method can reduce the amount of memory needed during rendering. There is almost no benefit of on demand construction in the context of a full global illumination algorithm when using the QBVH because in the usual scenes there is the possibility of light paths to touch all parts of the scene. Thus in the course of rendering a final image almost the full hierarchy gets built anyway. Additionally, for this method to work fast, the geometry should be fully loaded into the RAM. As the size of a full QBVH is only a fraction of the actual geometry data there is no real benefit of having only a partly build hierarchy.

Nevertheless, depending on application, it might be useful when only a quick preview of the scene using primary rays is needed or very large scenes that do not fit into the main memory should be rendered.

On demand construction can be added to all traversal methods by simply using an additional bit in the child pointer. If the child pointer is negative and the 30-th bit is set the remaining 29 bits point to a `SIMD_QBVH_Node` structure that is not yet built but contains the necessary information for building it. This information just consists of the start and end index pointing to the not yet used part of the triangle index array. This adds an additional conditional to the hierarchy traversal but only inside the check for a leaf intersection. The performance cost of adding on demand construction to the traversal can be neglected and the previously used hierarchy construction methods can be used without change.

3.5 Results and Discussion

In this chapter we have introduced n-ary bounding volume hierarchies and discussed their efficient memory layout and investigated how to perform efficient ray traversal. We will now present an analysis and comparison of the ray tracing performance using this data structure. These performance results show a clear advantage over previously used acceleration data structures, especially for physically based rendering applications. The actual construction of these hierarchies will be presented in the next chapter along with a detailed analysis of factors that influence the hierarchy quality. For completeness of the comparison we will also incorporate some results of the improvements we will develop in Chapter 5. The ray tracing performance is shown in 3.5.1. In Section 3.5.2 we will analyze the behavior of our acceleration structure under tight memory constraints and Section 3.5.3 will discuss the relation to coherent ray tracing.

For the performance analysis we integrated our new QBVH into a full featured rendering system. This system also contains highly optimized state of the art implementations of a bounding interval hierarchy (BIH) [Wach 06] and a memory-bounded *k*d-tree. A BIH is a variant of abounding volume hierarchy that has a very small memory consumption and can be build quickly. The rendering system uses different unbiased rendering algorithms which allows for a direct comparison of rendering performance on the same hardware and using the same rendering algorithm. All measurements were performed using a single thread on an Intel®Core$^{TM}2$ CPU at 2.33GHz. Note that all measurements were done in a full rendering system where sampling, shading, and texture lookup are a significant part of the rendering time and are not sped up by faster ray tracing. The system only supports the 36 byte triangle layout. Two versions of the QBVH are compared. The first one does not employ the SIMD triangle caching from Section 5.1 and is called ncQBVH. It uses leaves of 8 triangles. The second version uses a SIMD cache size of 128 entries with a leaf size of 16 triangles. In Section 5.1 we evaluate the speed impact of using vertex-index based triangle representation instead of the basic 36 byte per triangle data layout.

3.5.1 Rendering Performance

To estimate the performance in a usual rendering setup we used three scenes of different complexity and left all parameters at their default settings. Figure 3.5 shows the scenes used. The conference room consists of 1M triangles, the interior scene is a complete house with 900K triangles and the power plant model has 12M triangles.

Table 3.3 shows the measured rendering performance from invocation of the renderer to the final image (total time to image: TTI) for each acceleration structure (ACC). Additionally the time for constructing the acceleration structure (ACT), the memory consumed by the acceleration structure (MEM), and the time spent in the ray tracing kernel (RTT) is given. The QBVH uses very little memory while even outperforming the construction times of the BIH. Also note that the ray tracing kernels were all included in the same rendering system for a precise comparison. This made some optimizations in the hierarchy construction impossible—optimizations which would allow building the QBVH for the plant in around 14 seconds. For detailed statistics about the hierarchy quality and ray tracing behavior of the different kernels, see Table 3.4.

Table 3.5 gives an idea of how effective the use of SIMD instructions is, by comparing packet traversal to the QBVH in the setting for which it has not been designed: casting coherent primary rays only.

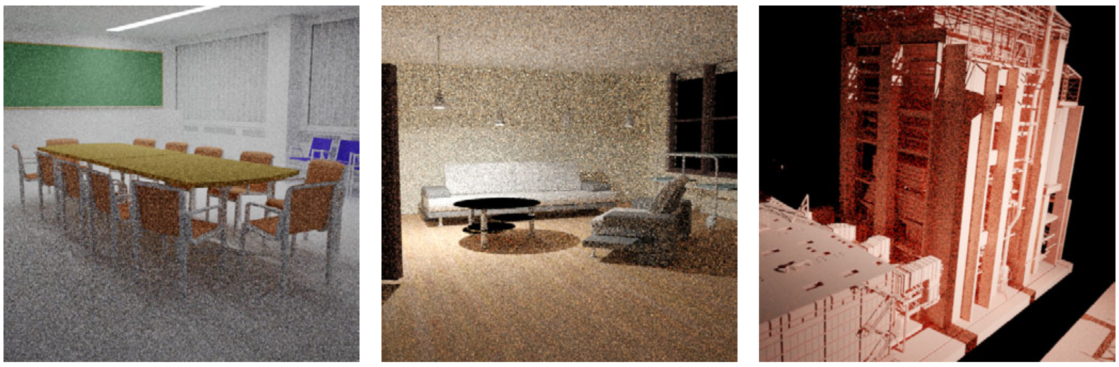


Figure 3.5: The three scenes (Conference, Interior, Power Plant) used for comparison of the QBVH against BIH and a *k*d-tree. The conference room and interior were rendered with a bi-directional path tracer using 8 bi-directional samples per pixel. The power plant was rendered with a path tracer and 32 paths per pixel.

3.5.2 Memory Bound Performance

Memory consumption is an important part of any rendering system. All three acceleration structures allow to a-priori bound the memory that is used. This was discussed for example in [Wach 07]. It allows to specify a fixed amount of memory that should not be exceeded. While the basic idea originates from the need to find good termination criteria for *k*d-tree construction it is also well suited for BVH constructions when the available memory is limited. For bounding volume hierarchies exists an inherent memory limit as never more than $2n - 1$ nodes can be created. Table 3.6 shows the performance of each acceleration structure with bounded memory. The same scenes and rendering settings as in the previous section are used. The *k*d-tree has serious problems with tight memory bounds (note that a build of the conference room with 5 MB was impossible to render in acceptable time) while especially the QBVH reacts robustly to limiting the memory.

3.5.3 Explicit Coherence in Ray Tracing

In this chapter we presented a novel acceleration structure for the acceleration of incoherent rays. Starting with work by Wald et. al. [Wald 01, Wald 04a] a lot of ray tracing research has focused on increasing the ray tracing performance by exploiting coherence. Recent results can also be found in [Bent 06]. They introduced the use of SIMD ray packets that trace 4 rays at the same time through a given acceleration structure. For this to work efficiently, rays need to have a similar origin and direction. If this is given, the traversal amortizes the additional cost as many of the memory accesses need to be done only once per packet. Depending on the amount of coherence even

Scene	MEM	ACC	ACT	TTI	RTT
Conf.	32 MB	*k*d	2.1s	113.7s	80.5s
(1M)	23 MB	BIH	1.4s	139.3s	104.7s
	24 MB	ncQBVH	1.5s	95.7s	63.6s
	16 MB	QBVH	1.2s	85.0s	53.8s
Interior	27 MB	*k*d	1.8s	107.3s	77.7s
(0.9M)	21 MB	BIH	1.4s	131.1s	101.4s
	22 MB	ncQBVH	1.5s	85.5s	56.3s
	14 MB	QBVH	1.1s	81.1s	52.0s
Plant	392 MB	*k*d	30.0s	118.5s	97.3s
(12M)	206 MB	BIH	21.5s	143.6s	112.4s
	310 MB	ncQBVH	23.9s	104.1s	78.0s
	185 MB	QBVH	19.1s	94.7s	56.1s

Table 3.3: Performance comparison of BIH, *k*d-tree, non-cached QBVH (ncQBVH) and cached QBVH. The cached QBVH is explained in Chapter 5. The scenes are shown in Figure 3.5. The number in brackets are the number of triangles in millions. MEM is the memory needed by the acceleration structure, ACC the type of acceleration structure used. ACT is the hierarchy construction time, TTI the total time to image and RTT the pure ray tracing time.

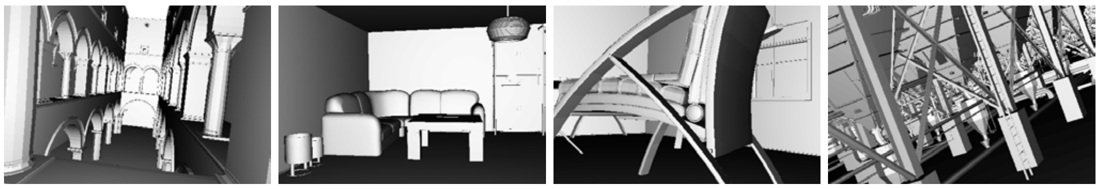
larger packets can be beneficial. For the ray traversal and intersection to give the correct results a modification to the standard traversal is needed and additional masks might be needed to mark rays that intersected already. In [Wald 06b] the authors develop an efficient coherent ray traversal algorithm for grids. For bounding volume hierarchies, large ray packets are explored for example in [Wald 07b] and for the use in *k*d-trees [Resh 05, Resh 07] presents nice algorithms. A good overview of the common construction and traversal algorithms up to 2007 can be found in the state of the art report on animated ray tracing [Wald 07c].

For these algorithms to give a speedup, coherent rays need to be found. This is simple for primary rays and a pinhole camera model as they have all the same origin, and neighboring pixels in the camera give roughly the same direction. Exploiting this coherence gives a speedup of about $2x$ to $3x$ depending on scene and implementation. But it is important to note that primary rays are not very important for ray tracing applications as the interesting effects come from secondary rays and the result of primary rays can be executed much faster by using a rasterization algorithm. For shadow rays there can be coherence when the rays are traced towards point light sources or compact area light sources. In [Over 08] the authors construct coherent rays for a Whitted-style ray tracer. For the kind of image synthesis algorithms we are interested in this thesis this coherence is very hard to find. The primary rays are only a small fraction of the computation needed for the final image and since we consider physically realistic scenes we have no simple point light sources but arbitrary modeled light sources. Additionally the Monte Carlo and Quasi-Monte Carlo algorithms we use are explicitly designed

Scene	ACC	inner nodes	leaves	tris/leaf	tris/ray	inner/ray	leaves/ray
Conf.	*k*d	371018	384349	13.2	16.2	40.7	6.36
(1M)	BIH	1066634	557709	1.8	7.2	71.2	3.78
	QBVH	44704	133323	7.4	20.4	11.8	2.29
Interior	*k*d	429294	450133	9.0	20.6	50.0	4.89
(0.9M)	BIH	931393	510266	1.8	8.4	101.1	4.58
	QBVH	41569	123144	7.4	21.7	18.1	2.48
Plant	*k*d	4379594	4801842	14.9	29.0	59.1	4.28
(12M)	BIH	10923900	6717548	1.9	10.5	141.4	5.01
	QBVH	548908	1603848	7.9	32.6	24.5	2.93

Table 3.4: Detailed statistics for different acceleration kernels (ACC). This table shows, for the respective kernel and scene, in this order: the total number of inner nodes, the total number of non-empty leaf nodes, the average number of triangles per leaf node, the average number of intersected triangles per ray, the average number of traversed inner nodes per ray, and the average number of intersected leaf nodes per ray. The measurements have been made under the same conditions as in Table 3.3.

to increase the incoherence because this leads to faster image convergence. For this reason we follow a different approach for accelerating ray tracing and increase the performance by designing an algorithm designed from ground up to handle incoherent rays instead of searching for coherence. Other researchers also recognized the need to develop algorithms for incoherent rays. For example in [Wald 08] the authors build a SIMD data structure for the incoherent rays that arise in a path tracer.



Kernel	Sponza	Interior 1	Interior 2	Plant
2x2 BIH	16.3	16.6	10.9	3.6
BIH	6.4	6.6	4.1	1.4
QBVH	14.1	11.9	9.03	2.4

Table 3.5: Comparing packet tracing in a BIH with mono ray tracing in a QBVH. The table shows frames per second for 2×2 SSE packets in a BIH vs mono ray BIH vs. mono ray QBVH on an Intel Core 2 Duo (480×320). In the packet version, flat shading and ray creation is also done using SIMD. Most of the performance benefit of the packets is due to coherent memory access, which is already lost for scene Interior 2, where QBVH and packet traversal perform about the same.

Conference

MEM	16 MB				5 MB			
ACC	*kd*	BIH	ncQBVH	QBVH	*kd*	BIH	ncQBVH	QBVH
ACT	1.3s	1.2s	1.2s	1.2s	—	0.7s	0.7s	0.7s
TTI	119s	138s	99s	85s	—	220s	224s	120s
RTT	89s	106s	86s	54s	—	189s	190s	91s

Interior

MEM	16 MB				8 MB			
ACT	1.3s	1.2s	1.1s	1.1s	0.8s	0.9s	0.9s	0.9s
TTI	113s	130s	91s	81s	250s	139s	118s	88s
RTT	84s	101s	63s	56s	219s	110s	90s	59s

Power Plant

MEM	185 MB				100 MB			
ACT	21s	18s	19s	19s	14s	15s	16s	16s
TTI	129s	139s	105s	95s	369s	145s	117s	95s
RTT	85s	98s	78s	56s	330s	107s	90s	68s

Table 3.6: The performance of the different acceleration structures when the available memory is a-priori bounded (MEM). ACC is the type of acceleration structure used. The timings are given for the acceleration structure construction time (ACT), the total time to image (TTI), and the time spend in the ray tracing core (RTT).

4 Efficiency of Hierarchies

In Chapter 3 we introduced a novel acceleration structure for incoherent rays. This acceleration structure used a given scene database without modifications. In this Chapter we will consider the quality and ray tracing performance of the acceleration data structure. The traversal of the hierarchy is independent of the construction but the way a hierarchy is build can have a major impact on the final rendering performance. Building acceleration hierarchies is a complex problem [Havr 01] and the usual approach is using a greedy algorithm with a splitting heuristic. Section 4.1 will provide an overview of the commonly used heuristics to build a bounding volume hierarchy and in Section 4.2 we will discuss the behavior of our QBVH for the split choices. A general problem with bounding volume hierarchy construction is node overlap, especially when large triangles are in the scene. Section 4.3 investigates this problem and provides a memory efficient solution to improve the quality of the build hierarchies by splitting large triangles that would result in inefficient bounding volumes. By identifying and splitting the inefficient triangles in a numerical robust way this method can dramatically improve the ray tracing performance. The splitting is performed without actually modifying the triangle database and thus integrates well into a full rendering system. In Section 4.4 the triangle intersection is considered again but this time not for speed improvements but to extend the algorithm to guarantee no holes during intersection computation. While this slows down the computation it is necessary for some physical simulations where under no circumstances cracks should occur. Finally, in Section 4.5 we will investigate BVH construction for animated scenes. There, the BVH is used as a simple clustering algorithm to reduce the overlap when rigid objects are animated.

4.1 Overview of Hierarchy Construction Heuristics

The quality of the hierarchy built for the scene has a major impact on the final ray tracing speed. When the acceleration structure is built, it defines for each ray the number of traversal steps and primitive intersections needed to obtain the final hitpoint. Several methods have been studied in detail by Havran in [Havr 01] and an overview of different acceleration structures was given in Section 3.1. Experience has shown that a good construction algorithm is connected to the kind of acceleration structure used. So in this section some of the approaches for binary hierarchy construction are discussed and later their influence on the QBVH is analyzed.

Since in almost all applications the rays that should be traversed through the scene are mostly unknown the construction of the acceleration structure needs to use some form of heuristic. Additionally, the speed of constructing the acceleration structure is important. Sophisticated heuristics that allow for backtracking during the build have such a high algorithmic complexity that it is infeasible to apply them to realistic scenes with millions of primitives.

Binary hierarchies are constructed by recursively finding a bisection of the primitives. This bisection process for a binary BVH is illustrated in Figure 4.1. All, in practice

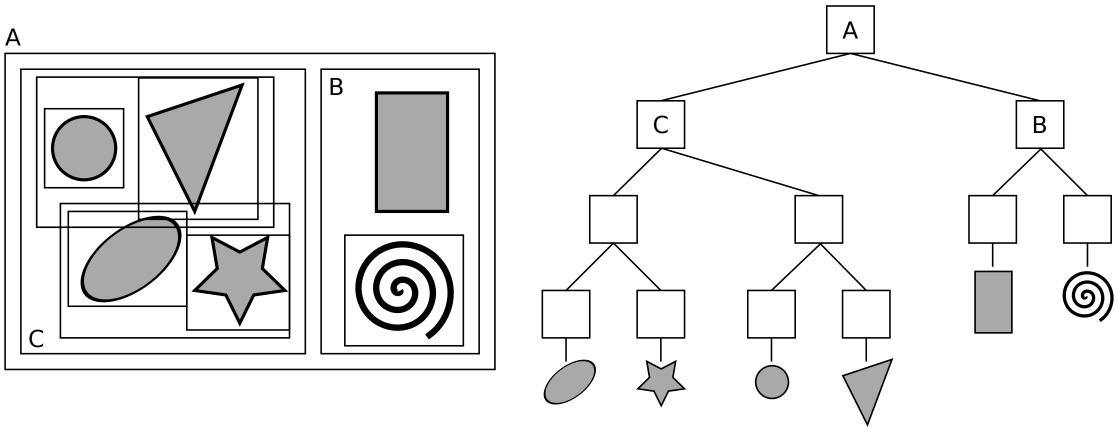


Figure 4.1: Illustration of the principles of a bounding volume hierarchy. The left figure shows a 2d scene consisting of 6 different objects. The list of these objects is recursively subdivided into sub-lists and a new bounding volume of each list is computed. As bounding volumes axis aligned boxes are used. Note that for illustration purposes the bounding volumes are not drawn as tightly as they should be in a real implementation. The figure on the right shows the resulting hierarchy. Each node contains the associated bounding box.

usable, construction heuristics are greedy. They find a bisection of the primitives at the current construction level and never change it again. For the following discussion we assume that the bounding volumes used are 3d intervals (axis aligned boxes) but the described split heuristics usually have an obvious equivalent when different bounding volumes are used.

Bounding volume hierarchies allow for an arbitrary bisection of the primitives and still produce the correct result. But for high performance the resulting bisection should be able to tightly bound close primitives and have as little overlap as possible. The first of three different split heuristics we want to discuss, is the median split.

4.1.1 Median Split

Median split is a simple approach for building that results in a well balanced hierarchy. The primitives are sorted along one axis using for example their center points and then the list is split at the median. The axis is chosen in the direction where the enclosing bounding volume has the largest extent.

4.1.2 Center Split

A more flexible approach than median splitting is to choose a plane intersecting the current node and use it to partition the primitives. Each primitive is classified as lying on the positive or the negative side of the plane (sometimes also called the left and the right side of the plane). For bounding volume hierarchies, primitives intersecting the plane need to be moved uniquely to one side. Heuristics to do this are using the center of the primitive for classification or the amount of overlap (measured as the distance of the farthest point on the primitive to the plane). Bounding volume hierarchies are originally defined so that the leaves contain only a single primitive. In contrast to space partitioning acceleration structures it is always possible to construct such hierarchies. But later we show that this is an inefficient choice (considering performance and memory) for the triangle based scenes used for rendering. But for the discussion of the splitting plane selection we can assume a single primitive per leaf. Using this assumption the algorithm to choose the splitting plane completely defines the resulting hierarchy for a given scene. The most used selection algorithms are based on selecting an axis aligned plane and classifying the triangles based on their center.

It is important to note that choosing a splitting plane and a left-right criterion for a primitive does not guarantee to result in a bisection of the primitives. For example using the center of triangles it can happen that all triangles are classified as lying on the left side. This happens especially in the deeper levels of the hierarchy where there are less primitives to partition. An implementation of a construction algorithm has to detect these cases and provide an appropriate solution. When the hierarchy should be built down to one primitive per leaf the list can be split arbitrarily into two halves. An improved solution is to try other splitting planes (for example trying the other 2 axis when using axis aligned split planes) or to create a leaf containing all the remaining primitives.

4.1.3 Surface Area Heuristic (SAH)

Using the surface area heuristic (SAH) to guide the construction of *k*d-trees as well as for BVH is commonly accepted to be the best choice when final ray tracing performance is the most important goal. The construction is more expensive than other methods but for most scenes the resulting hierarchies perform (from a traversal efficiency view point) better than any other known construction method. Even though a *k*d-tree and a BVH have a completely different notion of what splitting means during construction almost the same heuristic can be used to guide the choice. Using a heuristic instead of searching the optimal split is preferable as in [Szir 98, Berg 92] it is shown that the construction of optimal hierarchies takes in the worst case $O(n^{4+\epsilon})$ processing time and memory when a logarithmic search is desired.

While the original SAH was developed for bottom up construction [Gold 87] it has been common practice to use it for the faster top down construction also. During construction the SAH minimizes greedily (i.e. for a single decision) a cost metric of a partition $P = L, R$ described as:

$$C(P) = C_T + C_I \frac{(A(L)|L| + A(R)|R|)}{A(P)}$$

C_T is a constant to estimate the cost of traversing the node and C_I the cost of intersecting the objects. $A(\cdot)$ is the surface area of the object list. During *k*d-tree construction in each split step a plane has to be chosen. When a plane is chosen the partitions L and R can be created by sorting the primitives into one or the other list depending on which side of the plane they are. Objects that overlap the split plane are inserted into both partitions. In [Havr 01] it is shown that there exist only $6n$ plane positions for n primitives that need to be considered and in [Wald 06a] it is shown how these can be efficiently tested.

To find the minimum of this function during a BVH build an implementation would have to consider all $2^n - 2$ partitions (the two partitions where either $|L| = 0$ of $|R| = 0$ should be ignored). This is impractical even for small scenes. The usual solution to this problem is using a single axis aligned plane and partitioning the list of triangles based on their centroids as it is done during *k*d-tree SAH build. Now there are only $n - 1$ possible partitions along one axis (choosing the split plane through each centroid). Checking $3(n - 1)$ is now feasible but still slow especially at the higher levels of the hierarchy construction. To evaluate the cost function for a given split plane L and R still has to be computed by scanning over all primitives in P and deciding in what partition it falls. These are $3(n - 1) * n$ operations. Computing the area of the bounding volume requires another iteration over all contained triangles which is still quite slow. Thus many research has been done to improve the construction performance but still creating at least approximately the same quality hierarchies. In [Hunt 06] the authors develop an approximation of the SAH function that requires only few split plane samples. Parallel construction is discussed for example in [Shev 07]. They also introduce the min-max binning approximation to quickly evaluate the SAH cost function. In the next section we will describe our implementation of the SAH cost metric optimized for our novel acceleration structure in more detail.

4.2 Split Heuristics for the QBVH

In the previous section we discussed the common split heuristics for bounding volume hierarchies. We also showed that the construction of a QBVH can use any heuristic that was used to construct binary hierarchies. In this section we will analyze the behavior of the QBVH when a center-split is used and compare it to an SAH split. As can be expected, the behavior is similar to classical bounding volume hierarchies performance wise. In Section 4.2.1 we will provide a visual comparison of these two split heuristics and in Section 4.2.2 we will provide some experimental investigation into the quality of the SAH for physically based rendering.

Center-Split QBVH

Building a hierarchy using the center-split heuristic is one of the simplest and fastest construction method but the resulting hierarchies only work for some scenes well. To perform the center-split heuristic in a BVH construction, first the axis aligned bounding box for all objects is computed. The dimension in which the bounding box is largest is chosen as split axis for the current list of objects. The split plane is chosen then in the middle of this bounding box. For overlapping primitives there exist various possibilities to handle them all with slightly different performance results. We chose to put each primitive on the side where the overlap is larger thus minimizing the total overlap of the bounding boxes. For the center-split heuristic this choice is arbitrary and no consistent performance improvement could be found by choosing other decisions. Later in this section we will compare visually the results of this simple heuristic with our SAH build described below. This gives a better understanding of how the SAH construct hierarchies for such scenes. The performance of the center-split constructed hierarchies with a direct comparison the SAH built hierarchies can be found in Section 4.3.4.

SAH-Split QBVH

The fundamentals of the surface area heuristic (SAH) were already described in Section 4.1. Here we discuss the specifics on how we create the SAH hierarchies for a QBVH. The performance results and the comparison to other recent high performance data structures can be found in Section 3.5. The comparison to the center-split is found below.

In our implementation we used recent results [Shev 07, Wald 07a] to construct good quality hierarchies with fast construction times. We use min-max binning to approximate the SAH. This implementation is vectorized by the compiler and efficiently uses SIMD instructions. As proposed in [Shev 07], only a fraction of the primitives is used for the SAH approximation in higher levels of the hierarchy. Further speedup is achieved by pre-computing the bounding boxes of the triangles in a SIMD layout, overwriting the triangle data. This way, the implementation can benefit from SIMD operations during construction without using additional memory. After construction, the triangle data is reloaded from disk. The construction is additionally sped up by allowing more primitives per leaf, resulting in flatter hierarchies. In all measurements, when not otherwise noted, we create a leaf whenever the primitive count drops below 17. Using this implementation, it is possible to get very fast construction times that are even comparable to the BIH [Wach 06]. The performance results of this construction were already presented in 3.5.

4.2.1 Visual Comparison of Center-Split and SAH-Split

We now provide a visual comparison of the quality of center-split QBVH vs. SAH-split QBVH. The results can be seen in Figure 4.2. The images show the normalized number of bounding box that can be seen per pixel. For a better spatial understanding of the hierarchy it is shown as a projection from the three major axes. A dark color means that there are many bounding boxes that get potentially intersected when a ray is traversed. What can be directly observed, when the projection of the center-split heuristic is compared to the SAH-split, is that the images are much brighter meaning there is less overlap in an SAH build hierarchy. This means the SAH creates a more balanced hierarchy with respect to spatial distribution of bounding boxes. This is especially notable in the Sponza scene.

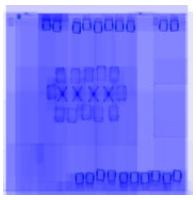
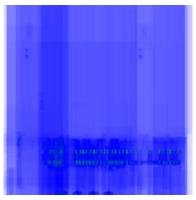
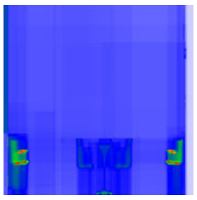
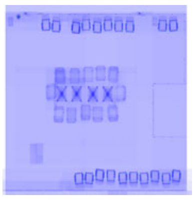
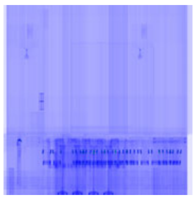
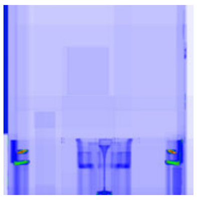
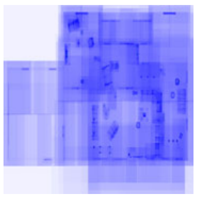
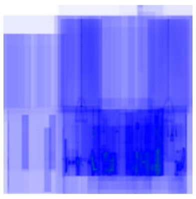
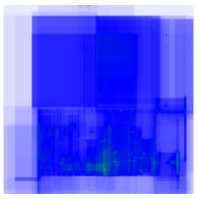
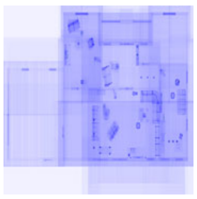
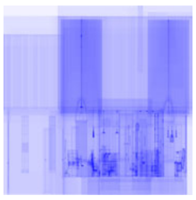
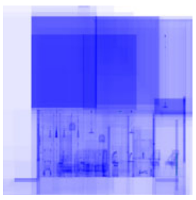
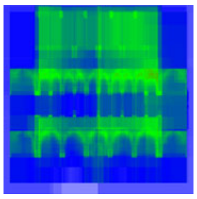
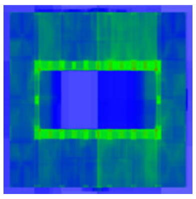
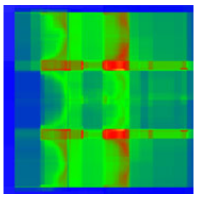
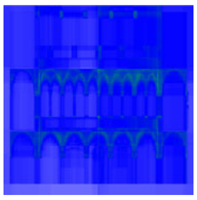
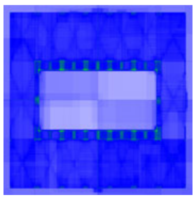
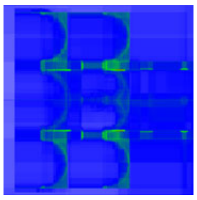
4.2.2 Validity of the SAH for Physically Based Rendering

The SAH is a simple greedy heuristic and it is impossible for complex scenes to verify how close to the optimum the resulting hierarchy is since searching an optimal hierarchy for millions of triangles is impossible. Nevertheless the results are quite good when compared to other heuristics. But there is still a lot of argument if it is the right choice. In this section we justify the SAH from an experimental viewpoint for physically based rendering. The SAH is based on evaluating and greedily minimizing the surface area metric in each step. We will thus experimentally analyze the assumptions of the SAH by computing the surface area metric (SAM) for the whole hierarchy and relate it to the final rendering performance. The SAM as we use it is simply

$$SAM = \sum_{T_i} A(T_i) + c_T \sum_{T_l} A(T_l)|T_l|$$

Where T_i are the interior nodes, c_T the cost of intersecting a triangle relative to intersecting a inner node, T_l the leaf nodes and $|T_l|$ the number of triangles in this leaf node. Now we take the hierarchy generated by the SAH as baseline and use random movements of the split plane to create hierarchies of larger SAM. We vary the plane position by movements from 0% to 100% where 100% is a completely random split plane. This leads to hierarchies that result usually in worse rendering performance. The assumption is now that the change in the SAM should also reflect an equivalent change in the rendering performance. Figure 4.3 shows the graphs of the results for three test scenes. The test scenes that were used are Sponza, Conference, and Interior which can be seen in Figure 3.4. To create the statistics a path tracer with 8 paths at a resolution of 800×450 pixels was used for each hierarchy. The graphs show the relative degradation of the SAM and compares it to the relative increase of the number of bounding box intersections. Without any randomness both, the relative SAM and the relative number of bounding box intersections are 100%. Two interesting observations can be made in these graphs. The first is that the SAM indeed is a good quality index for the expected final rendering performance. In almost all cases, an increased SAM results in a similarly increased number of bounding box intersections. It can also be seen that the factor depends on the scene that is tested. While in the Sponza scene the relative SAM is almost always greater than the relative number of bounding box intersections the situation is opposite in the Interior scene. The next observation is related to the heuristic nature of the hierarchy construction. The choice of random movements can result in better hierarchies but still, none of the degraded hierarchies resulted in better performance than using the full SAH.

These results show experimentally that the SAM is a good choice for incoherent rays as the behavior is similar in all three test cases. But the variation in the performance when random perturbations are introduced let us assume that there might be even better heuristics. It remains an open question if there are better greedy heuristics that avoid backtracking during hierarchy construction and consistently create better hierarchies than the SAM.

Hierarchy	XY-Plane	XZ-Plane	YZ-Plane
Conference MID			
Conference SAH			
Interior MID			
Interior SAH			
Sponza MID			
Sponza SAH			

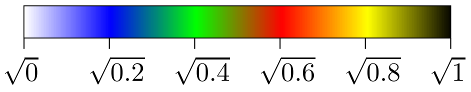


Figure 4.2: Visual comparison of the hierarchy quality for three different scenes when using the center-split (MID) or the SAH split heuristic.

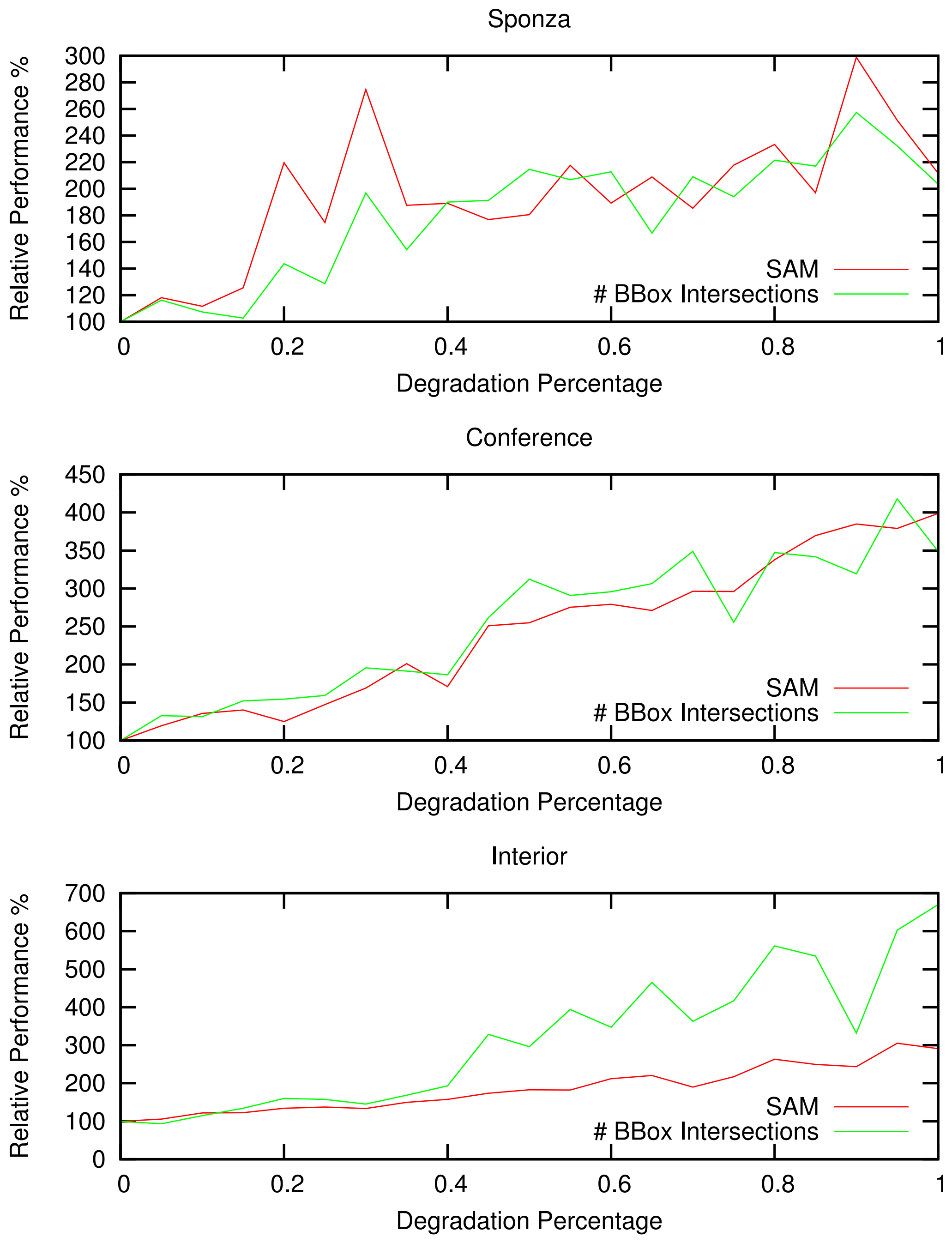


Figure 4.3: These three graphs show the relative increase in number of bounding box intersections compared to the relative increase of the surface area metric (SAM) when random split plane perturbations are used. On the x-axis the random movement of each split plane increases from 0 (normal SAH) to 1 which is a completely random split plane choice. The three test scenes are the same as shown in Figure 3.4.

4.3 More Efficient Hierarchies by Triangle Subdivision

An important advantage of a BVH as a ray tracing acceleration structure is the small memory footprint compared to a standard *k*d-tree, because each object is only referenced once. Even though a BVH can be as fast as a *k*d-tree, there are scenes where the resulting performance is far worse. This is a direct consequence of the principle that every object should only be referenced once: Bounding boxes that contain large amounts of empty space increase the number of ray object intersections. This problem becomes especially apparent for axis-aligned boxes enclosing non-axis-aligned geometry as it results from e.g. rotation: A triangle with a normal along one of the canonical axes has a zero volume axis-aligned bounding box, while any other orientation increases the volume and causes the triangle to be tested against more rays although the probability of hitting the triangle remains the same. This problem is illustrated in Figure 4.4.

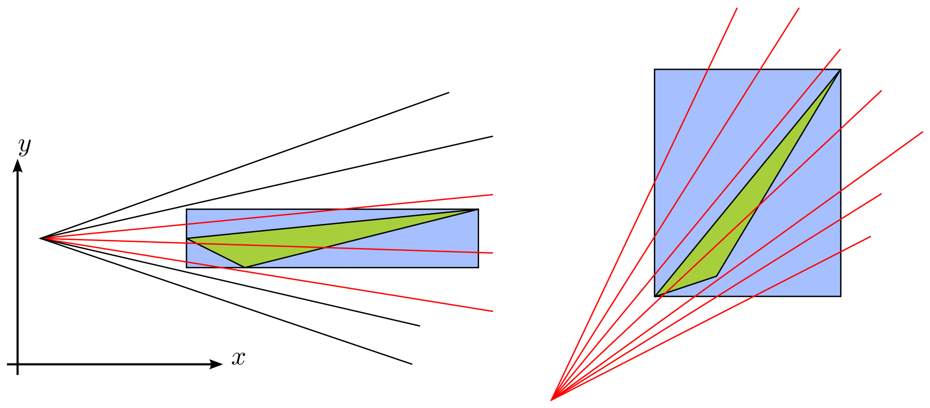


Figure 4.4: 2d illustration of the coordinate system dependence of axis aligned bounding boxes around triangles. On the left side the AABB of the triangle is almost flat and of the seven rays shot, only three intersect the box. On the right side the scene and the rays are rotated in a way that the same image would be computed. But here all seven rays intersect the box and thus are tested for intersection with the triangle.

In Section 4.3.1 we introduce the Edge Volume Heuristic (EVH) that provides a memory efficient way of improving the quality (and thus ray tracing performance) of a BVH by modifying the scene database. In Section 4.3.2 we develop an additional subdivision heuristic that is based on an area measure similar to what was done in [Erns 07]. The main difference is that we still use spatial subdivision instead of triangle splitting. This heuristic is not designed especially towards a hierarchy construction algorithm and needs in many cases significantly more memory then the EVH. But it results in triangles of roughly the same size. Section 4.3.3 introduces leaf compaction that improves the performance of hierarchies constructed with these triangle subdivision methods. In Section 4.3.4 we use the triangle subdivision methods to compare the influence of different hierarchy construction algorithms when the average size of triangles changes.

4.3.1 Edge Volume Heuristic

In the context of ray tracing with bounding volume hierarchies the overlap problem has been investigated in [Erns 07] and the presented approach was called "early split clipping". The authors present a triangle splitting method based on the surface area of the triangle bounding volume and use an axis-aligned plane to split a triangle into three triangles (resulting from one triangle and one quadrangle). The approach of early split clipping reduces empty space contained in bounding volumes, which in addition reduces overlap and thus improves overall performance. However, considering surface area also causes triangles to be split that are already tightly packed in a bounding volume (e.g. larger triangles with normals parallel to the canonical axes). In addition the splitting threshold is based on user experimentation per scene and triangles are split even when no speedup can be achieved. While one might argue that the same benefits can be obtained using a *k*d-tree [Wald 04a], especially when bounding the memory of the *k*d-tree construction [Wach 07], both approaches have to clip triangles against planes, which is a numerically tricky and costly operation.

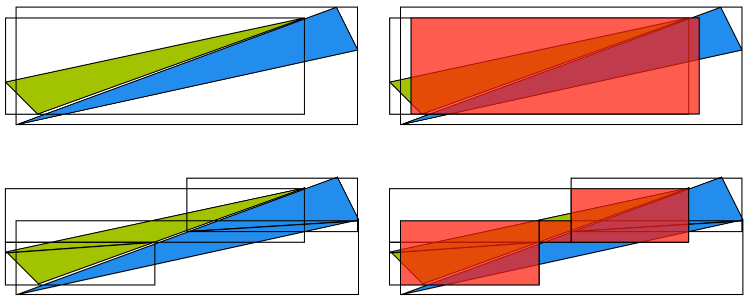


Figure 4.5: The top row of this image shows the problem of adjacent triangles and the overlap of the inefficient bounding boxes. In the bottom row each triangle was subdivided into two along the middle of the longest side.

In the following we address some of these disadvantages by introducing a new, numerically robust triangle subdivision (in contrast to splitting) algorithm that only subdivides triangles where required thus keeping the biggest advantage of bounding volume hierarchies: The small memory footprint. Some of this research is published in [Damm 08b]. This subdivision algorithm is based on a volume measure. The heuristic is based on the observation that not all large triangles affect the performance of a SAH [Gold 87] based BVH significantly but only the one that cannot fit tightly into a bounding box. That subdivision of triangles reduces the overlap of adjacent bounding boxes is illustrated in Figure 4.5. In addition we provide a simple heuristic to automatically choose the level of subdivision. While designing the algorithm we tried to achieve the following goals: First was to subdivide as few triangles as possible to keep the memory footprint low. This includes also the need to avoid subdividing triangles where no speedup can be achieved. Second was the need for an acceptable automatic way to determine the number of subdivision. This is especially important when animations are rendered and the scene configuration changes over time. Finally it should be guaranteed that the resulting mesh is still watertight and no numerical problems are introduced.

The new heuristic is called *Edge Volume Heuristic* (EVH) and measures the tightness of the bounding box of each triangle edge and subdivides the triangle until a certain threshold ϵ_v (see below) is met.

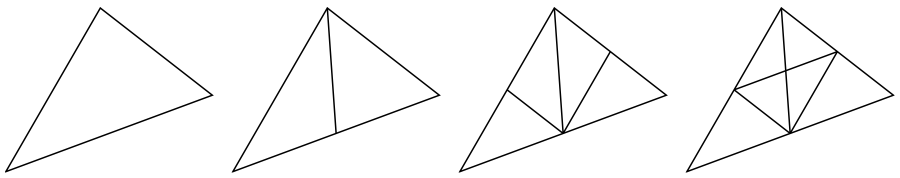


Figure 4.6: Recursive subdivision of a triangle along the longest edges. Although only edges are considered unaware of topology, the tessellation remains watertight if edge subdivision is symmetric.

For each edge of a triangle the subdivision algorithm determines its axis-aligned bounding box. The volume of the largest of the three boxes is compared to a volume threshold ϵ_v. If the volume is larger than the threshold the triangle is subdivided in the middle of this edge, which is easily implemented in a numerically robust manner. The procedure is repeated for the two new triangles until it terminates (see the illustration in Figure 4.6).

The heuristic guarantees that without any knowledge of topology, identical operations will be performed on shared edges. Consequently the resulting bounding boxes fit without gaps, which overcomes precision issues of clipping and watertight meshes will remain watertight after subdivision. This is true for any symmetric subdivision of edges [More 01].

Note that using a heuristic not based on edges, like e.g. bounding box surface area, cannot guarantee watertight subdivision: Cracks can occur, because shared edges are not necessarily subdivided in an identical way. In addition a surface area criterion would divide large triangles regardless of the tightness of the bounding box, which is not memory efficient. In fact the edge volume heuristic is economical as it only subdivides triangles with very inefficient bounding boxes.

Determining the Edge Volume Threshold

The edge volume threshold

$$\epsilon_v(t) := \frac{V}{2^t}$$

is determined as a fraction of the volume V of the scene bounding box and thus controls the amount of triangles generated by subdivision.

Over a broad range of scenes it turned out that choosing the threshold parameter $t = 14$ as a default value yields good results with respect to increased triangle references and performance. With this threshold value many scenes that already exhibit high ray tracing performance are not subdivided at all or the increase of triangle references is less than 1%. Thus it is safe to rely on a fixed parameter t. But as with any heuristic, specially constructed scenes may break the assumption of course (for example a single thin diagonal triangle). For this kind of scenes the user may have to choose the parameter t by hand. In the results section below the impact of varying t is quantified for different bounding volume hierarchy unfriendly scenes. This simple threshold selection is of course not limited to the edge volume heuristic but can be used (with a different scale) for example for the early split clipping approach.

Implementation

The procedure can be applied either as a pre-process before ray tracing or for on-demand subdivision at the beginning of the hierarchy construction. The first variant is especially useful as it can be used to upgrade any existing ray tracing module without modifying the internals. The pre-process should be implemented directly into the geometry exporter of the modeling application and allows to test the benefit of the subdivision without any changes of the underlying ray tracing system. A drawback of this method is an enormous memory increase for highly detailed scene as not only the triangle numbers increase but also all associated data like texture coordinates and shader information.

The second variant is transparent for the user and just produces a different hierarchy during construction. As written previously pre-computing the bounding boxes of the triangles prior to hierarchy construction already benefits the construction speed. This can now be used to efficiently implement the triangle subdivision. For each triangle the algorithm performs the recursive subdivision procedure. As the bounding volume hierarchy construction only uses the bounding boxes, it is memory-efficient to output the bounding boxes with the original triangle reference instead and in place of the subdivided triangles. As a result the algorithm produces an array of the bounding boxes and two index arrays. The first index array is the same as if each bounding box would contain a distinct triangle and the second array contains the indices to the original triangles. During construction the first array is used as usual to index the bounding boxes and sort along the splitting planes but the second array is reordered in the same way. Finally in the leaves of the hierarchy the indices of the second array are stored. This way it is guaranteed that only the original triangle data is needed when a leaf is intersected. The scan over the triangles is so efficient, that it even pays off, to have a pass that only counts the number of generated triangles, to allocate memory accordingly, and to scan the data again to generate the bounding boxes.

The remaining problematic cases are overlapping bounding boxes that cannot be separated. This problem is ameliorated by the fact that efficient bounding volume hierarchies usually reference more than one triangle per leaf thus grouping some of the overlapping geometry in one box, which reduces the overall overlap.

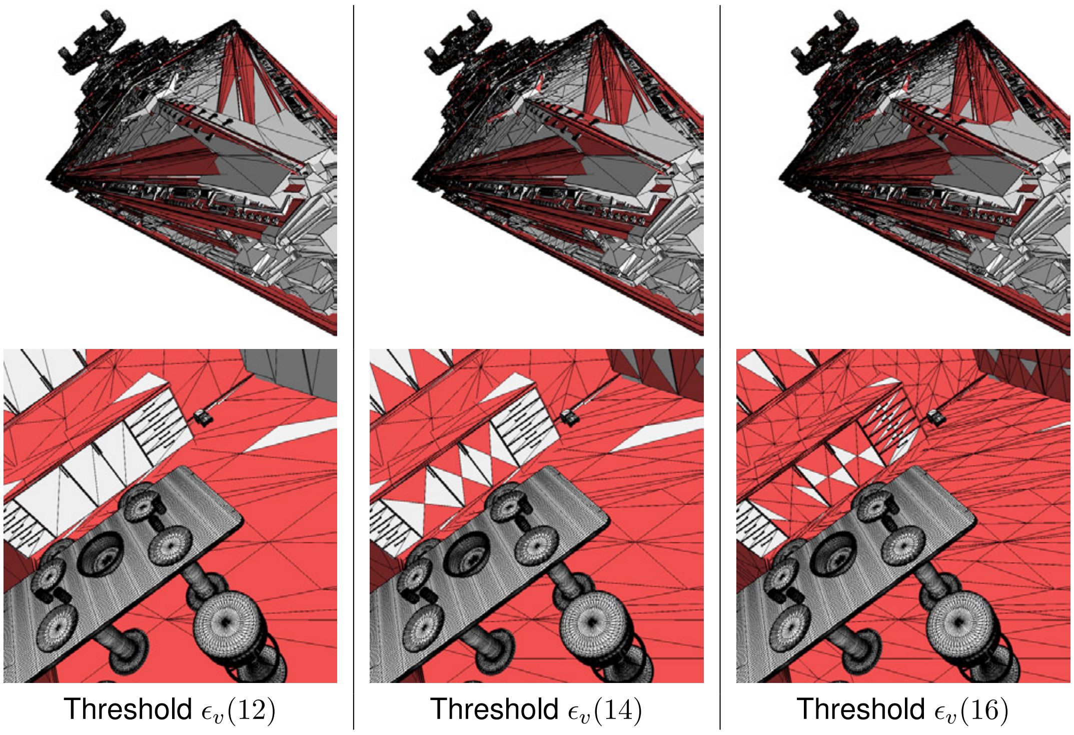

Threshold $\epsilon_v(12)$
Threshold $\epsilon_v(14)$
Threshold $\epsilon_v(16)$

Figure 4.7: Visualization of the effect of different threshold parameters t, where the subdivided triangles are highlighted in red. Note how the long thin triangles with huge axis-aligned bounding boxes become subdivided, while others that would not hurt bounding volume hierarchy performance remain untouched (for the space ship this is observed best at the center of the images).

Results

We apply the subdivision heuristic to a set of quite different scenes in order to verify its versatility (see Figure 4.8). The benchmarks were performed using the QBVH from Chapter 3 and using single thread primary rays on a Intel®Core$^{TM}2$ 2.4GHz processor.

We first consider four static scenes:

1. The Dragon, Buddha, and Bunny scenes are just included to verify that for many small triangles of roughly the same size the heuristic does not add any new triangles in the used parameter range. This kind of scenes do not benefit from subdivision.

2. The space ship scene represents a kind of worst case scenario for classic bounding volume hierarchies: It consists of long thin triangles for the outer hull and many small triangles for details. Additionally this object is rotated by 45 degrees in space. The effect of different threshold parameters t is visualized in Figure 4.7.

3. The kitchen scene is one frame from the BART animation repository [Lext 00],

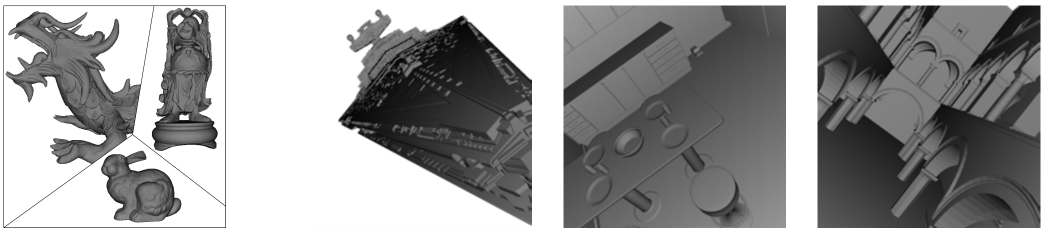


Figure 4.8: Images of the test scenes used for inspecting EVH subdivision. The space ship consists of 1554416 triangles. The kitchen has 110561 triangles and the Sponza Atrium 66454.

where instead of moving the camera the triangles are transformed.

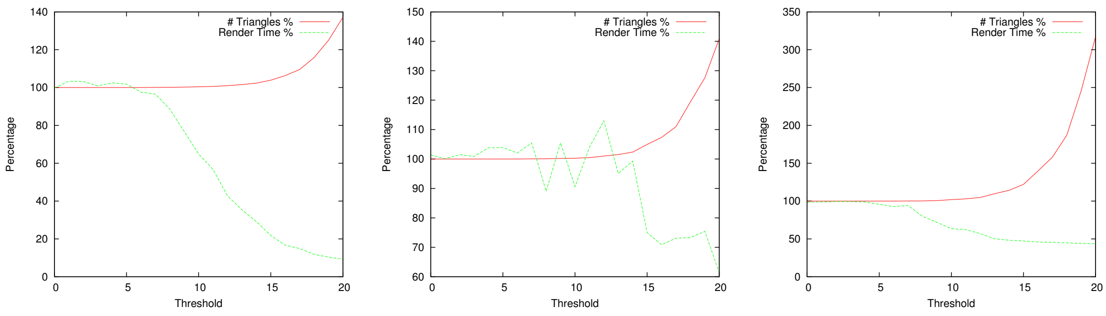
4. The Sponza atrium scene is a typical architectural model, where many triangles are parallel to the canonical planes. While classic heuristics like the surface area heuristic can build efficient bounding volume hierarchies for such scenes, performance drops dramatically, if geometry is rotated and bounding boxes increase in volume.

A higher threshold parameter t improves performance, but also increases the memory footprint. Both numbers are related in Figure 4.9, where we show the relative increase in the number of triangles and corresponding rendering time for a range of thresholds parameters t. A clearly consistent improvement over the test scenes can be observed and it is especially interesting that major performance improvements are obtained at already a moderate increase of the number of triangles.

The second test consists of rotating the well known Sponza atrium scene to illustrate the adaptivity of the edge volume heuristic. The scene is first rotated by $90\,^\circ$, $20\,^\circ$, and $30\,^\circ$ around the x-, y-, and z-axis in 32 steps. Second it is rotated another 32 steps to it's final position $-180\,^\circ$, $0\,^\circ$, and $90\,^\circ$ where all large triangles are again axis aligned. Figure 4.10 shows the performance figures and the number of generated triangle references over the animation for several threshold parameters t. Again the heuristic proves to be reliable: In simple cases no triangles are added. When bounding boxes become inefficient a moderate increase in the number of triangle references avoids the dramatic performance drop.

Subdividing an edge in the middle results in two new bounding boxes that each have $\frac{1}{8}$ of the original volume, because the subdivided edges remain diagonals of their bounding boxes. Since our construction is based on the SAH it is interesting to look at the reduction of the triangle's bounding box surface area upon subdivision and corresponding statistics are collected in Table 4.1. Even though the factor of the worst area reduction is quite large the average area reduction shows the effectiveness of the heuristic.

The EVH is an economic heuristic to subdivide triangles such that the amount of empty space in bounding boxes is efficiently reduced. The algorithm is simple, numerically robust, and can be used as a topology unaware pre-process to any renderer. Big



Space Ship   Kitchen   Sponza Atrium (Frame 16)

Figure 4.9: Relative performance for three example scenes. As expected render time improves until it asymptotically reaches saturation and the number of triangles increases with an asymptotically exponential behavior. The consistent behavior over quite different scenes clearly shows dramatic performance improvements at already very moderate increase in the number of triangles.

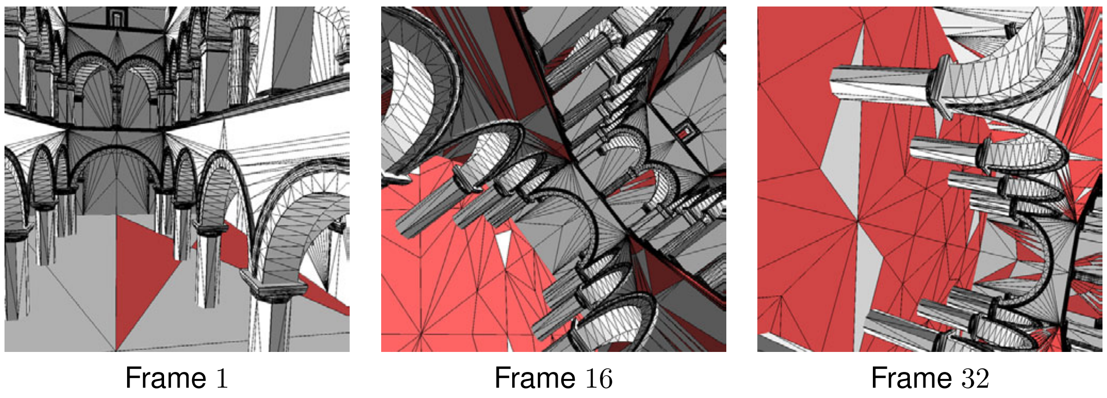
Scene	best	average	worst
Space Ship	0.25	0.435	1.0
Kitchen	0.25	0.463	0.992
Sponza (Frame 16)	0.25	0.457	0.99

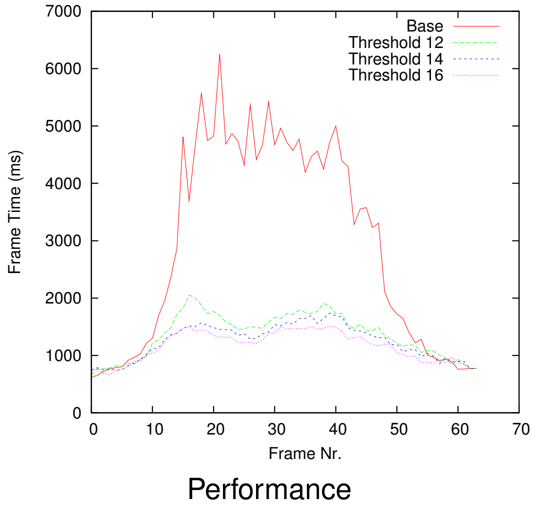
Table 4.1: Factors of best, average, and worst surface area reduction resulting from applying the edge volume heuristic for triangle subdivision. The theoretical maximum of 0.25 is achieved in some cases.

performance improvements already result from very moderate additional memory requirements. While the technique certainly has also applications in collision detection and occlusion culling, too, there are two more points of future interest: There are situations, where a global volume threshold ϵ_v may be not sufficient and a local threshold may perform better. Furthermore there are situations where neither our new heuristic nor the surface area heuristic can reduce overlap. Finding a criteria to identify these situations could be an interesting problem.

4.3.2 Bounding Box Area Heuristic (BBAH)

The EVH was optimized to be used with an SAH based hierarchy construction and considered only few triangles for subdivision in the most scenes. A more uniform subdivision is achieved by changing the threshold heuristic. The one presented now uses the area of a triangle bounding box to determine the subdivision threshold. The threshold





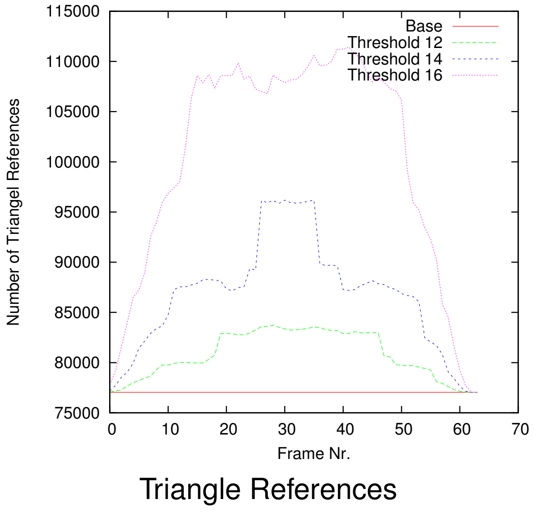


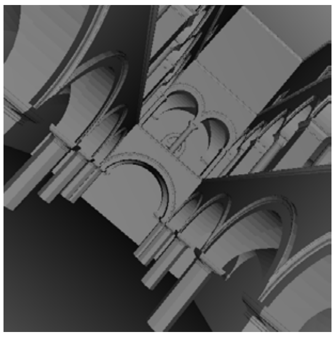
Figure 4.10: The Sponza atrium under rotation. The top row shows 3 example frames out of the total 64 frames, where subdivided triangles are highlighted in red. The more the architectural model becomes rotated, the more bounding boxes of previously axis-aligned geometry become inefficient, which is reliably avoided by the subdivision heuristic (threshold parameter $t = 14$). The graphs in the bottom row show how the frame time is improved and how many triangles are added by the subdivision heuristic for the un-subdivided geometry (Base) and three subdivision thresholds over the 64 frames of the animation.

is computed as:

$$\epsilon_A(t) := \frac{A}{2^t}$$

The surface area is computed from a bounding box $\{(x_{min}, y_{min}, z_{min}), (x_{max}, y_{max}, z_{max})\}$ as

$$A = 2((x_{max}-x_{min})(y_{max}-y_{min})+(x_{max}-x_{min})(z_{max}-z_{min})+(y_{max}-y_{min})(z_{max}-z_{min}))$$


Rotated Sponza 2.747fps to 2.97fps

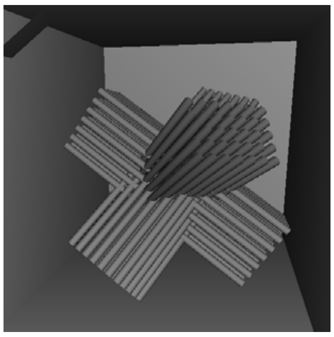

Worst Case 2.326fps to 3.277fps

Figure 4.11: The two test scenes used to illustrate the benefit of leaf compaction. Both scenes were tested single core at a resolution of 512×512. The results are given as frames per second. Both scenes used the Edge Volume Heuristic subdivision with $\epsilon_v(14)$. In the rotated Sponza scene from 87233 generated triangles 4793 duplicates were removed. In the worst case scene were from 423394 generated triangles 288502 duplicates removed.

We use again a power of 2 in the threshold computation to have a more intuitive parameter t. The rest of the implementation is exactly the same as in Section 4.3.1. Please note that this heuristic does not guarantee that the resulting mesh is watertight as connected triangles may have a different area of their bounding boxes. The results of this subdivision and the comparison to the EVH can be found in Section 4.3.4.

4.3.3 Leaf Compaction

Section 4.3.1 described how to efficiently implement the subdivision as a pre-process using only bounding boxes instead of actual triangles. The final leaves in the acceleration structure contain then only references to these original triangles. In Section 3.2 it was described that using fat leaves (i.e. referencing many triangles per leaf) is also beneficial for the memory consumption and the rendering speed. Considering these optimizations together leads to the conclusion that in many cases the leaves of a hierarchy that was constructed after triangle subdivision has to reference the same triangle. An additional optimization is now to remove these duplicates. While there exist efficient methods to remove duplicates in an array in our case a simple brute force implementation is in most cases fast enough because the arrays are very short.

The benefit of using leaf compaction is illustrated in the performance measurements for the two scenes shown in Figure 4.11. Of course leaf compaction only results in a notable speed up when many triangles are subdivided. For this case the *Worst Case* scene was constructed. It contains many long diagonal triangles that result in bounding boxes as large as the whole scene when not subdivided.

4.3.4 Triangle Subdivision and the SAH

As described in Section 4.2 the surface area heuristic (SAH) is used to construct a high quality QBVH. The main reason for using the more complex SAH hierarchy construction over, for example, the very simple mid split heuristic was a significant performance improvement. This improvement showed especially when the scene had many triangles of different size and a more irregular structure. Even though the SAH improved the performance in these scenes it was shown previously in this section that it can be additionally helpful to subdivide triangles.

The *Edge Volume Heuristic* (EVH) has been developed to be used with the SAH hierarchy construction as it tries to subdivide only worst case triangles. The other subdivision heuristic based on the area of bounding boxes (BBAH) more uniformly subdivides all larger triangles. What is now interesting is to analyze how this subdivision of triangles affects the rendering speed when the simpler center-split construction is used. Previous experiments showed that for scenes with an equal distribution of triangles of almost equal size (i.e. the Stanford Scanning Repository Models) the difference between SAH and the mid split construction is not significant. In this section we further investigate this observation and assumption by an experimental analysis of 4 different scenes. These scenes are used in their original form and additionally are “optimized” by the two triangle subdivision algorithms described so far. This analysis also gives insights into the performance impact of triangles of different size and non-uniform distribution.

To compare and understand the influence of the scene structure on rendering speed, we compare the rendering speed of primary rays for different scenes at a different subdivision threshold using the SAH hierarchy construction and the center-split construction. For the evaluation the *Edge Volume Heuristic* (EVH) and the bounding box area based subdivision algorithm (BBAH) is used. Note that the two subdivision algorithms should not directly be compared because the EVA is based on a volume threshold while the BBA uses area measure.

The four tables 4.2, 4.3, 4.4, and 4.5 show the 4 scenes used for the comparison and the results obtained. The scenes are named Interior, Space Ship, Rotated Sponza, and Worst Case. They are ordered such that the first has the highest irregularity in the triangle distribution and the last is the most compact.

The images in the tables 4.2 and 4.3 show the density of the triangle bounding boxes in space. The density is computed by projecting all axis aligned bounding boxes from the triangles to the three axis aligned planes XY, XZ, and YZ. Then the resulting rectangle is rasterized into a fixed resolution floating point image and accumulated. The final image is generated by normalizing the accumulated image so that each pixel value is in the interval $[0, 1]$ and using the square root of this value to interpolate the color gradient. The square root is used to give more details in areas where there are few triangles. We use the bounding boxes of the triangles here because this is what the hierarchy construction algorithms use when the acceleration structure is build. Each of the two tables 4.2 and 4.3 shows the scenes in three different versions. The first line contains the original scenes as it was modeled. The next line shows the resulting triangle distribution when the *Edge Volume Heuristic* is used with a threshold of $\epsilon_v(14)$.

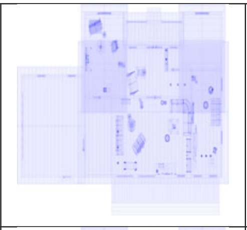
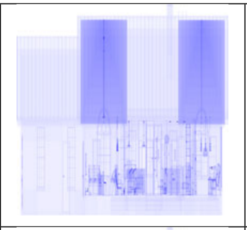
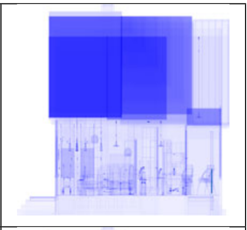
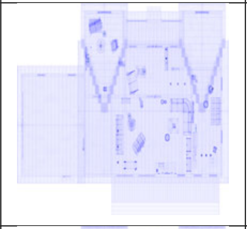
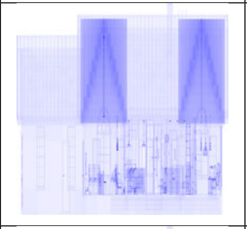
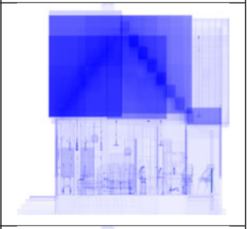
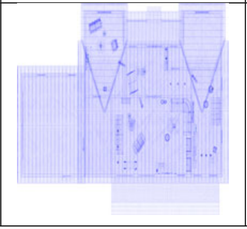
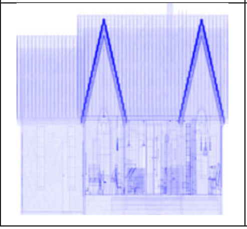
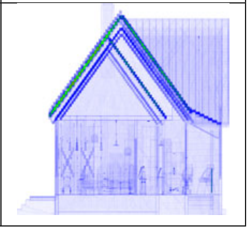
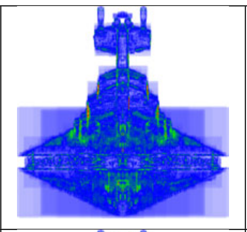
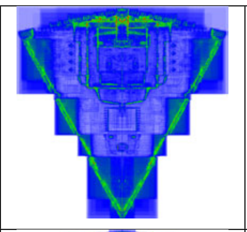
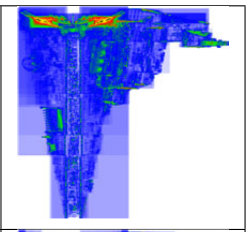
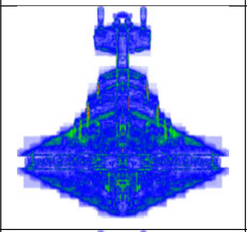
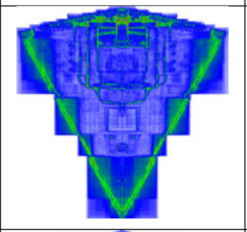
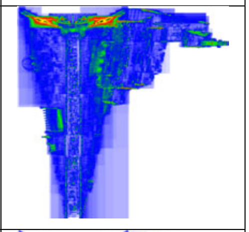
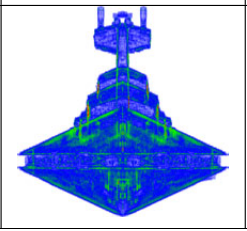
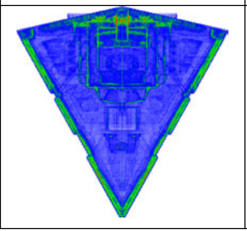
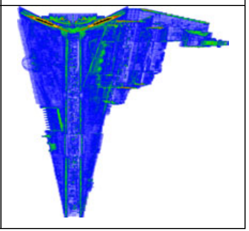
The third line contains the results when the *Bounding Box Area Heuristic* is used. When studying and comparing the images it is important to keep the actual number of triangles in mind because they show only the relative distribution. It can be observed that both subdivision heuristics reduce the overlap of the bounding volumes. The BBA heuristics produces significantly more triangles than the EVH and thus has much less overlap. The two tables (4.4 and 4.5) show the performance results when comparing the two different hierarchy construction methods with the different subdivision methods. Here especially the ratio of rendering time using the center-split construction and the SAH construction gives insights into the difference of the chosen test scenes. It can be observed that the EVH in combination with the SAH construction gives very good results with only a very small increase in the number of triangles. It can also be observed that forcing all triangles to roughly the same size with the BBA significantly increases the quality of center-split constructed trees. Still, SAH constructed trees always perform significantly better.

The results using the four test scenes give very interesting insights into the structure of the rendering performance of different scenes. The first two scenes (the interior scene and the space ship) have a highly irregular distribution of triangles. Especially the interior has, due to the arrangement of furniture and wide empty space, a more non-uniform distribution. Additionally, the density pictures show almost no green or higher colors because the most triangles are used by small objects standing in the shelves. The triangle subdivision then mostly generates triangles in the roof framework where the largest triangles are used for modeling. This highly irregular triangle distribution also explains the large speed difference between the center-split construction and the SAH hierarchy construction that can be seen in Table 4.4.

A similar situation, but not as pronounced as in the interior scene, can be seen in the space ship scene. Here the most triangles are used at the outer hull and for the engines.

The other two scenes have a more uniform triangle distribution and the difference between SAH and center-split construction is much smaller. The difference almost vanishes when the scenes are subdivided even more. To illustrate this up to the limit, the graph in Figure 4.12 shows the performance of center-split vs. SAH of the rotated Sponza scene subdivided up to $\epsilon_v(19), \epsilon_A(19)$ using both subdivision methods. The minimum in rendering time for the BBA subdivision is reached at a threshold of about $\epsilon_v(14), \epsilon_A(14)$ for both the center-split and the SAH hierarchy construction. For higher ϵ there are too many triangles generated. The EVH algorithm has not reached a true minimum up to this ϵ_v but it subdivides much less triangles and the resulting hierarchies are very similar. Experiments up to $\epsilon_v(24)$ showed only a variance of $< 0.1\%$ in the rendering time.

To summarize the result of this section, one could say that for compact scenes (i.e. more or less uniform distribution of triangles) the SAH clearly looses the advantage and converges (performance-wise) to the center-split solution. When sufficient knowledge

Scene	XY-Plane	XZ-Plane	YZ-Plane
Interior (original) 909441 Triangles			
Interior EVH Subdiv. $\epsilon_v(14)$ 918142 Triangles			
Interior BBA Subdiv. $\epsilon_A(14)$ 2095607 Triangles			
Ship (original) 1574657 Triangles			
Ship EVH Subdiv. $\epsilon_v(14)$ 1582217 Triangles			
Ship BBA Subdiv. $\epsilon_A(14)$ 2485195 Triangles			

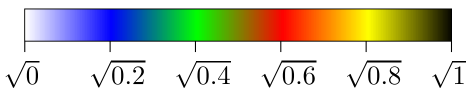
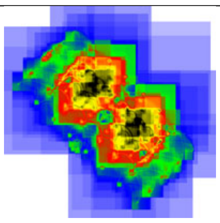
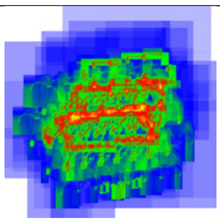
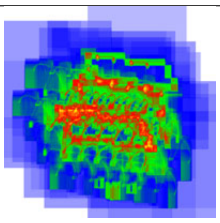
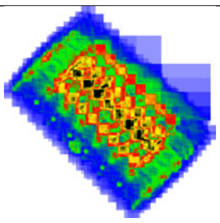
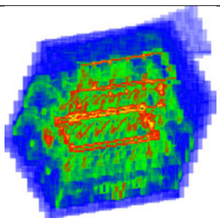
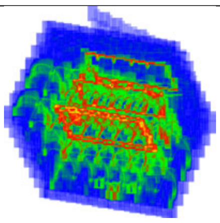
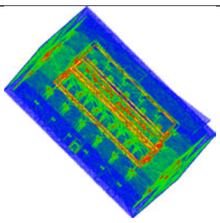
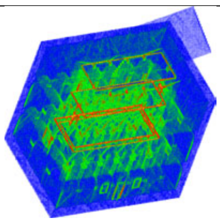
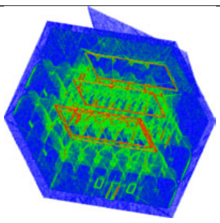
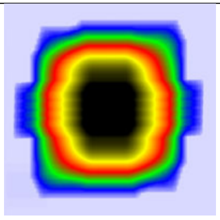
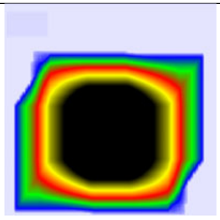
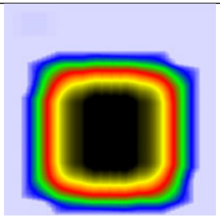
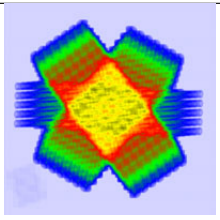
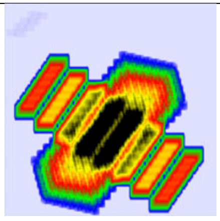
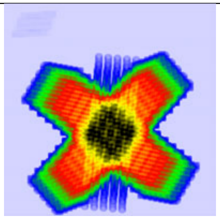
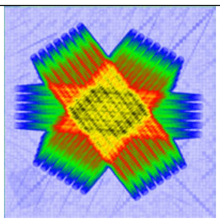
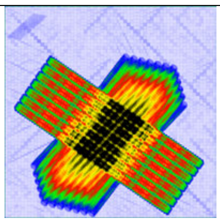
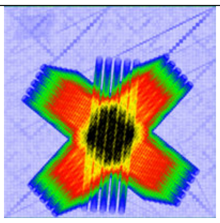


Table 4.2: Projected relative triangle density for the two irregular scenes Interior and Space ship. The images show color coded the relative triangle density of the scenes without subdivision, EVH subdivision, and Bounding Box Area Heuristic (BBA) subdivision.

Scene	XY-Plane	XZ-Plane	YZ-Plane
Rotated Sponza (original) 77051 Triangles			
Rotated Sponza EVH Subdiv. $\epsilon_v(14)$ 87233 Triangles			
Rotated Sponza BBA Subdiv. $\epsilon_A(14)$ 645503 Triangles			
Worst Case (original) 6732 Triangles			
Worst Case EVH Subdiv. $\epsilon_v(14)$ 423394 Triangles			
Worst Case BBA Subdiv. $\epsilon_A(13)$ 4988912 Triangles			

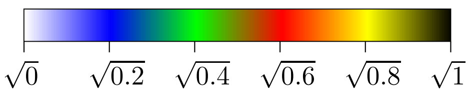


Table 4.3: Projected relative triangle density for the two compact scenes Rotated Sponza and the Worst Case scenario. The images show color coded the relative triangle density of the scenes without subdivision, EVH subdivision, and Bounding Box Area Heuristic (BBA) subdivision.

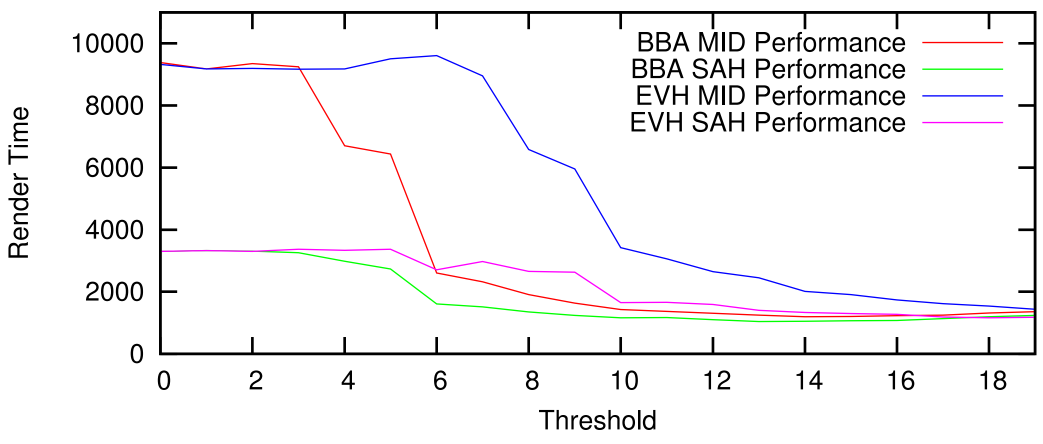
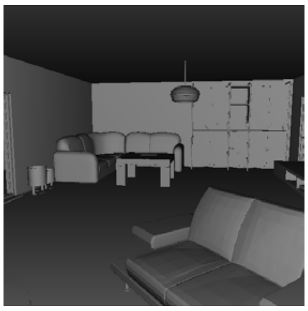


Figure 4.12: Comparison of SAH and center-split performance for the rotated Sponza scene with even higher triangle subdivision levels. Note that the BBA algorithm produces 15 Million triangles for $\epsilon_A(19)$. The render time is again the milliseconds for rendering four frames at a resolution of 512×512.

about the scenes that should be rendered is available it is a viable alternative to use simple construction algorithms. But for complex scenes it pays off to use a more sophisticated hierarchy construction method. What this section also has shown is the utility of the EVH subdivision when using the SAH hierarchy construction. The tables 4.4 and 4.5 show a clear advantage when comparing the number of generated triangles and the resulting rendering performance to the BBA subdivision.



Interior - Edge Volume Heuristic

$\epsilon_v(t)$	#Tris	MID-T	SAH-T	M/S
$\epsilon_v(0)$	909441	6006	799	7.5168
$\epsilon_v(6)$	909639	7549	784	9.6288
$\epsilon_v(7)$	909767	8549	894	9.5626
$\epsilon_v(8)$	909873	7896	924	8.5454
$\epsilon_v(9)$	910501	8364	816	10.2500
$\epsilon_v(10)$	911095	11531	775	14.8787
$\epsilon_v(11)$	911685	13311	807	16.4940
$\epsilon_v(12)$	913729	11734	847	13.8536
$\epsilon_v(13)$	915647	12692	811	15.6498
$\epsilon_v(14)$	918142	11762	793	14.8322

Ship - Edge Volume Heuristic

$\epsilon_v(t)$	#Tris	MID-T	SAH-T	M/S
$\epsilon_v(0)$	1574657	19915	4656	4.2772
$\epsilon_v(6)$	1574761	16931	3351	5.0525
$\epsilon_v(7)$	1574829	17880	2905	6.1549
$\epsilon_v(8)$	1575059	18713	2851	6.5636
$\epsilon_v(9)$	1575417	15694	2684	5.8472
$\epsilon_v(10)$	1575667	19461	2395	8.1256
$\epsilon_v(11)$	1576347	16858	2476	6.8085
$\epsilon_v(12)$	1577955	16944	2310	7.3350
$\epsilon_v(13)$	1579375	14696	2058	7.1409
$\epsilon_v(14)$	1582217	16131	2236	7.2142

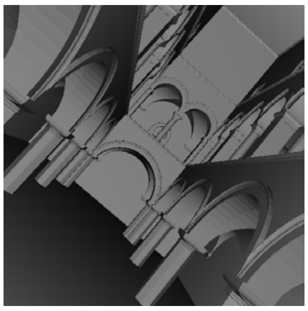
Interior - BBA Subdiv. Heuristic

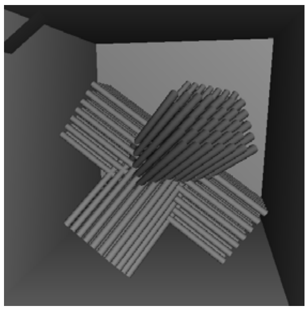
$\epsilon_A(t)$	#Tris	MID-T	SAH-T	M/S
$\epsilon_A(0)$	909441	5940	812	7.3152
$\epsilon_A(6)$	914079	7198	697	10.3271
$\epsilon_A(7)$	918445	5763	691	8.3400
$\epsilon_A(8)$	932357	4457	650	6.8569
$\epsilon_A(9)$	948845	5268	661	7.9697
$\epsilon_A(10)$	998031	6126	660	9.2818
$\epsilon_A(11)$	1061021	7437	643	11.5660
$\epsilon_A(12)$	1235635	6622	656	10.094
$\epsilon_A(13)$	1470019	4598	613	7.5008
$\epsilon_A(14)$	2095607	2505	623	4.0208

Ship - BBA Subdiv. Heuristic

$\epsilon_A(t)$	#Tris	MID-T	SAH-T	M/S
$\epsilon_A(0)$	1574657	19515	4641	4.2049
$\epsilon_A(6)$	1576723	15196	2560	5.9359
$\epsilon_A(7)$	1579653	13334	2329	5.7252
$\epsilon_A(8)$	1585429	10395	1950	5.3308
$\epsilon_A(9)$	1598057	8209	1716	4.7837
$\epsilon_A(10)$	1621891	5664	1580	3.5848
$\epsilon_A(11)$	1676503	4502	1407	3.1997
$\epsilon_A(12)$	1783367	3513	1253	2.8037
$\epsilon_A(13)$	2023323	2782	1137	2.4468
$\epsilon_A(14)$	2485195	2442	1055	2.3147

Table 4.4: Performance measurements for different subdivision thresholds. The two hierarchy construction algorithms SAH and center-split are compared. T is the subdivision threshold as described above. A threshold of 0 results in the original scene. #Tris is the number of resulting triangles. MID-T and SAH-T are the rendering times for the shown camera position when using the center-split construction or the SAH hierarchy construction. The time is given in milliseconds and are the time needed to render 4 frames at a resolution of $512x512$. M/S is the ration of center-split rendering time to SAH rendering time.





Rotated Sponza - Edge Volume Heuristic

$t, \epsilon_v(t)$	#Tris	MID-T	SAH-T	M/S
$\epsilon_v(0)$	77051	9203	3289	2.7981
$\epsilon_v(6)$	77085	9668	2718	3.5570
$\epsilon_v(7)$	77101	8974	2974	3.0174
$\epsilon_v(8)$	77157	6554	2659	2.4648
$\epsilon_v(9)$	77337	5847	2711	2.1567
$\epsilon_v(10)$	78503	3398	1636	2.0770
$\epsilon_v(11)$	78814	3027	1666	1.8169
$\epsilon_v(12)$	80173	2690	1592	1.6896
$\epsilon_v(13)$	84775	2329	1422	1.6378
$\epsilon_v(14)$	87233	2019	1327	1.5214

Worst Case - Edge Volume Heuristic

$t, \epsilon_v(t)$	#Tris	MID-T	SAH-T	M/S
$\epsilon_v(0)$	6732	34496	22313	1.5460
$\epsilon_v(6)$	47052	5909	4497	1.3141
$\epsilon_v(7)$	47052	5977	4499	1.3285
$\epsilon_v(8)$	47052	5920	4492	1.3179
$\epsilon_v(9)$	141134	3702	1873	1.9760
$\epsilon_v(10)$	141134	3692	1841	2.0054
$\epsilon_v(11)$	141136	3688	1841	2.0032
$\epsilon_v(12)$	422820	2406	1221	1.9705
$\epsilon_v(13)$	423384	2412	1200	2.0100
$\epsilon_v(14)$	423394	2383	1218	1.9565

Rotated Sponza - BBA Subdiv. Heuristic

$t, \epsilon_A(t)$	#Tris	MID-T	SAH-T	M/S
$\epsilon_A(0)$	77051	9183	3299	2.7835
$\epsilon_A(6)$	78905	2593	1581	1.6401
$\epsilon_A(7)$	79803	2313	1529	1.5127
$\epsilon_A(8)$	86009	1886	1328	1.4202
$\epsilon_A(9)$	91625	1642	1235	1.3295
$\epsilon_A(10)$	115275	1430	1163	1.2296
$\epsilon_A(11)$	140785	1368	1143	1.1968
$\epsilon_A(12)$	225771	1313	1111	1.1818
$\epsilon_A(13)$	336047	1241	1042	1.1909
$\epsilon_A(14)$	645503	1195	1053	1.1348

Worst Case - BBA Subdiv. Heuristic

$t, \epsilon_A(t)$	#Tris	MID-T	SAH-T	M/S
$\epsilon_A(0)$	6732	34467	22393	1.5392
$\epsilon_A(6)$	141812	2997	1688	1.7754
$\epsilon_A(7)$	182690	1693	1367	1.2384
$\epsilon_A(8)$	424732	1443	1110	1.3000
$\epsilon_A(9)$	549618	1077	957	1.1254
$\epsilon_A(10)$	1271630	1035	907	1.1411
$\epsilon_A(11)$	1654964	884	822	1.0754
$\epsilon_A(12)$	3816586	932	838	1.1122
$\epsilon_A(13)$	4988912	872	809	1.0779
$\epsilon_A(14)$	—	—	—	—

Table 4.5: Performance measurements for different subdivision thresholds. The two hierarchy construction algorithms SAH and center-split are compared. T is the subdivision threshold as described above. A threshold of 0 results in the original scene. #Tris is the number of resulting triangles. MID-T and SAH-T are the rendering times for the shown camera position when using the center-split construction or the SAH hierarchy construction. The time is given in milliseconds and are the time needed to render 4 frames at a resolution of $512x512$. M/S is the ration of center-split rendering time to SAH rendering time.

4.4 Crack-Free Triangle Intersection

So far the presented methods improved the acceleration hierarchies and accelerated the general ray tracing computation for incoherent rays in different situations. While this acceleration enables to simulate complex lighting situations within a reasonable time limit another important aspect of numerical simulations was ignored so far. Sometimes numerical robustness is more important than pure rendering speed. Especially for finely tessellated triangular models using the usual triangle tests as described in Section 2.3.3 may lead to problems at edges or vertices in the form of missed intersections. These problems occur even when the mesh is modeled correctly. Figure 4.13 illustrates this problem. While for simple image generation these infrequent inaccuracies are not a real problem because the are not visible in a final image they pose a serious problem for accurate numerical simulations. For example when simulating energy distributions with highly energetic sources a single missed intersection may destroy the final result. In [Damm 06], a method to accurately ray trace free form surfaces was presented. A

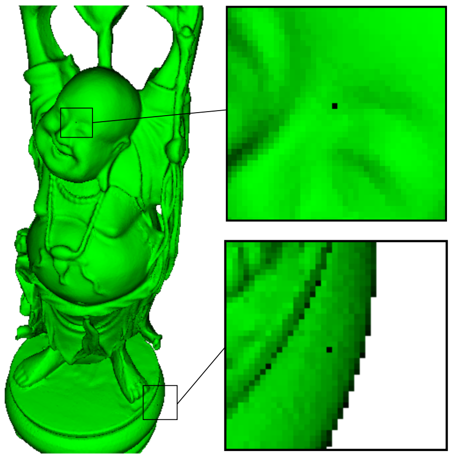


Figure 4.13: The problem of missed intersections when using analytical triangle tests with floating point operations. This happens even when the given scene is modeled water-tight.

given ray is intersected with a Bézier patch by recursively subdividing the patch until the floating point limit is reached. The patch is represented as its axis aligned bounding box and the final intersection is returned as the last and smallest axis aligned bounding box resulting from the subdivision. This same intersection method can be applied to triangles. Instead of analytically computing the intersection a recursive subdivision is started. A triangle is split into four smaller triangles by subdividing each edge exactly in the middle. The middle is used because division by two is a numerically robust operation. This subdivision is performed recursively when a ray intersects a bounding box. In the same way as the edges are subdivided, the barycentric coordinates are subdivided. So for a final intersection box there is also the correct barycentric coordinates for this intersection without further computation. This method results of course in a significant slow down when compared to the classical triangle intersections. For example the scene shown in Figure 4.13 can be ray-traced with 3.64 fps on a 2 GHz Pentium 4 processor with the standard triangle intersection and with 0.74 fps using the subdivision method. This is a factor of 5 slower. But the advantage is that no missed intersections can occur anymore.

A way to speed this up significantly without sacrificing the benefit of accurate intersections is to use a hybrid method. The triangle intersection is split into two parts. In a first step the ray is intersected with the triangle using the standard triangle test. Now only when this test reports a missed intersection the subdivision test is performed. This also removes any missed intersections and increases the performance to 2.24 fps for the Buddha test scene. This is 3 times faster than the original method. Still this method is too slow to be considered a replacement for the standard triangle tests and should only be used when the guarantee of watertight meshes is needed.

4.5 SAH Clustering for Ray Tracing of Dynamic Scenes

There has been a lot of research on the topic of ray tracing non-static (or dynamic) scenes. An overview is given in [Wald 07c]. The fundamental problem of ray tracing dynamic scenes is that rebuilding the acceleration structure for millions of triangles is usually the bottleneck. In this section we will consider a special case of dynamic scenes that only contain rigid body transformations. All benchmarking in this section is done using single rays. Rigid body transformations allow to use affine transformations like rotation and translation to be performed on single objects but individual triangles are not transformed. Additionally the number of triangles and objects stays the same over the whole animation. This kind of scenes are practically relevant for product visualization. For example when a car is ray traced tires should rotate or a door opened. Another realistic setting is the ray tracing of an interior scene where individual furniture should be movable.

The approach presented here is very simple to implement and can be considered an extension to the high performance ray tracing core for single rays developed in Chapter 3. The obvious approach to ray trace geometry animated by rigid body transformations is to compute the acceleration data structure for the rigid components in advance, to apply the rigid body transformations, and then to combine the resulting transformed objects in a bounding volume hierarchy. As most of the hierarchies are cached, building the top-level per frame hierarchy is relatively cheap. This is similar to scene graph based approaches where all objects are managed by a top level hierarchy. However, this approach fails, when large parts of the geometry overlap, because all overlapping parts have to be checked, which results in a notable slowdown. This problem is related to the overlap-problem of Section 4.3 where the triangle bounding boxes intersected. This overlap can happen e.g. for rotating tires at cars, or the joints of a door, where a convex part of geometry is enclosed by a non-convex one. The solution we present here is rather simple. Instead of caching the hierarchies for the rigid components, the hierarchies are cached for small parts of the components. Building the top-level bounding volume hierarchy is slower now, however, this is amortized, because the overlap can be dramatically reduced resulting in a higher performance of the hierarchy.

Implementation

In the following we detail the steps we took for an efficient implementation of the concept described above. A scene with n individually movable parts is stored in a way that each object can be accessed on its own and has its own set of transformations (position, rotation). There may also be a scene-graph used for the scenes. This does not change the underlying implementation as it only assumes that for any given object the transformation to world coordinates can be computed. For each loaded object a QBVH is constructed as described in Section 3.2. But this hierarchy is constructed in the local coordinates of the object and not in world coordinates. After this construction there are n single acceleration structures. The standard way of ray tracing these objects would now be to simply build a top-level hierarchy around the bounding boxes of each object but as described above this would result in a performance loss due to high overlap. This fact is illustrated on the left side of Figure 4.15. Here a car test scenes with $n = 61$ parts was loaded (see Figure 4.16) and only the object bounding boxes are shown. In the region of the engine or the interior this high overlap can be seen.

The next step is to define a cut through each of the individual QBVHs such that each element of this cut is of the same size (or at least as close as possible). The basic idea of a the hierarchy cut is illustrated in Figure 4.14.

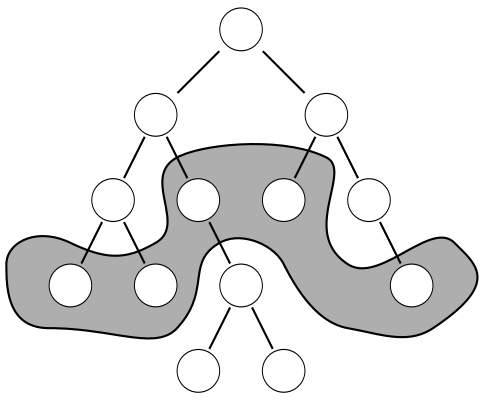


Figure 4.14: Illustration of a cut for a binary hierarchy. The cut is a list of nodes such that each sub-hierarchy of these nodes taken together contains all leaves exactly once.

We call an element of a cut an *entry point* in this context. An *entry point* consists of two axis aligned bounding boxes, a pointer to the hierarchy (or object) it originated from and an index of its position in this hierarchy. The first axis aligned bounding box is the one in object coordinates and is a direct result from the object QBVH. The second axis aligned bounding box is used for the rigid body transformations as described below. To get *entry points* of roughly the same size we use a scene-dependent threshold and size measure that is similar to the one described in Section 4.3.1 for the Edge Volume Heuristic. As a size measure $size(bbox)$ we use the maximum extent of a bounding box:

$$size(bbox) = ||bbox.max - bbox.min||_\infty$$

To define a global scene-dependent threshold ϵ_e we use the size measure of the full scene bounding box divided by a user parameter t:

$$\epsilon_e(t) = \frac{size(sceneBBox)}{t}$$

The cut is constructed by recursively traversing the hierarchy until the measure of the visited node bounding box $size(nodeBBox) < \epsilon_e(t)$ or the visited node is a leaf. This node is then added to the list of *entry points* for this hierarchy. The result of this process is a list of *entry points* for the whole scene. The bounding boxes of such a list are illustrated on the right side of Figure 4.16 for the threshold $\epsilon_e(64)$. Up to now all steps are a pre-process that need to be performed only once per scene. All the following steps are repeated per frame.

The first per frame step is to transform the *entry points* according to the corresponding object transformation $\mathbf{M}$. This can be done efficiently by transforming the object space bounding box that is stored with the affine transformation and recomputing the world space bounding box. The re-computation is needed because a rotation or shear may distort the original bounding box. Listing 4.1 shows a simple implementation of this computation.

```
Vec3 origBBox[2]; // min and max of the original bounding box
Vec3 resBBox[2];  // the result bounding box

for (int i = 0; i < 8; i++) { // iterate over all eight corners
  Vec3 q = qrtVector3(origBBox[i/4].x, origBBox[(i/2)%2].y, origBBox[i%2].z);
  // Multiply transformation Matrix (has to include translation)
  Vec3 p = q * M;
  if (i == 0) {resBBox[0] = resBBox[1] = p;}
  else {
    resBBox[0].SetIfLess(p);
    resBBox[1].SetIfGreater(p);
  }
}
```

Listing 4.1: This code illustrates how to recompute a world space axis aligned bounding box from a rotated AABB by transforming each of the 8 corners and performing the min/max operation.

After transforming all *entry points* the top level BVH needs to be constructed. Here it is important to note that the top level hierarchy needs to be constructed down to a single entry point per leaf because else some of the benefit of deep entry points would be lost. So instead of using a QBVH for the top level a simple BVH is more efficient here. For the construction all different split plane heuristics could be used. Below we compare again the binned SAH construction against the center-split. The performance critical decision is to balance the reconstruction time of the top level hierarchy and the transformation of the *entry points* to the benefit of smaller top level leaves and reduced overlap. Finally, the ray tracing starts with a traversal of the top-level hierarchy for each ray. When a top-level leaf is reached the ray is transformed with $\mathbf{M^{-1}}$, the inverse of the *entry points* object transformation. Now the ray is in object coordinates and can be intersected with the object hierarchy starting from the index of the *entry point*

Even though for the standard ray tracing the object hierarchy levels above the entry points could be discarded to save memory it is not advisable to do this. First it is only a fraction of the actual memory that is needed for the object hierarchies and second these levels are needed to allow for the shadow-caching optimization from Section 5.3. Using the described form of the implementation this allows to use all previously developed optimizations in combination with the new handling of dynamic rigid body scenes.

Another interesting view on this technique is to understand it as a scene-adaptive clustering algorithm that also considers object relations. The resulting *entry points* are nothing else than a collection of primitives that are close together and got clustered by the SAH hierarchy construction.

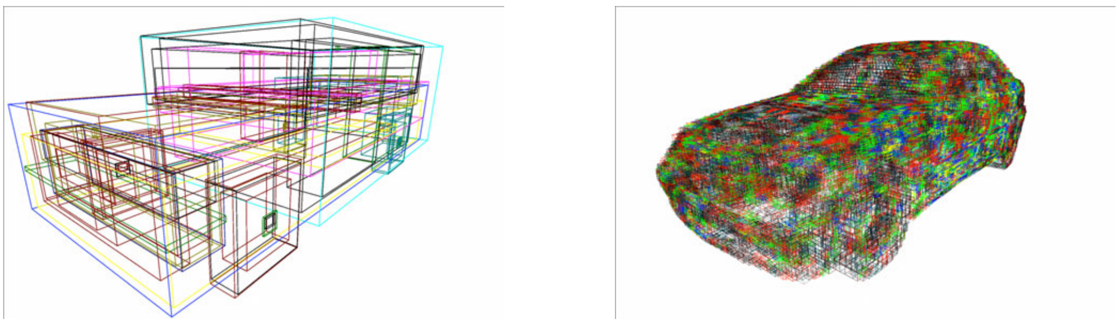
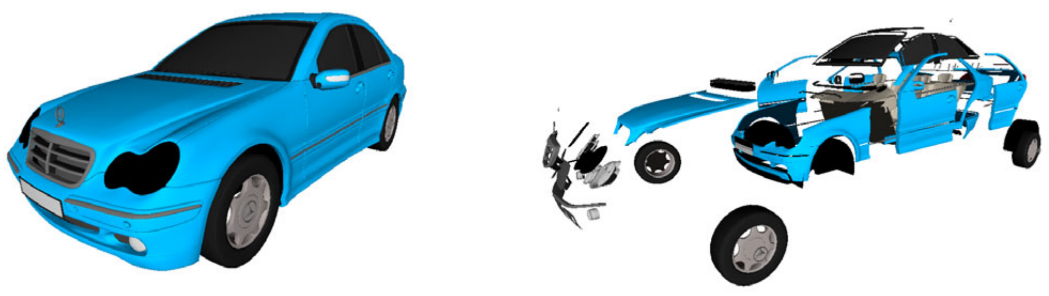


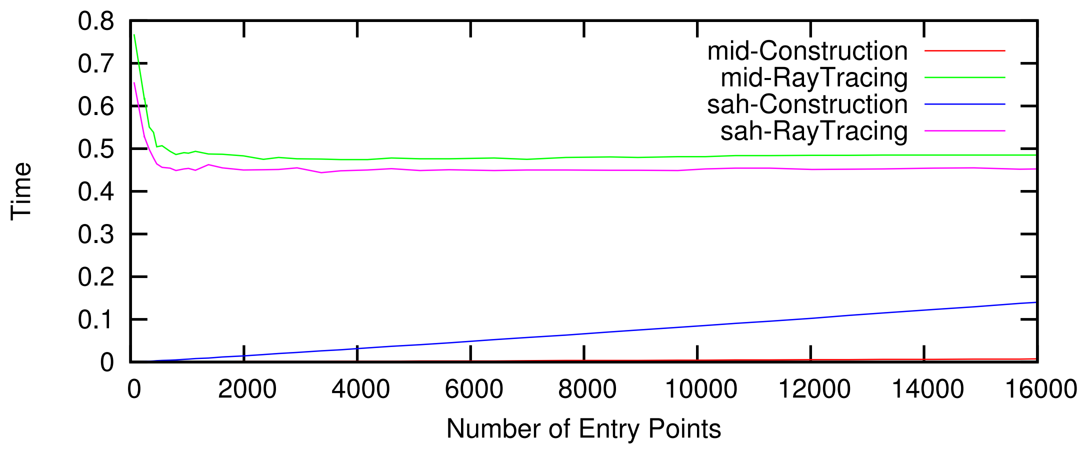
Figure 4.15: These two images illustrate the bounding boxes of the entry points for the car scene for two different thresholds. On the left, only the top level nodes of the object hierarchies are used resulting in 61 entry points. On the right a threshold of $\epsilon_e 64$ was used, resulting in 39526 entry points.

Results

To test the dynamic qualities of the implementation discussed above we used two test scenes. The first scene was a car with 61 movable parts and a total of 1398768 triangles. It is shown with the performance results in Figure 4.16. The second test scene was the interior scene that is shown in Figure 4.17. It consists of 123 individually movable objects and 909441 triangles. The performance measurements were done on a single 2.0 GHz Intel®Core$^{TM}2$ using only a single thread and single ray tracing. The resolution was 960×600 and the timings give the seconds per frame. The graph in the middle of the figures shows the ray tracing time and the top-level hierarchy re-construction time. In all cases the center-split construction is worse compared to the SAH hierarchy construction. The table in the bottom shows the detailed performance.

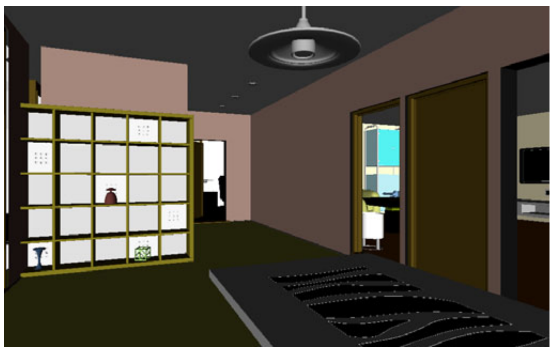
The results illustrate that using the proposed clustering method improves the ray tracing speed considerably. Already a few number of additional entry points can give a benefit. Since this method is very simple to implement and maintain it can be integrated into any rendering system that needs rigid but movable objects in a scene.

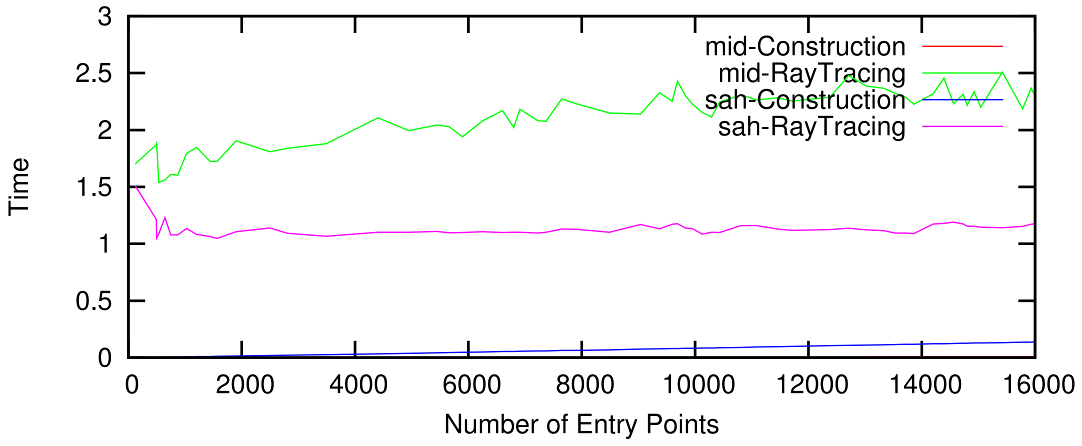




$\epsilon_e(t)$	#Entries	MID T-T	MID RC-T	MID RT-T	SAH T-T	SAH RC-T	SAH RT-T
$\epsilon_e(0)$	61	0.768	0.00000	0.768	0.656	0.00025	0.656
$\epsilon_e(1)$	244	0.615	0.00013	0.615	0.528	0.00156	0.527
$\epsilon_e(2)$	247	0.619	0.00006	0.619	0.529	0.00144	0.528
$\epsilon_e(4)$	406	0.538	0.00006	0.538	0.481	0.00250	0.478
$\epsilon_e(8)$	802	0.487	0.00038	0.486	0.454	0.00531	0.449
$\epsilon_e(16)$	2614	0.480	0.00106	0.479	0.471	0.01988	0.451
$\epsilon_e(32)$	11272	0.489	0.00506	0.483	0.550	0.09588	0.454
$\epsilon_e(64)$	39526	0.520	0.01912	0.501	0.828	0.37188	0.456

Figure 4.16: The car test scene with 61 movable parts with a total of 1398768 triangles. On the top right side the individually movable parts are shown. The timings give the seconds per frame at 960×600. In the table, the fist column shows the number of entry points. T-T is the total time to render a primary ray image. RC-T is the time needed to reconstruct the top level hierarchy and RT-T is the pure ray tracing time.





$\epsilon_e(t)$	#Entries	MID T-T	MID RC-T	MID RT-T	SAH T-T	SAH RC-T	SAH RT-T
$\epsilon(0)$	123	1.701	0.00006	1.701	1.51087	0.00069	1.51019
$\epsilon(1)$	492	1.872	0.00006	1.872	1.21806	0.00325	1.21481
$\epsilon(2)$	501	1.881	0.00031	1.881	1.05262	0.00325	1.04937
$\epsilon(4)$	645	1.560	0.00019	1.560	1.23562	0.00425	1.23137
$\epsilon(8)$	1206	1.848	0.00031	1.848	1.09225	0.00844	1.08381
$\epsilon(16)$	4947	1.996	0.00213	1.993	1.14106	0.03844	1.10263
$\epsilon(32)$	9690	2.432	0.00438	2.428	1.25656	0.07994	1.17663
$\epsilon(64)$	16911	2.389	0.00769	2.381	1.29106	0.14688	1.14419

Figure 4.17: Two views of the interior scene with 123 movable objects that was used for benchmarking. The scene consists of a total of 909441 triangles. The graph in the middle shows the ray tracing time and the top-level hierarchy reconstruction time. The timings give the seconds per frame at 960×600. In the table, the fist column shows the number of entry points. T-T is the total time to render a primary ray image. RC-T is the time needed to reconstruct the top level hierarchy and RT-T is the pure ray tracing time.

4.6 Discussion

In this chapter we investigated the factors that influence the ray tracing efficiency of acceleration hierarchies. We started with an overview of commonly used construction heuristics and then applied them successfully to our novel n-ary acceleration structure. The visual comparison of the center-split heuristic and the surface area heuristic gave insights into their performance difference and we showed with experimental results that the SAH is indeed a very good heuristic when incoherent rays are traced. We showed this by investigating the correlation of the surface area metric with actual performance measurements. These comparisons also gave the insights needed to develop our triangle subdivision methods. Especially the edge volume heuristic as a memory efficient and numerically robust triangle subdivision method allowed one to modify the scene database in a way that hierarchies are constructed that perform significantly faster. This was achieved by efficiently identifying and subdividing triangles that reduce the quality of the hierarchy. The investigation of the SAH as a construction algorithm also led to the use of hierarchy construction as a simple clustering algorithm. We showed that with this method, animations of rigid scene objects can be ray traced faster without introducing complex pre-processes or novel data structures.

5 Software Caching for Ray Tracing

So far we considered algorithmic and architecture specific optimizations to reduce the number of intersection steps and increase the computational performance of each step. In this chapter we will investigate software caching as an additional method for accelerating ray tracing. Caching is a common acceleration technique for algorithms that require a lot of memory accesses. Here we do not mean the hardware caches that are available in all modern CPU architectures. These are considered architecture specific optimizations. But instead we will consider manually implemented caching strategies that make explicit use of the algorithmic knowledge to reduce the amount of work. We call them software caches and discuss several such manually implemented caching strategies. In Section 5.1 and 5.2 we will investigate the triangle intersection and present a cached method for parallel triangle intersection. In Section 5.3 we will introduce a novel software cache that uses the acceleration structure of Chapter 3 to great advantage.

It is an interesting observation that the use of coherence in ray tracing (see Section 3.5.3) can also be seen as a form of software caching. For example during bundle traversal a hierarchy node is accessed once and intersected with multiple rays basically caching the node for the multiple ray accesses. All caching mechanisms use some form of coherence as they only work when the assumption that multiple independent computations (in the previous example these are multiple independent rays) need to access the same data. So first we want to distinguish between explicit and implicit coherence. Explicit coherence is what is done in bundle tracing or when space filling curves are used for tracing primary rays. The coherence is explicitly constructed into the algorithm. If it is violated for example by random secondary rays that result is usually slower than using no caching at all. We already discussed that the use of explicit coherence in full global illumination algorithms does generally not work very well. In contrast to this we will consider implicit coherence. Here only assumptions about the behavior of an algorithm are made without explicitly constructing it to exhibit coherence. With knowledge about the underlying operations a caching scheme can be developed that accelerates the algorithm when the assumptions are met but does not slow down the algorithm if they are violated. An example of implicit coherence is the on demand construction (see Section 3.4) of the acceleration structure. Only actually needed nodes are created and stored in memory but no assumptions about which nodes are needed are made beforehand.

In Section 5.1, the paradigm of streaming processing and low memory consumption is applied to leaf intersection. This is achieved using a small cache transforming the triangles into a SIMD-friendly form without the need to explicitly store all triangles in a different format as was done for example in OpenRT [Wald 02a]. This also allows to use more memory efficient, but slower, representations of the triangle data without a major speed impact. We also compare the vertex-index representation of triangles to the direct 9 floats representation often used in real-time ray tracing. Even though other methods for memory reduction exist [Laut 07], they are not as easily integrated into an existing rendering system. Cache-friendly padding of the node structure provides us with some additional memory which can be used to improve performance. In Section 5.3 we will introduce a strategy for shadow rays that directly benefits from the QBVH data structure but could also be integrated (but with a higher overhead) into other acceleration structures like a *k*d-tree. This uses backtracking in a way related to [Havr 07] but without additional memory requirements and with entry points at every node. Our algorithm is also much simpler.

5.1 Triangle Intersection and SIMD Caching

In this section we will investigate how the storage and access patterns of the scene data can be modified to improve the performance. As described in Section 2.3.3 the whole scene is approximated and stored in the triangle representation. When computing an image, the triangles need to be accessed and intersected for every ray and efficient data structures and access patterns are highly important. In real-time rendering systems, the triangle data is often sorted directly during hierarchy construction for more efficient memory access in the leaf intersection. However, this is not an option in production rendering systems, as other parts of the rendering system may depend on the original order like for example the material representations. For this reason, an additional index array is used instead. It is important to use an index array using relative offsets instead of absolute pointers because the size of memory addresses varies between architectures. This is needed because the data structure chosen for a QBVH node is size optimized to be aligned with the cache size and so, changing its size would break the optimization.

Triangle Storage

To store a single triangle one needs to save only the three vertices $p_0, p_1, p_2 \in \mathbb{R}^3$ but it has to be considered that the order of the three vertices is important. Many algorithms rely on a definition on which side of a triangle is processed. The side of a triangle is defined by constructing a plane from the three vertices and using the sign of the distance to this plane to distinguish the positive and the negative side. By defining the orientation of the vertices as either clockwise or counterclockwise, this definition allows a unique decision, which side of a triangle is seen. To compute the plane normal the cross product of the two edge vectors $e_0 = p_1 - p_0$ and $e_1 = p_2 - p_0$ defines the plane normal $n = e_0 \times e_1$ and the positive and negative side. In this thesis we use a counter clockwise definition. This means that when looking on a triangle on screen the positive side is seen when the 3 vertices are in counter clockwise order. The simplest but also most memory consuming mesh data structure is to save each triangle completely independent of the others by storing the three vertices as 9 floating point numbers (three for each vertex). A more compact but still simple data structure is to store only distinct vertices of the mesh and use 3 indices for each triangle. But accessing a triangle in this format needs one additional indirection. In Section 5.1 we will analyze the impact of this representation in the ray tracer we develop here.

The data layout and intersection code for the primitives of the scene have a major impact on the final rendering speed as well as on the total memory consumption. Here we present the results of directly developing data representations for the QBVH described in the Section 3.2. In fact the choice of the primitive intersection is closely related to the acceleration structure and its construction. Similar to [Resh 07] we can even get away with much flatter hierarchies at the cost of leaves that contain more triangles. This further improves traversal speed and also enables the efficient use of streaming SIMD instructions to process multiple objects at once [Chri 06]. Compared to [Resh 07], we use single rays and avoid performance losses caused by masking.

If memory were of no concern, the triangle data in the leaves could be precomputed and stored in a SIMD-friendly layout. For high performance ray tracing, Wald [Wald 04a] uses 48 bytes per triangle, Wächter [Wach 06] 36 bytes. The more compact vertex-index layout stores all vertices in an array and only three indices per triangle referencing these vertices. This is a widely used layout in 3d modeling applications but usually this introduces performance penalties due to an indirection in memory access. We discuss this layout for the QBVH below. Most of the time it is not practical to use a specialized triangle layout, because it would mean that the triangle data has to be duplicated in a rendering system since other parts like shading or an animation subsystem need access to the scene data. This was already discussed in Section 2.3.3. So for practical reasons one constraint for a general ray tracing system is that the original triangle data is left untouched. Our implementation does not assume a specific triangle layout and for benchmarks we implemented the 36 bytes layout as well as the vertex-index form. We efficiently intersect four primitives at once using a branch-free SIMD version. The details of the intersection computation can be found in Section 5.2. For this test, the triangle data has to be collected and swizzled to an SoA SIMD layout. This already works surprisingly fast, but to maximize performance, we use a small direct-mapped cache for each rendering thread to store the precomputed layout. This caching mechanism also allows for the use of vertex-index representation for the triangle data without significant performance loss (see Table 5.2).

Data Layout Cache

Efficient data layout is not only needed for fast computation but may also reduce the total memory consumption of an algorithm implementation. As described in Section 3.2 we favor large leaves to reduce the total amount of memory needed for the acceleration structure. This is done under the assumption that leaves with several triangles can be efficiently intersected using streaming and SIMD instructions. For the computation it is required that the triangle data is loaded in structure of array (SoA) layout inside the SIMD registers. Additionally the number of triangles should be a multiple of four or the rest of the register has to be filled with data in a way that the final result is still correct. As described above the original triangle data should be left untouched when the ray tracer has to work inside a full image generation system. The solution to this problem is to add a software-managed cache to the triangle intersection code. The development of this cache has to take into account the frequency of the cache access and the cost for a cache miss. A QBVH leaf stores the number of triangles and the starting-index into the triangle index array. Using the starting-index stored in a leaf modulo the number of cache lines as tag is a good choice because the number of triangles per leaf is locally uncorrelated. As cache-index the starting-index is used because it is unique. A cache line then consists of the cache-index and enough space for a maximum number of n swizzled triangle vertices. For performance reasons it is a good idea to use a fixed n for all cache lines. When using a BVH acceleration structure this is no problem because during construction it is easy to build a hierarchy with a maximum number n of triangles per leaf. Section 3.2 described already an upper limit for the number of triangles per leaf when the optimized QBVH node of Listing 3.1 is used so this limit is the natural choice for the cache line size. Each cache line holds a multiple of 4 triangles in a SoA layout. The basic data swizzling for a group of 4 triangles is shown in Figure 5.1. When the number of triangles in the given leaf is not a multiple of 4 the cache can not be filled correctly. A branch-free solution to this problem is to use the next values in the triangle index array. This does not change the correctness of the algorithm and avoids a branching instruction. While the benefit of avoiding a branch largely depends on the access time of a triangle vertex it is a elegant solution for the CELL architecture where the access to external data is done via DMA transfers. For this method to work in all cases the original triangle index array has to be extended by 3 replications of the last index. This assures correctness also for the leaf containing the last triangles.

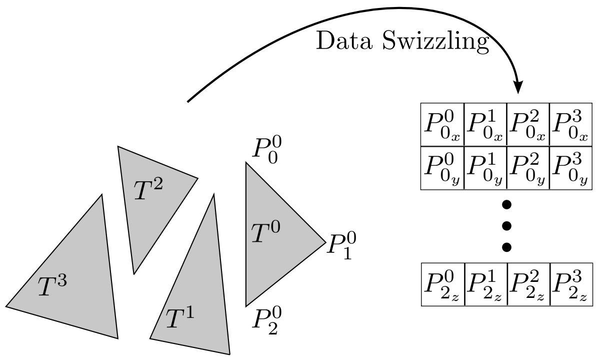

Data Swizzling
$P_{0_x}^0$ $P_{0_x}^1$ $P_{0_x}^2$ $P_{0_x}^3$
$P_{0_y}^0$ $P_{0_y}^1$ $P_{0_y}^2$ $P_{0_y}^3$
$P_{2_z}^0$ $P_{2_z}^1$ $P_{2_z}^2$ $P_{2_z}^3$
P_0^0
T^2
T^0
P_1^0
T^3
T^1
P_2^0

Figure 5.1: Conversion of the vertex data of four triangles into SoA SIMD layout. The original data of the triangles can be in any format as long as for a given triangle index the 3d vertex coordinates can be computed.

For multi-threaded applications an efficient solution to the problem of multiple threads writing to the same cache line is to give every thread its own cache. This not only avoids conflicting behavior but increases the performance on NUMA architectures and is the only fast possibility for threads running on CELL SPEs.

The described triangle cache can of course be used with other acceleration structures than a BVH. When a space subdivision like a *k*d-tree or a grid is used it is not possible to restrict the maximum number of triangles allowed per leaf but only few leaves have a large number of triangle indices. So for an efficient cache it is still beneficial to restrict the maximum number of cache entries per line. When a cache line is accessed and a cache miss occurs it is filled to this maximum number and the leaf intersection function checks whether the number of triangles in this leaf exceeds the maximum number. For the overlap the un-cached (potentially scalar) triangle intersection function is used. As this usually happens in few leaves it is not a major performance problem.

Table 5.1 shows the behavior of the triangle caching algorithm for three different algorithms. The number of cache entries is varied from 256 to 16. The algorithms that are used are a full path tracer, an instant radiosity based global illumination renderer and a simple primary ray casting algorithm. As can be seen from the numbers the implicit caching works very well for the highly incoherent path tracing algorithm even with only 16 entries. One of the reasons that it works well is the larger number of triangles in a QBVH leaf. This of course increases the chance of reusing the node with a different ray. Another important aspect to mention is in the nature of the path tracing algorithm. Here a new ray needs to start from the end point of the previous ray. Since the nearest intersection point is searched the acceleration structure traversal needs always to intersect the node the previous ray terminated in. When using a sorted traversal it is also almost always guaranteed that this node is visited first. So it is guaranteed that the cache already contains this node.

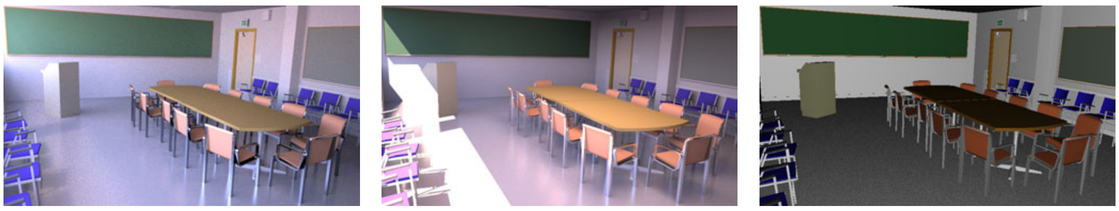


Figure 5.2: The conference scene rendered with three different ray tracing algorithms used to benchmark the triangle cache behavior. From left to right: path tracing without next event estimation, the renderer from Chapter 6 and the last one just primary rays. The numerical results can be found in Table 5.1.

# cache lines	cache misses PT	cache misses IR	cache misses PR
256	11.5%	6.52%	2.4%
128	13.9%	8.87%	3.4%
64	17.5%	13.62%	4.6%
32	24.2%	23.75%	9.8%
16	32.5%	35.02%	14.0%

Table 5.1: The triangle cache misses for different rendering algorithms. PT is a simple path tracer without next event estimation. IR is the renderer described in Chapter 6. PR is just tracing the primary rays in scan line order. The three renderings for this measurement are show in Figure 5.2.

Vertex-Index Triangle Representation

Here we shortly compare the flat 36 byte triangle representation and a more memory efficient index-based representation. The indexed representation is more time consuming for ray tracing because an additional indirection has to be performed prior to triangle intersection. This overhead is reduced when the leaf caching based QBVH implementation is used. The indexing works by storing the vertices of a mesh in a separate array and triangles are stored only as integer values indexing three vertices. This allows one to efficiently reuse shared vertices in the mesh but needs to access two distinct memory locations per triangle access.

The same scenes as in the previous sections are used for this comparison (see Figure 3.5). Each rendering was performed at a resolution of 800×600 pixels with 16 passes. Table 5.2 shows the rendering times and the memory used by the triangle data of the scenes. The resulting QBVHs are the same as in Section 3.5.1. The cached versions both use a cache with 64 entries.

Scene	36 TTI	36 TTI NC	36 MEM	VI TTI	VI TTI NC	VI MEM
Interior	107s	127s	32 MB	108s	166s	16 MB
Plant	191s	226s	438 MB	197s	317s	213 MB
Conference	62s	72s	43 MB	66s	99s	18 MB

Table 5.2: This table compares the impact of using the flat triangle layout with 36 bytes per triangle (36) to a vertex-index (VI) based representation. TTI is the total time to image and MEM is the memory consumption of the triangle data. NC is the time using no cache.

5.2 Parallel Triangle Intersection

As we want to exploit the parallelism in all stages of the ray tracing process we intersect 4 triangles with a single ray. The basic intersection code of [Moll 97] is easy to apply to four triangles in parallel when the triangles are stored in the layout described in Section 5.1. The problem is that only a single ray is intersected with all triangles and so only a single intersection should be returned as a result. In theory it is easy because the required result is just the minimum of the four intersections computed by the parallel code. Additionally the correct index of the triangle is needed for later shading. In practice performing these two operations can quickly be the bottleneck of the whole computation resulting in a slower intersection than the simple scalar code when branches are used. The solution to this problem is to use branch free code to replicate the correct result over the SIMD registers inside the `QRay_t` data structure at the end of the computation.

The code shown in Listing 5.1 below is a parallel implementation of the triangle intersection described in [Moll 97]. The interesting parts are the generation of the `result` mask. This checks if the barycentric coordinates that were computed at the beginning of the intersection are inside the triangle and thus indicating a hit. Additionally the t-far value of the ray is compared to the distance of the intersection. The final result of this computation is a bit-mask indicating which of the 4 triangles is actually intersected by the given ray. If this mask contains only 0, none of the four triangles was hit. Otherwise the mask is used to select the newly intersected triangles and a register called `min` is filled with the previous intersections and the newly found one. This is necessary, because finding the minimum inside a register requires all entries set to a proper value. The minimum is then found by two `SIMD_MIN` operations used on swizzled input data. The result is that the minimum value is contained in all four slots of the `min` register. Finally it is required to get the index of the correct triangle. Here one has to consider also that the result should be correct even if the same intersection was found multiple times. This can occur for example when the original triangle data set contained twice the same triangle and always happens in the last 4-block of the last QBVH leaf when the triangle cache is used (see Section 5.1) and the number of triangles is not a multiple of 4. The `SIMD_MOVEMASK` operation collects the first bit of each of the four slots in an SIMD register. Applying it to a comparison of the `min` register and the original `t` register gives a 4-bit representation of the positions where these two registers are equal. The possible 16 different values are then translated by the `mask_to_idx` array to an index. This index can then for example be used to select and store the correct barycentric coordinates of the intersection that is needed later by the shading computations. The code for this is also shown below. Note that this code can directly be extended to support wider SIMD instructions.

5.3 Accelerating Shadow Rays

Visibility rays (or: shadow rays) are an essential part in many rendering algorithms with deterministic connections of points, e.g. in bi-directional path tracing or instant radiosity. They are merely used to determine the mutual visibility of two given points. Any intersection of an interval on a ray with an object is sufficient to determine occlusion. Therefore, three optimizations are to quit the traversal on the first intersection found, to omit the sorting of the nodes pushed on the stack, and not to compute intersection distances, which can simplify object intersection routines [Smit 98]. All these optimizations are easily implemented into the QBVH traversal. Note that not only the traversal can be optimized but also the triangle intersection code. Since a visibility ray only needs to provide a Boolean result the return values can be simplified by omitting the barycentric coordinates. Also the t parameter of a potential intersection does not need to be returned.

We present here an additional optimization that takes advantage of the shallow QBVHs with the large node structure to further accelerate the shadow rays. This optimization makes use of implicit coherence. It can be easily implemented for other acceleration structures also but when using small nodes it has a much higher overhead than for the QBVH.

Caching Occluders with Backtracking

When tracing a visibility ray the search for an occluding object can be stopped as soon as the first intersection is found. We use this fact to accelerate visibility computations by starting the search deeper in the hierarchy instead of at the root node. The idea is now to use implicit coherence of consecutive visibility rays to start the search from the leaf of a previously found occluder.

Related to this idea is [Resh 05]. They use an entry point search for large bundles of rays in *k*d-trees . In our approach we use a backtracking method that allows every node of the hierarchy to be used as an entry point and will give correct results. For the backtracking, we store the parent with each node. As the `SIMD_BVH_Node` (see Listing 3.1) still provides some room to store additional information, it is possible to keep the index of the parent node without additional memory requirements. This means that the optimization described in this section can be implemented in a QBVH without any overhead. Accessing the parent pointer during construction introduces also no overhead.

```
int intersect4TrianglesSIMD(QRay_t* ray, const SIMD_Triangle_t* tri,
                             float* pu, float* pv) {
  SIMD4_Vec3 e1, e2, tvec, pvec, qvec;
  SIMD_float invDet, t, u, v;
  const SIMD_float ZERO = SIMD_SET1(0.0f);
  const SIMD_float ONE = SIMD_SET1(1.0f);
  const int mask_to_idx[] = {-1,3,2,2,1,1,1,1,0,0,0,0,0,0,0,0};

  SIMD4_VEC3_SUB(e1, tri->p1, tri->p0);
  SIMD4_VEC3_SUB(e2, tri->p2, tri->p0);
  SIMD4_VEC3_CROSS(pvec, ray->d4, e2)

  invDet = SIMD_DIV(ONE, SIMD4_VEC3_DOT(e1,pvec));
  SIMD4_VEC3_SUB(tvec, ray->o4, tri->p0);
  u = SIMD4_VEC3_DOT(tvec,pvec);
  u = SIMD_MUL(u, invDet);

  SIMD4_VEC3_CROSS(qvec, tvec, e1);
  v = SIMD4_VEC3_DOT(ray->d4, qvec);
  v = SIMD_MUL(v, invDet);

  t = SIMD4_VEC3_DOT(e2, qvec);
  t = SIMD_MUL(t, invDet);

  // computing the result mask by checking barycentric coordinates and t
  SIMD_float result = SIMD_CGE(u, ZERO);
  result = SIMD_AND(result, SIMD_CGE(v, ZERO));
  result = SIMD_AND(result, SIMD_CLE(SIMD_ADD(u, v), ONE));
  result = SIMD_AND(result, SIMD_CGT(t, ZERO));
  result = SIMD_AND(result, SIMD_CLT(t, ray->t4));

  if (!SIMD_ISNOTALLZERO(result)) return -1; // no triangle was hit

  // computing the minimum over all intersections
  t = SIMD_AND(result, t);
  SIMD_float min = SIMD_OR(t, SIMD_ANDNOT(result, ray->t4));
  min = SIMD_MIN(min, SIMD_SHUFFLE(min, min, 0,1,2,3));
  min = SIMD_MIN(min, SIMD_SHUFFLE(min, min, 1,0,3,2));
  int idx = SIMD_MOVEMASK(SIMD_CEQ(min, t));
  int hit = mask_to_idx[idx];
  ray->t4 = min;
  *pu = ((float*)&u)[3-hit];
  *pv = ((float*)&v)[3-hit];
  return hit;
}
```

Listing 5.1: The SIMD Triangle intersection computing the nearest intersection of a single ray with 4 triangles. The returned result is the index of the triangle that got intersected or -1 if none of the 4 triangles was intersected. For further processing the t-far value in the SIMD ray is also correctly replicated.

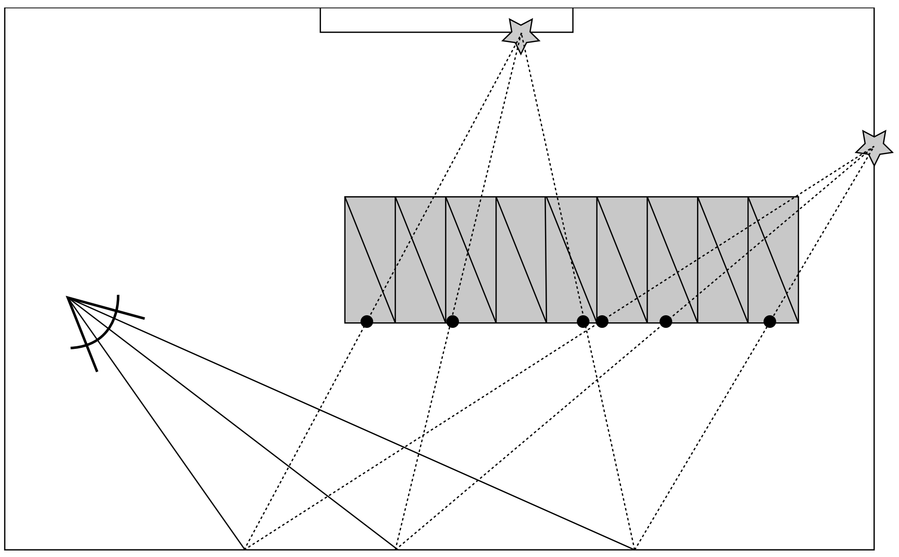


Figure 5.3: Implicit spatial coherence of occluders when using point light based global illumination. The entry point cache is able to quickly find occlusions, since the visibility rays to the same point light source (depicted with a star) exhibit some coherence. The big leaves of the QBVH make a hit even in the same leaf more likely. The performance gain is achieved in a way which is completely transparent to the underlying renderer, no additional complexity is introduced to existing systems (as would be the case for ray packets).

Figure 5.3 shows the idea in the setting of a point light source based global illumination renderer. Consecutive visibility queries result in occlusion by different but spatially nearby triangles. This fact can be exploited by using the node of the last occlusion as the entry point for the next shadow ray. The hierarchy is then searched for intersecting geometry using a backtracking algorithm: the regular traversal stack is initialized with the children of the entry point node and intersected the usual way. If no occlusion has been found so far, the entry point node is the root node, so there is no special code for this case. Next, if the stack is empty, all children of the parent of the entry point node are pushed to the stack, except the one that has just been processed. For the next iteration, the entry point node is set to its parent. The algorithm terminates when the parent of the root node is about to be processed. So the regular traversal is only extended by one variable and one additional loop. If a leaf node is found as an occluder, its parent node is cached for this point light source. This algorithm is illustrated in Listing 5.2. The inner while loop is exactly the same as the unsorted traversal code described in Section 3.3.2. What remains is finding a suitable entry point for an occlusion query. The described algorithm above is correct for any entry point, but an acceleration can only be expected when the hit is close to a previous occlusion. The rendering system uses occlusion queries given by two points, where the first one is the path's hit point

```
.. // the usual SIMD ray setup

int visited = 0;
int hashpos = getEntryHash_idx(p1);
entry = hash_ec[hashpos];

do {
  int top = -1;
  SIMD_int br = IntersectSIMD_Ray_4Box(ray, nodes[index]);

  if (br[0] && pNodes[entry].child[0] && (visited != pNodes[entry].child[0]))
    qStack[++top] = pNodes[entry].child[0];
  if (br[1] && pNodes[entry].child[1] && (visited != pNodes[entry].child[1]))
    qStack[++top] = pNodes[entry].child[1];}
  if (br[2] && pNodes[entry].child[2] && (visited != pNodes[entry].child[2]))
    qStack[++top] = pNodes[entry].child[2];}
  if (br[3] && pNodes[entry].child[3] && (visited != pNodes[entry].child[3]))
    qStack[++top] = pNodes[entry].child[3];}

  while (top >= 0) {
    int index = qStack[top];
    if (index < 0) {
      intersectLeaf(index, ray);
    } else {
      SIMD_int boxResult = IntersectSIMD_Ray_4Box(ray, nodes[index]);
      stack[top] = pNodes[index].child[0]; top -= boxResult[0];
      stack[top] = pNodes[index].child[1]; top -= boxResult[1];
      stack[top] = pNodes[index].child[2]; top -= boxResult[2];
      stack[top] = pNodes[index].child[3]; top -= boxResult[3];
    }
    top--;
  }

  visited = entry;
  entry = pNodes[entry].parent; // now, go up one step
} while (entry >= 0);
```

Listing 5.2: Visibility ray traversal code with the entry point caching optimization. The outer loop performs the backtracking and the inner loop is the standard unsorted QBVH traversal. The hash is explained in Listing 5.3. The variable `p1` is one of the 3d points from the visibility query.

```
#define HASH_SIZE 4096
int hash_ec[HASH_SIZE];

int getEntryHash_idx(const float* p) {
        static const float s = 16.0f;
        int i0 = p[0]*s;
        int i1 = p[1]*s;
        int i2 = p[2]*s;
    return (((i0*73856093) ^ (i1*19349663) ^ (i2*83492791)) & (HASH_SIZE-1));
}
```

Listing 5.3: Simple hash function for a point in 3d space that is given as 3 float values. This hash function is used in Listing 5.2

and the second one the light source position. In our implementation we use a simple unbounded (i.e. hashed) voxel grid and the light position as hash. For each position the last found occlusion entry point is recorded. If no occlusion occurred, the root node is used. In the statistics we used a very simple hash table with 4096 entries and divided the scene into $(6.25cm)^3$ voxels (assuming that 1.0 is $1m$).

Listing 5.3 shows the implementation of the hash function as we used it. Any other good hashing strategy can be used here. The speed of the hashing function is also not very important as it has to be called only once per visibility ray. This optimization can be transparently incorporated into any existing rendering system which spawns shadow rays with two points to be checked for visibility. Of course the order of the points is important in this implementation as only the first of the two visibility points is used for caching.

Results

All benchmarks were done using a single thread on an Intel®Core$^{TM}2$ processor. The resolution for the benchmarks are 800×600 pixels and for the point light based approach 16 rendering passes were used with an average of 22 point light sources per pass. Primary rays were created per scan-line, additional speedup can be expected when using Hilbert curves. We compare the normal implementation of the QBVH (using only the standard shadow optimizations) with the one using the entry point caching. The statistics contain the total number of bounding box intersections performed for the final image. The speedup is of course larger in scenes with a larger amount of occlusion. This explains why the interior scene only shows a marginal speed up. The results are shown in Table 5.3.

The caching method could also be used for other acceleration structures. But compared to acceleration structures like the *k*d-tree, the BIH or binary BVHs which use many, very densely packed, small nodes, this method works especially well for the QBVH. The additional memory requirement of one more parent index per node would result in 50% more memory for standard 8 bytes *k*d-tree nodes as each node needs to

Scene	N#BBox	NTTI	C#BBox	CTTI
Conference	16.8	73s	11.7	62s
Conference 2	20.7	92s	9.9	60s
Interior	17.3	109s	15.4	107s
Plant	44.8	297s	25.0	191s

Table 5.3: Comparison of the number of box intersections and total time to image image between the normal QBVH version (N#BBox, NTTI) and the entry point cache version (C#BBox, CTTI). The images in Figure 3.5 show the test scenes used. For the measurement they were rendered only at a resolution of 800×600 pixels with 16 passes and an average of 22 point light sources per pass. Conference 2 is the same conference scene but with the camera placed partly under the table.

be extended by a 4 byte parent pointer.

The approach can also be used for non-shadow rays. Then the procedure cannot benefit from the early-out of shadow rays, i.e. processing must always continue up to the root node. Still it can be beneficial, if the rays expose some kind of coherence, for example primary rays from the eye or coherent reflections. This allows the ray tracing core to transparently exploit implicit coherence by caching previous intersection entry points without changing the interface and without performance loss when incoherent rays are used. This of course is only a theoretical benefit as traversal rays use an ordered traversal as described in Section 3.3 for intersection rays and thus no speedup is achieved in our case.

5.4 Discussion

In this chapter we investigated the use of software caching for further acceleration of the core ray tracing algorithm. The presented methods are complementary to the previously presented methods and can be integrated into any ray tracing algorithm.

The use of a thread-local triangle cache allows to store triangles in a memory efficient way and still be able to employ a high performance parallel intersection without noticeable overhead from data transformations. As an example we compared the often used vertex-index representation to the plain triangle storage. We also presented a branch-free implementation of a parallel triangle intersection that intersects a single ray with 4 different triangles. The concepts in this intersection are scalable and could be used for wider SIMD architectures. Of course the decision of using a manually managed cache in a full ray tracing system has to be done with care. As already mentioned, each thread needs to use its own version of the cache or some other memory protection mechanisms need to be implemented. The overhead of these can easily remove the benefit of the software cache and an implementation needs to be evaluated on each target architecture.

We also presented an acceleration method for visibility rays. These rays are needed for example for shadow computations. Using a small occluder cache we are able to exploit implicit coherence in the ray tracing algorithm to significantly speed up the computations of visibility rays. By employing a back-tracking procedure in the acceleration hierarchy we are able to quickly find neighboring occluders. This method works especially well for visibility rays as the first intersection can terminate the traversal directly. One can consider using this method also for normal (nearest) intersection rays. This would exploit implicit coherence in the same way as it does for occluders. But due to the use of ordered traversal from Section 3.3.3 there is no additional benefit to be gained. In our experiments the number of total bounding box intersections is roughly the same when comparing the use of back tracing with caching to ordered traversal. But the overhead of the ordered traversal is lower and thus the overall rendering time is faster when no entry point caching is used for normal intersection rays.

6 Practical Physically-Based Image Synthesis

In this chapter we will develop and describe a novel and full featured Monte Carlo global illumination system that uses several distinct algorithms to solve the rendering equation (see below) in an efficient way. This system allows to create production quality global illumination images that are smooth and can be computed fast.

After we discussed the fundamental rendering techniques in Section 6.1 we will introduce a novel stopping criterion for Monte Carlo integration in Section 6.2. We will then introduce the core concepts of our novel rendering system in Section 6.3. The system splits the rendering equation in several parts and solves each part individually by reusing information where appropriate. The resulting algorithm can handle all light paths and produces much smoother images than previous physically based rendering systems. Since the rendering system is based on ray tracing for all parts it uses all the fundamental acceleration strategies developed in Chapters 3, 4 and 5.

In Section 6.4 we will solve the problem of sampling environment in the context of physically correct light tracing. In Section 6.5 we address the problem of interactive preview and present a novel image space filtering method for Monte Carlo ray traced images. It is based on the À-Trous wavelet transform and can filter the noisy input images of path traced images very well. Finally, in Section 6.6 we present the integration and results of on demand spectral rendering with our ray tracing acceleration methods presented in previous chapters.

6.1 Principles of Light Transport Simulation

The primary task of computer graphics is the computation of an image. Usually this image is represented on a rectangular lattice (see Section 2.1) where each element is a pixel on the screen but during image computation other representations may be chosen to allow for filtering and sample weighting for an improved perceived image quality. An overview of image filtering in the context of image synthesis can be found in [Glas 94]. The principal goal of physically based image synthesis is to solve the rendering equation for a correctly modeled scene and a predefined camera setup.

The rendering equation was first formulated by [Kaji 86] and describes the steady-state distribution of the light energy in each surface point of the sceneboundary B. It can be written as an integral over all incoming directions (S^2)

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{S^2} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) \langle n_x, \omega_i \rangle d\omega_i \tag{6.1}$$

where $L(x, \omega_o)$ is the outgoing or exitant radiance at a point $x \in B$ in the direction ω_o. It consists of two parts. The first is the radiance L_e that is emitted at this point. Surfaces where each point emits are often simply called light sources in the context of renderer development. The second part is the energy reflected at the point x in direction ω_o. This is computed by summing (integrating) all incoming light L_i at the point x and weighting it with the incoming cosine $\langle n_x, \omega_i \rangle$ and the BRDF f_R that describes the reflection behavior of x.

In this section we describe in some detail the fundamental rendering methods used in physically based rendering based on ray tracing. It is an overview of the related fundamentals that we later need to develop our own full rendering system in Section 6.3 and are used also in the subsequent sections. Some of these algorithms were used in Chapter 3 for a realistic performance analysis of the QBVH ray tracing acceleration structure.

Light Path Classification

In [Heck 90] Heckbert introduced a simple notation to describe different types of light paths in a scene. It is based on regular expressions and uses only 4 different symbols. Each symbol stands for a specific kind of surface interaction.

L is a point on a light source.

E is a point on the camera (eye) or image plane.

D is a diffuse surface.

S is a specular surface.

A light path can only start or end on L or E. Surface interactions happen at D and S points. S usually also encompasses transmissive surfaces like glass objects. The reason for the distinction of S and D is that usually these kinds of surfaces need different handling. We will use this notation in Section 6.3 to describe the capabilities of different algorithms. The order in which a light path is defined (either from the eye or from the light source) depends on the algorithm that is described. As an example all possible light paths that connect the eye to the light source are written as $E(D|S)^*L$. All light paths that have a single diffuse bounce (also called direct lighting) are written as EDL. Note that it is in principle not important whether the path is started from the eye position E or the light position L. For example EDL could also be written as LDE. But the usual convention is to make the notation dependent on the workings of the algorithm that is described. Some light transport algorithms start to generate a path from a light source, others from the eye or camera position. To reflect this, the order is chosen accordingly.

6.1.1 Path Tracing

Path Tracing (PT) is the simplest algorithm to solve the light transport equation and can generate all possible transport paths ($E(S|D)^*L$) without special algorithmic handling. It uses the Monte Carlo method to sample light paths and compute the pixel integral in a straight forward manner. Given the scene description (with materials, camera, and light sources) and an efficient ray tracer this algorithm can be implemented in a few lines of code. The algorithm starts at the eye/camera by choosing a sampling direction (usually in pixel order) and intersecting this ray with the scene. Next the surface point is checked for emittance properties. If it is a light source the illumination into the incoming direction is evaluated and weights with the current path weight. This weight starts with the camera initialized weight and adjusted upon each bounce. Next the BRDF at this position is evaluated to generate a new sampling direction and this process is repeated recursively until the path is either terminated via Russian roulette or by leaving the scene. The contribution of each path is added to the image using an accumulation buffer. A more formal description of the algorithm can be found for example in [Dutr 06].

This method is the simplest Monte Carlo sampler for the light transport integral (Equation 6.1) but it is also quite inefficient as it is based on randomly hitting light sources. This can work well when the light sources are large relative to the scene but fails for small light sources or heavy occlusion. A simple extension is to use next event estimation at each intersection point. Now instead of waiting until a light source is randomly hit, each intersection is connected deterministically with a randomly chosen sample point on the light source. When this connection is not occluded, which is tested with a shadow ray (see Section 5.3), the contribution is computed and accumulated. This significantly reduces the variance in most scenes but comes at the cost of a slightly more complex renderer implementation and also can not compute caustics ($L(S|D)^*SDE$ paths) anymore without further code changes. This can be explained by the fact that a deterministic connection of two points through a light refracting or reflecting surface is not possible since the direction of the ray is changed at the surface interaction point.

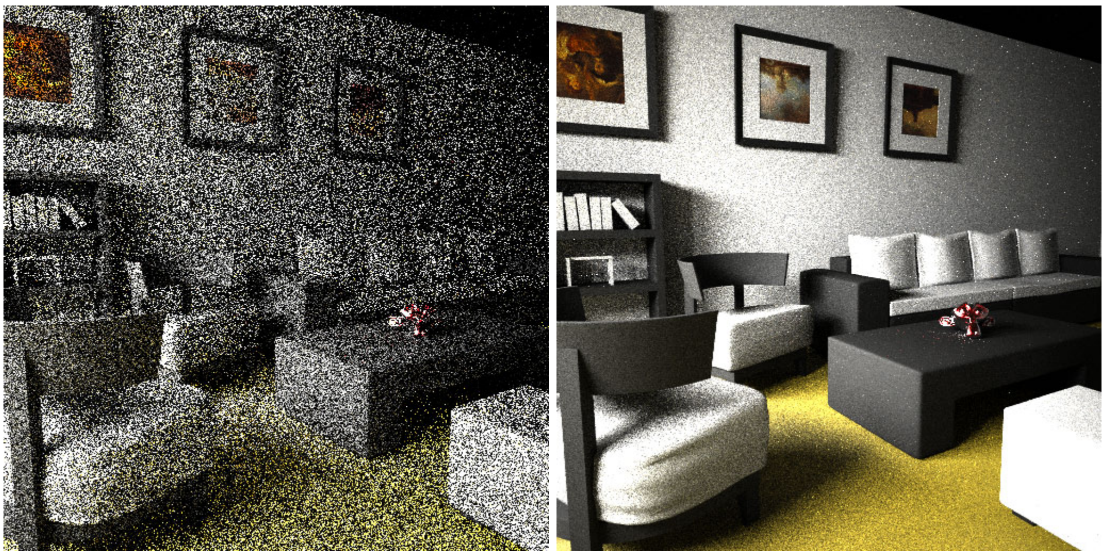


Figure 6.1: Equal time comparison ($30s$. single core) of path tracing and path tracing with next event estimation. The left shows a simple path tracer and the right the image when at each vertex a deterministic connection to a light source sample is made.

Figure 6.1 shows an equal time comparison of these two algorithms. Even though the light source in this scene was quite large the deterministic computation gives much better results in this case. Since the algorithm basically samples the direct lighting at each vertex of an eye path it is also called PTDL (Path Tracing with Direct Lighting).

6.1.2 Light Tracing

Light tracing can be seen as the same algorithm as path tracing (Section 6.1.1) but instead of starting at the camera, the paths are started at the light sources. This requires that samples can be generated on light sources but the random walk through the scene is the same as before. Since often a simple pin-hole camera model is used, randomly hitting the camera is not possible. Thus a direct connection is made at each vertex of the random walk instead that is very similar to the next event estimation in the path tracing algorithm. It is important to note that with this connection it is not possible to capture for example mirrors or transparent objects using a pure light tracing algorithm ($L(S|D)^*SE$ paths are missing).

6.1.3 Photon Mapping

Photon Mapping is a technique that is able to approximately solve all light paths in a realistically modeled scene. It can generate images with for example caustics, diffuse inter-reflections, and subsurface scattering. It was published in [Jens 96a, Jens 96b] and subsequently many improvements and extensions were developed. An in-depth discussion of the photon mapping technique and it's applications can be found in [Jens 09] and in [Jaro 08]. Since some of the fundamental ideas of photon mapping are later used in Section 6.3 to develop a new caustic rendering method we discuss the photon mapping algorithm here in more detail. Additionally, the technique described in Section 6.4 is an example of a technique that could directly be used in a photon mapping based renderer and allows one to correctly generate the required maps from environment maps.

Photon mapping is a two pass rendering algorithm that generates in a first pass a camera independent representation of incident illumination on all surfaces. This representation is stored in a data structure that is called the photon map and that is based on a point *k*d-tree [Bent 75]. This data structure allows for an efficient query of nearest neighbors and is then used in a second rendering pass from a specified camera to generate the final image via density estimation.

Photon Tracing

In the first step the photon mapping algorithm distributes photons in the scene. Random walks are started from the light sources and are terminated via Russian roulette. This is the same procedure that is used in the light tracing algorithm (see Section 6.1.2) or in the distribution of point light sources in the instant radiosity algorithm (see Section 6.1.4). Each photon that hits a diffuse surface is stored in the photon map data structure described below. Depending on the application there may by multiple photon maps each storing a different subset of the photons. For practical applications there are usually at least two different photon maps. One storing the diffuse direct and indirect illumination paths (LDE, $L(S|D)^*DDE$) and one storing caustic paths ($L(S|D)^*SDE$). Multiple photon maps account for the fact that different light paths may need different parameters and numbers of photons for a visually smooth appearance. In the case of participating media and sub-surface scattering a third photon map containing photons in the volume is used.

Photon Storage

For high quality image rendering a huge number of photons needs to be stored in the photon map (at least several million photons but quality depends on scene and target resolution). So the photon map needs a memory efficient storage and also a high speed data structure to quickly access nearest neighbors for the density estimate in the rendering pass. In the original implementation a balanced *k*d-tree is used as described in [Bent 75]. Using a balanced *k*d-tree has the advantage of minimal storage requirements since only the original photons with the position data and two additional bits for the split planes are needed. Of course the photons can be stored in any spatial data structure that allows for efficient range queries. In [Wald 04b] the authors explicitly unbalance the *k*d-tree to achieve faster look-up speeds. Inspired by the surface area heuristic (SAH) used to construct ray tracing acceleration structures (see Section 4.1.3) they derive a heuristic called Voxel Volume Heuristic to build these trees.

Rendering with the Photon Map

To compute an image with the previously generated photon map the rendering equation 6.1 is rewritten using the correspondence that the incident radiance L_i at a surface point is

$$L_i(x, \omega_i) = \frac{d^2\Phi_i(x, \omega_i)}{\langle n_x, \omega_i \rangle d\omega_i dA_i}$$

where Φ_i is the incoming flux. This information can be approximated using the photon map. Equation 6.1 can now be rewritten as

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega_x} f_r(x, \omega_i, \omega_o) \frac{d^2\Phi_i(x, \omega_i)}{dA_i}$$

The incoming flux Φ_i is now approximated with the photon map using a density estimation. The estimate locates a predefined number n of the nearest photons to x and estimates the flux. Each photon p has the discrete associated power $\Delta\Phi_p(x)$ and so the estimate can be written as a sum:

$$L(x, \omega_o) \approx L_e(x, \omega_o) + \sum_{p=1}^{n} f_r(x, \omega_i, \omega_o) \frac{\Delta\Phi_p(x)}{\Delta A}$$

With the fixed n a circular area is assumed that has the radius r that is the distance of the farthest photon in the estimate. So the area is $\Delta A = \pi r^2$ and the full equation is

$$L(x, \omega_o) \approx L_e(x, \omega_o) + \frac{1}{\pi r^2} \sum_{p=1}^{n} f_r(x, \omega_i, \omega_o) \Delta\Phi_p(x)$$

This estimate has a practical problem that needs to be considered when using a photon map for global illumination computations. Since the photon map stores only points on surfaces it has no notion of the geometry that lies in between. Consider for example to adjacent rooms with a thin wall. Estimating the flux close to this wall will also consider stored photons from the neighboring room resulting in light leaks. These result in visible light seems, especially if one of the rooms is not illuminated at all. A non-illuminated room shows also a second problem. In a non-illuminated room are no photons and the basic algorithm tries to expand the search radius until n photons are found. So light leaks might not only be visible at corners but also far away from any illumination. Since the search radius in these cases is quite large, the resulting flux is quite small, but when using HDR rendering and automatic tone mapping this illumination still becomes visible in a final image.

Algorithmic Details

Here we discuss some of the basic implementation details of a photon map. These principles are used later in Section 6.3.3 in a similar way to store the eye points of the histogram method.

The most memory efficient way to store the *k*d-tree of photons is a balanced *k*d-tree. It removes the need to store explicit child pointer as the tree root is element 1 and for each element i the two children are stored at $2i$ and $2i+1$. This storage type is similar to a heap data-structure [Sedg 92]. The construction of a balanced (point) *k*d-tree works the following way: First a splitting dimension is selected (usually based on the largest extent of the axis aligned bounding box of the active point set). Next the median of the points in this axis direction is found and the array split into the points left to it and the points right to it. The median is the new node and the two partial arrays are used to recursively continue the construction.

During rendering this *k*d-tree is then used to locate the n nearest photons to a given 3d query location point. This is best done using a max heap with n entries. The generic search begins at the root node and adds elements to the heap during traversal. As soon as the heap is full the largest distance can be used to cull the traversal. Upon traversal completion the heap is guaranteed to contain the n nearest photons to the given query location. An important optimization for this strategy is to start already with a bound on the maximum distance to allow for early culling and only when not enough photons were found to restart the search unbounded.

Final Gathering

Final gathering is an additional indirection step [Chri 02] when rendering with the photon map to address the low frequency artifacts that are usually visible. Instead of directly evaluating the photon map at a primary hit point the incident illumination is computed with a hemispherical integral. For this a Monte Carlo method is used and the hemispherical integral approximated with several samples. The photon map is used at the intersection points of these samples only. This can be seen as an additional smoothing step as another integral is involved. It also solves the light leaking problem but is a very expensive operation as many additional rays are needed for a noise free result. An improvement to final gathering was presented in [Havr 05].

One problem with final gathering occurs in regions of concave surfaces like corners. There, the final gathering rays are highly local and cannot gather a good approximation from the photon map. This problem is solved for example in the Mantra Renderer [1] by using the photon map approximation for the final gather rays only when they are far enough away. If it would be gathered from a too close surface point, simple path tracing is performed. This problem was already observed and discussed in [Atki 00] where the radiosity equation has a singularity in corners. The basic idea is also called near-field and far-field separation and is used in [Arik 05] in the context of final gathering. This

[1] Houdini Mantra Renderer (PBR module), http://www.sidefx.com

is also similar to the robust global illumination approach [Koll 04] where the geometric term is clamped and the missing illumination gathered by additional ray tracing.

6.1.4 Instant Radiosity

The Instant Radiosity algorithm was presented in [Kell 97]. It is one of the fundamental algorithms we use in Section 6.3 to develop efficient and fully featured progressive global illumination. We will also discuss some of its extensions and enhancements that were developed by other authors over the years since its release.

The basic algorithm generates an approximation to the diffuse radiance in a 3d scene using an accumulation buffer [Haeb 90]. Since it is a radiosity method it assumes that all surfaces have perfect Lambertian reflection behavior. The image generation process starts by tracing particles from the light sources through the 3d scene. At each interaction point on the surface a virtual point light source (VPLS) is created. This point light source illuminates the scene visible from the intersection point. In the original work this illumination was computed using graphics hardware as it allowed for very fast shadow computation using shadow maps. The result of this illumination computation is then added to the accumulation buffer and the process is repeated until a predefined number of point lights is generated. Due to the progressive nature of this algorithm (the random walk can be performed continuously) this method describes the full global illumination solution for a diffuse scene. Included in [Kell 97] is also a short discussion on extending this method to specular reflections. For this special light sources that illuminate only parts of the scene are created.

One problem though is the geometric term that occurs in the computation of the contribution of one point light. It grows rapidly when the distance of the VPLS and the point that needs to be illuminated get close together and has a singularity (i.e. a division by zero) when the two points coincide. This is mathematically correct but has two practical problems. The first one is an increased variance that is especially noticeable in corners of a scene. The second problem are numerical issues. The accuracy of floating point arithmetic is limited and the distance between two floating point numbers increases with their value. This can result in computing a wrong image when using an accumulation buffer. While it is mathematically correct to compute these large values accumulating them leads to the problem that previously accumulated (small) values might be completely eliminated. The second, related, problem is that once these large values are in the accumulation buffer they will only slowly be reduced with increasing number of passes. Thus additional (small) values computed in subsequent passes might not end up in the accumulation buffer.

6.2 Hierarchical Monte Carlo Integro-Approximation

Progressive image synthesis, in contrast to final frame rendering, produces an image incrementally. When starting the rendering process a first image is shown very quickly.

This image is then refined progressively. With a good progressive rendering algorithm (it has unconditional convergence) this form of image generation has many advantages over final frame rendering. First, the user gets very quickly a rough preview of how the final image would look like. Second, a progressive algorithm is automatically adaptive to complex illumination situations. For a final frame renderer the user has to set many parameters correctly before the image computation starts. In a progressive algorithm the user just has to wait until the features are shown correctly in the final image. Algorithms that are based on single-pass Monte Carlo integration (see below) are usually progressive since it is natural to just add more samples to the final image. Multi-pass algorithms like photon mapping (see Section 6.1.3) are in their basic implementation not progressive. Note that any progressive algorithm can be made a final frame algorithm by letting the user specify a fixed amount of time. This comes with the disadvantage of not having a guaranteed quality. In Section 6.2.2 we therefore propose a stopping algorithm which robustly separates the remaining noise in the rendered image from the variation due to the scene content, which is easy to implement, requires very little additional memory and we also show how to use it to adaptively sample from the light sources.

6.2.1 Monte Carlo Integration and Monte Carlo Integro-Approximation

The Monte Carlo method (see for example [Kalo 86] for an introduction) can be used as a numerical tool to solve integrals using random numbers. To compute an approximation to the integral of a known function $f(x)$ in the s-dimensional interval $I^s := [0,1]^s$ the Monte Carlo method uses a set of n independent uniformly distributed random numbers $x_i \in I^s$ and computes

$$\int_{I^s} f(\mathbf{x}) d\mathbf{x} \approx \frac{1}{n} \sum_{i=0}^{n-1} f(\mathbf{x}_i)$$

For this kind of integration a probabilistic error bound can be stated [Nied 92]. The error is less than $\frac{3\sigma(f)}{\sqrt{n}}$ with a probability of:

$$P\left(\left\{\left|\int_{I^s} f(\mathbf{x}) d\mathbf{x} - \frac{1}{n} \sum_{i=0}^{n-1} f(\mathbf{x}_i)\right| < \frac{3\sigma(f)}{\sqrt{n}}\right\}\right) \approx 0.997.$$

A simple but very important observation can be made in this equation. Due to the square-root of the number n of samples a reduction of the integration error by a factor of 2, 4 times as many samples are needed. In the context of Monte Carlo methods for rendering a good introduction can be found in [Veac 97a] and in [Dutr 06].

As described in Section 2.1, an image is stored as a two dimensional plane of pixels. With Monte Carlo integration the color of a pixel in a final image is determined by projection of the image function $f(\mathbf{x})$ onto each pixel

$$I(k,l) := \frac{1}{|P_{k,l}|} \int_{P_{k,l}} f(\mathbf{x}) d\mathbf{x}.$$

where $I(k,l)$ is then the average radiant flux through the pixel (k,l) and the integral is over the pixel area. This allows to formulate the problem as an integro-approximation problem because the image can be considered as functional:

$$I(k,l) := \int_{I^s} f(\mathbf{x},k,l)d\mathbf{x} \approx \frac{1}{n}\sum_{i=0}^{n-1} f(\mathbf{x}_i,k,l).$$

Now we have a parametric integral for the image function $I(k,l)$ that computes the average radiant flux for each pixel simultaneously [Kell 01]. In the next section we will use this notion to derive an iterative termination criterion for Monte Carlo image synthesis.

To use the Monte Carlo method in a computer program, a fast source of random numbers is needed. Since the creation of real random numbers using a computer program is impossible pseudo-random number generators are used instead. In [Knut 81] a very thorough introduction into the topic of computer generated random numbers and the testing of their quality is given. In this thesis we use the Mersenne Twister algorithm [Mats 98] as the source for all random numbers that are needed. Where appropriate we also use the SIMD optimized version [Sait 08].

6.2.2 Iterative Block Partitioning and Termination

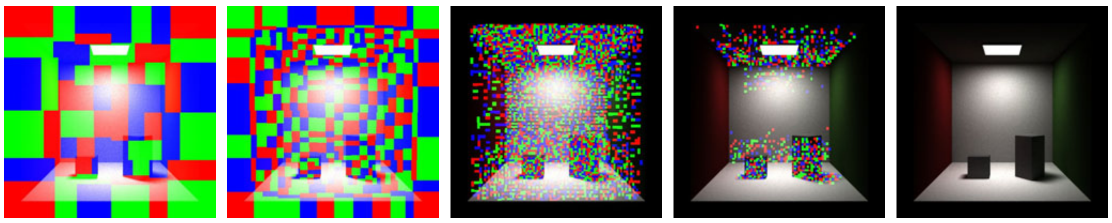


Figure 6.2: An example sequence of blocks. Initially large blocks (shown in saturated colors) are recursively refined and terminated individually as the image converges. Note that complex areas in the image, where indirect illumination dominates the appearance (e.g. the ceiling), are detected and refined by the algorithm.

In this section we introduce a hierarchical image-space method to robustly terminate computations in Monte Carlo image synthesis, independent of image resolution. Parts of this are published in [Damm 09b]. The technique consists of a robust convergence measure on blocks which are either recursively subdivided or terminated independently, using a criterion which separates signal and noise based on integral estimates from two separate sample sets. The criterion takes into account that the human visual system is less susceptible to small color differences with high luminance. The technique can be easily implemented, as the evaluation of the error measure only requires a second framebuffer and a list of non-terminated blocks. Based on the stopping criterion, one can furthermore sample both the image plane as well as the light sources adaptively. Adaptive sampling reduces the number of samples required to gain the same root mean square error to one quarter in some of our test cases.

Following the idea of Dippé and Wold [Dipp 85], we estimate the rate of change as the difference between images generated with two different sample sets. A lot of work has been done to quantify the perceived differences by the human visual system [Mysz 98, Boli 98, Rama 99, Yee 01, Sund 04, Farr 04, Sund 07, Daly 93, Mant 05], error measures based on confidence intervals, contrast, variance, saliency and entropy have been investigated. For simplicity and fast evaluation, we base our stopping criterion on the remaining color noise in relation to the logarithmic luminance of the sample, but any other error measure could be used as well. Rigau et al. [Riga 03] presented a similar method based on entropy, but they only consider the image as a whole and sub-pixel refinements, not blocks of pixels. Hachisuka et al. [Hach 08a] also subdivide a hierarchy of samples very similar to the MISER algorithm [Pres 92], but our error measure is based on two integral estimates, which is able to separate signal from noise. Also, additional reconstruction is not necessary because sampling is dense enough (at least one sample per pixel), and the samples do not have to be stored explicitly in our case.

In [Over 09] the authors describe an adaptive wavelet rendering that is similar to our approach but their scaling functions are more restrictive than our blocks and additionally we also adaptively sample the light source and are not restricted to 2d.

For a reliable error estimate, it is not sufficient to look at the variation inside a single pixel, due to the limited number of samples [Mitc 87]. To overcome this, we evaluate the error measure on a hierarchy of blocks, which are recursively refined only as the remaining error drops below a certain resolution-independent threshold (see Figure 1 for an illustration of the refinement process). This makes sure the algorithm adapts to the true signal, not a noisy estimate. Assuming that the samples available so far have not completely missed any major light contribution (i.e. one “firefly”-path is enough), our scheme operates conservatively. Painter and Sloan [Pain 89] also used a hierarchy, an image-space *k*d-tree, but explicitly stored all samples and all levels of the hierarchy. We, on the other hand, simply store the integral estimates of the two sampling sets per pixel in two framebuffers. Furthermore, we only keep track of the leaf blocks, which significantly simplifies bookkeeping. Based on our convergence estimate, adaptive sampling can be performed by simply supersampling only those blocks which are not yet terminated. Note that this is different to most existing adaptive sampling approaches, since it is not based on placing samples in regions considered interesting at an early stage but splits and terminates rather conservatively. This is why the common problem of missed features such as small objects is not a problem. Furthermore, as the technique is framebuffer-based, it is completely independent of the underlying rendering algorithm.

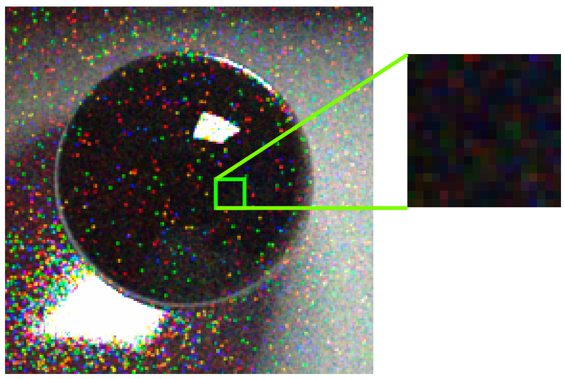


Figure 6.3: A caustic rendered with way too low sampling density. It is not sufficient to make decisions based on only a few samples, i.e. on small blocks or even pixel-wise. The magnified block for example appears to be converged already, but it still misses important features (evident through adjacent bright pixels) and thus needs more samples.

Algorithm

To make sure the algorithm does not stop computations before all important features have been detected, a lot of samples should be drawn before making a decision (see Figure 6.3 for an illustration, and Figure 6.4 for a comparison to a pure per-pixel approach). Since it is not desirable to shoot a lot of possibly unnecessary rays per pixel, we initially base the evaluation of the error on the image as a whole.

As the variance in the image will be distributed unevenly, depending on the problem and the type of path space sampler used, the stopping decision should be done locally. This is reflected by splitting the blocks when the algorithm has enough confidence not to have missed any important features, i.e. the per-block error measure drops below a certain threshold ε_s. A block only consists of an axis-aligned bounding box covering an area of the image. The blocks are managed in a linear list. That is, if a block is split, it is simply removed and the two resulting child blocks, disjointly covering the same area of the image, are added to the list again. This way, no explicit hierarchy has to be maintained. The algorithm continues by drawing new samples (one for each pixel in each remaining block in our implementation). If stratification in image space can be guaranteed, e.g. by using a simple backward path tracer, only non-terminated blocks have to be sampled further. If certain sampling methods cannot be restricted in this way, such as for example light tracing methods, their contribution can still be used, as the stopping condition is not directly coupled to adaptive sampling. In fact, it is also possible to sample adaptively from the lights (see Section 6.2.3).

The image is converged when the error of all blocks is smaller than a threshold $\varepsilon_t < \varepsilon_s$. For an overview of the algorithm see Figure 6.5. An example sequence of blocks is depicted in Figure 6.2. Note how the algorithm automatically adapts the block size to the image and is thus independent of the resolution.

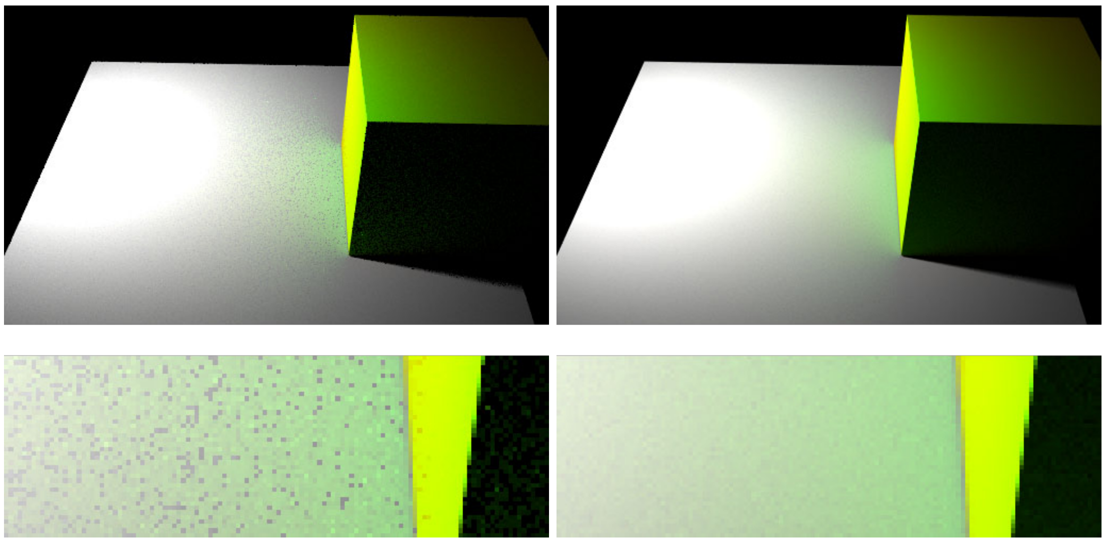


Figure 6.4: This shows the problem of using a per pixel termination criterion. Due to the nature of Monte Carlo sampling some pixels terminate too early which is clearly visible in the left image. The right image shows our proposed termination criterion with equal sample count.

Error Metric

We use an error metric based on the pixels in the final image, but to get a robust criterion we always evaluate a block of pixels. We compare two independently computed images with the same number of samples similar to [Dipp 85, Dmit 04].

In the implementation, the evaluation of the error metric can be simplified by introducing just a single additional accumulation buffer and keeping the normal accumulation buffer used to compute the final image. In the second buffer we accumulate only samples every second rendering pass. This is based on the simple observation that computing a single image I with an even sample count can be split into two images A and B with $I = A/2 + B/2 \Rightarrow B/2 - A/2 = I - A$. Thus, using an RGB buffer we estimate the per pixel error as

$$e_p = \frac{|I_p^r - A_p^r| + |I_p^g - A_p^g| + |I_p^b - A_p^b|}{\sqrt{I_p^r + I_p^g + I_p^b}}.$$

The square root in the denominator is motivated by the logarithmic response of the human visual system to luminance. The term here behaves similarly, is easier to evaluate and was found to yield slightly better results. The error per block is computed by summing over all pixels:

$$e_b = \frac{r}{N} \sum_p e_p$$

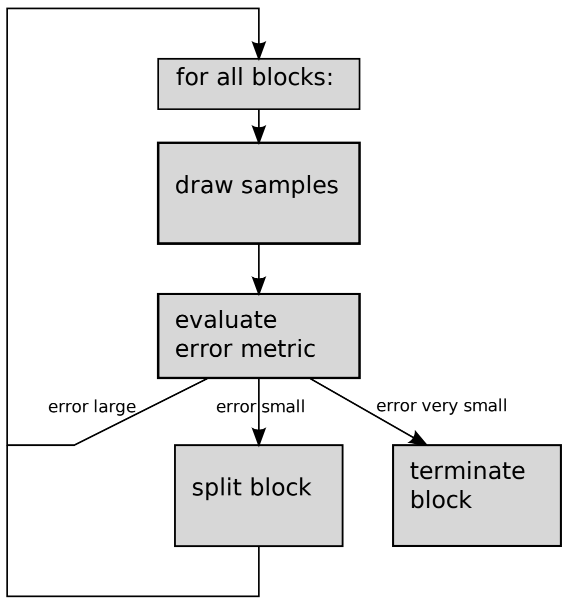


Figure 6.5: This flowgraph illustrates the integration of our stopping algorithm into a rendering system.

where N is the number of pixels contained in this block and r is a scaling factor computed as $\sqrt{A_b/A_i}$. A_i is the area of the image, A_b the area of the block under consideration.

Block Splitting and Block Termination

The user specifies a single error value v that is used in all further decisions. From this user-specified value we compute two error thresholds ε_s (splitting) and ε_t (termination). For simplicity we use $\varepsilon_t = v$ but it is of course possible to rescale v to a more intuitive parameter range. In all our tests we used $v = 0.0002$. Given ε_t, we compute $\varepsilon_s = 256 \cdot \varepsilon_t$. This is an empirical choice that worked well in all our tested scenes. The splitting is performed axis-aligned by choosing the axis where the block has the largest extent. The split position is chosen such that the error measure is as similar as possible on both sides.

6.2.3 Adaptive Light Tracing

We now show how to extend our criterion to also be able to adaptively sample in a light tracing setting (see Section 6.1.2). A light tracer starts paths only from the light sources and connects hit points to the camera. This is very well suited to compute caustics but it is not possible to directly refine samples on the image plane as it is unknown where exactly a light sample will connect. For our termination criterion this is not a problem as it works equally well independent of the used rendering algorithm. Adaptive sampling of the image plane is not straightforward for light tracing, so all samples need to be distributed uniformly.

To facilitate adaptive sampling also from the light sources, we back-project the image error metric to an importance map around each light source. Such an importance map consists in a quantized hemisphere around each emitting triangle, thus representing only outgoing directions, ignoring the starting point on the triangle and higher-dimensional bounces. This works well under the assumption that triangles are small with respect to the illuminated area. This can always be achieved by subdividing the emitting triangles which does not change the final image. We chose this approach over explicitly storing the starting point as well to reduce the dimensionality and thus the size of the map. This also simplifies the integration into an existing rendering system. The algorithm proceeds by calculating an initial image calculating a few samples. After that, each new light sample accumulates the remaining image error back into its importance map bin at the light source side. This way, the image error metric is back-projected with a delay of one iteration. The importance map can now be used in the next iteration to importance sample light directions (similar to [Clar 08]).

As test scene we chose a distorted glass object casting a large caustic (see Figure 6.6). The top row of Figure 6.7 shows three importance maps at three different iterations. The bottom row shows the associated image space error blocks. While the top row is heat-map colored by the importance, the colors in the bottom row are solely to distinguish the different blocks. It can be clearly seen how large areas of the hemisphere do not contribute at all to the final image and also how the termination of regions in image space is reflected in the hemispherical importance map. Figure 6.8 shows the RMS error graph using no adaptivity at all and using the described back-projected importance map. In this case the number of samples required to get an RMS error below 0.01 is reduced from $570M$ light paths to $160M$ light paths (28%).

6.2.4 Results and Discussion

We now analyze our proposed stopping criterion for different scenes and additionally examine how well the proposed error metric can be used as an adaptive sampling criterion.

Figure 6.10 on the left shows a simple test scene of a diffuse sphere illuminated by an area light source casting a large smooth shadow. The right image shows the distribution of the number of samples in the final image as a heat map. White is the maximum number of samples and black the minimum. As each block is sampled uniformly the block structure is still apparent in this image. The graph shows the number of samples required by the normal rendering algorithm and by our technique to achieve an RMS error below 0.01. In this simple scene the sample count was reduced from $700M$ to $400M$ (57%). This indicates that our error metric is meaningful also with respect to the RMS error. The evaluation of the error metric costs about 10% of the rendering time in our implementation.

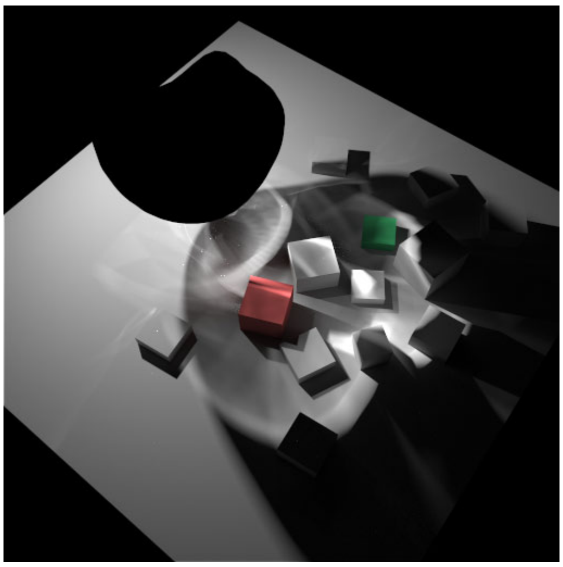


Figure 6.6: The light tracing test scene. The glass object is black because a pure light tracer cannot deterministically connect singular materials to the camera.

As another example we show a living room scene in Figure 6.11. Again on the left the final image can be seen and on the right the sample count distribution. This is a more realistic scene as it is closed and strong indirect illumination is present. The complexity is distributed almost equally over the whole image. This explains why in this scene sampling adaptively does not present a huge benefit. This is also visible in the RMS error graph. Still our termination criterion works robustly and also isolates the two small areas in the image where the illumination is more complex due to glossy objects and a small caustic cast by the monkey head on the table (see Figure 6.9).

In summary we introduced in this section a stopping condition for Monte Carlo image synthesis which automatically adjusts itself to image resolution, robustly adapts to local variance, and is very simple to implement on top of any rendering system. Due to this simplicity it can be easily integrated in GPU based ray tracing systems. The criterion can also be used to facilitate adaptive sampling. Additionally, we have shown how back-projection of the image error metric onto hemispheres around the light sources can be used to profit from adaptive sampling also when starting paths at the light sources.

In the future, it should be possible to extend the system to complete bi-directional light transport. That is, combine adaptive image space sampling and starting the light paths in important directions based on the back-projected image space error at the time and also consider all deterministic connections. It would also be interesting to extend the method to higher dimensionality, similar to Hachisuka et al. [Hach 08a], i.e. to remove the dependency on the framebuffer. Furthermore, a mathematical analysis of the variance in path space should be done to gain some knowledge about the worst case integration error at a given sampling density. This way, the initial sampling rate

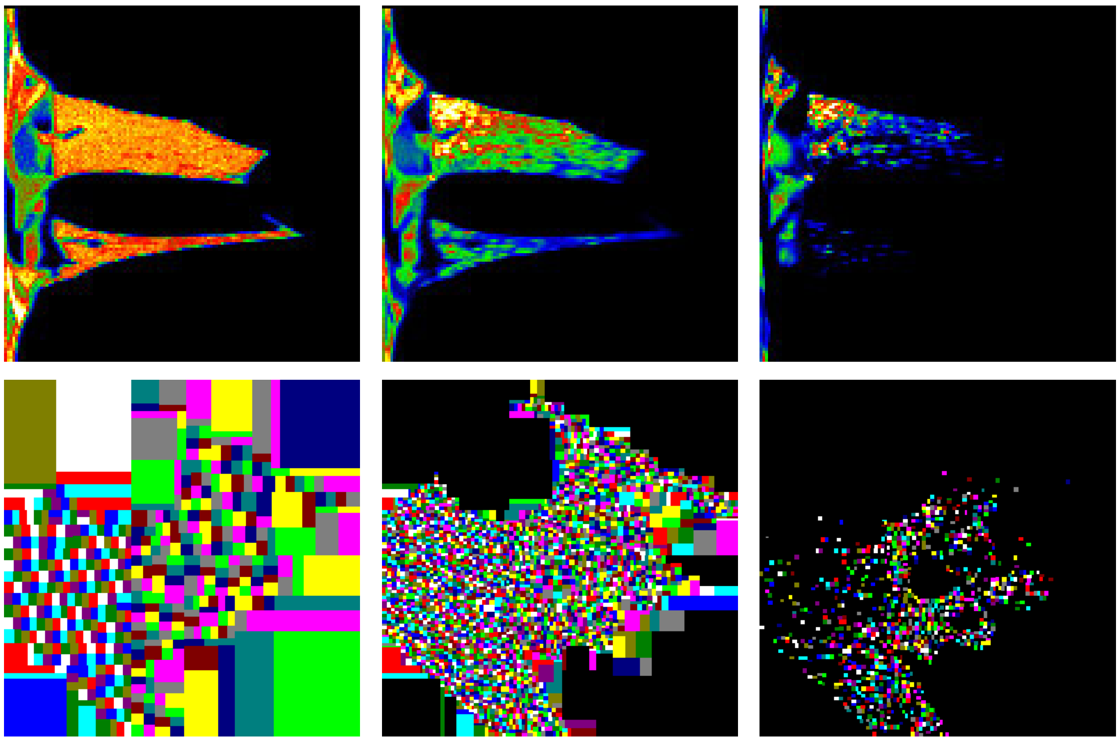


Figure 6.7: Hemispherical importance map at the light source (top row) and respective image error blocks for the caustic scene (Figure 6.6).

required to assure no features are missed when first evaluating the error measure could be determined.

The adaptivity of the method could also be improved upon, if one does not only wish to use it as a termination condition, but also increase efficiency. In analogy to building fast spatial acceleration hierarchies for ray tracing, the image space block splits could be optimized to cut off empty space, i.e. always separate blocks which are likely to terminate in the next iteration.

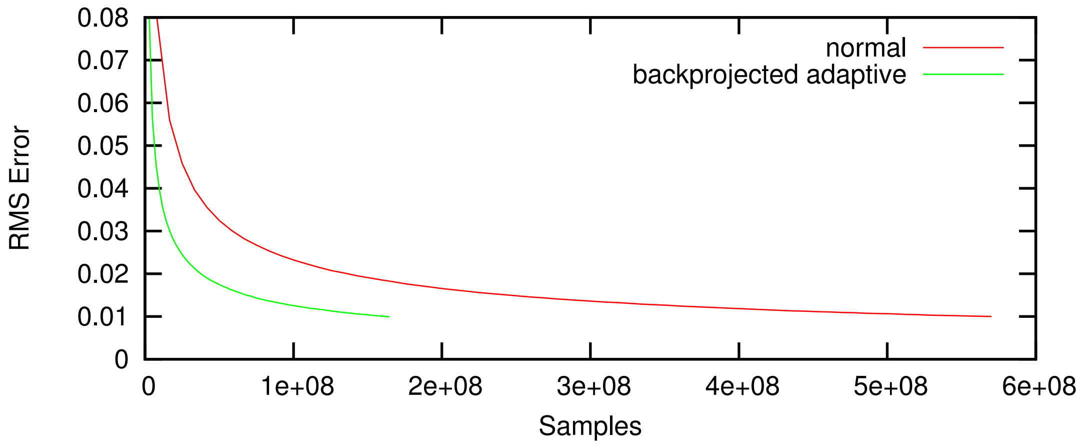


Figure 6.8: RMS error graph for the light tracing experiment comparing no adaptivity and our proposed back-projection. The curves terminate when the RMS error drops below 0.01.

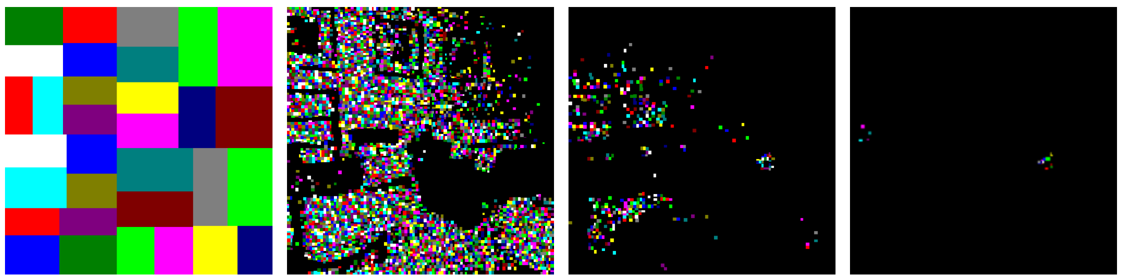


Figure 6.9: Four images taken from the sequence of active blocks when rendering the living room scene seen in Figure 6.11. The difficult spots are isolated quickly.

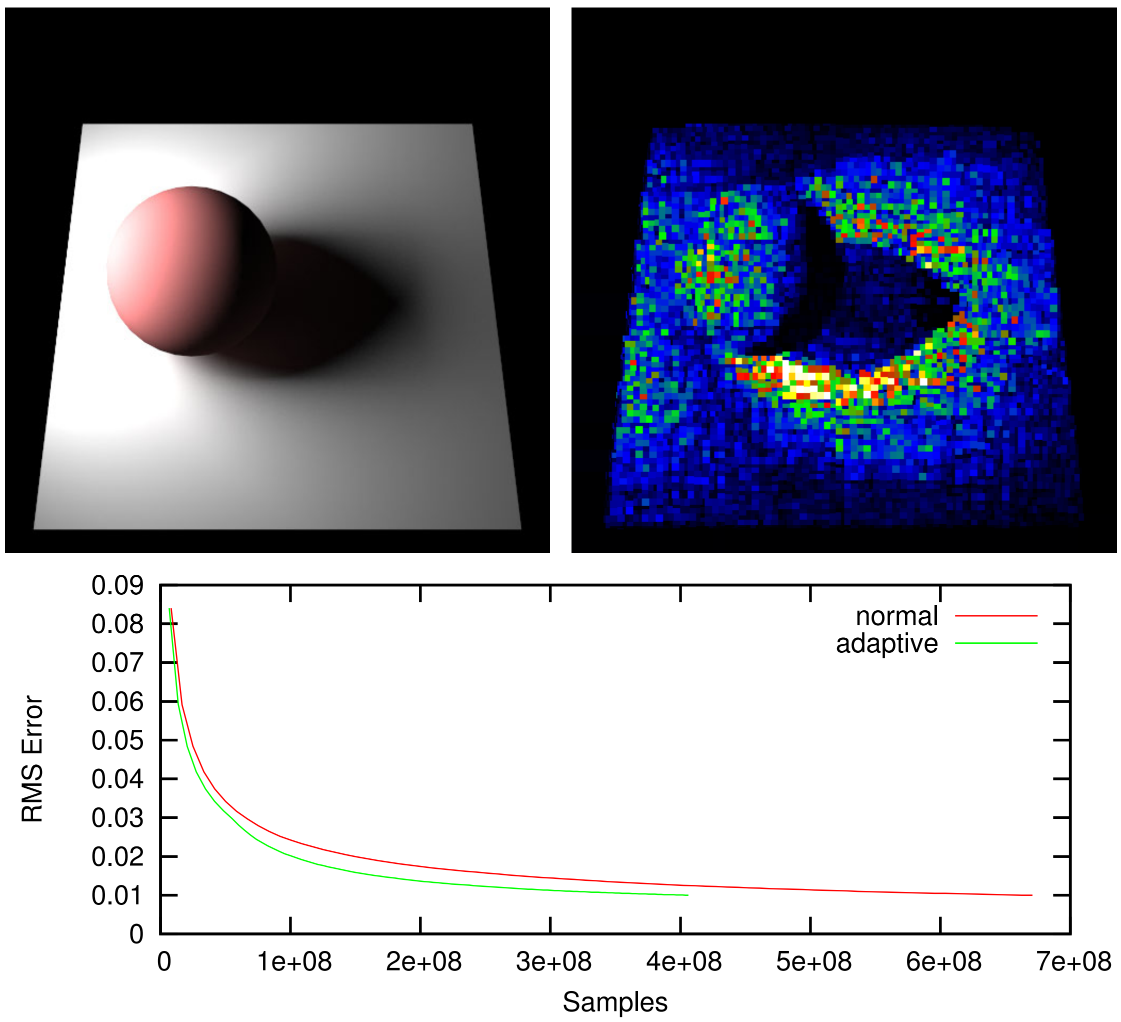
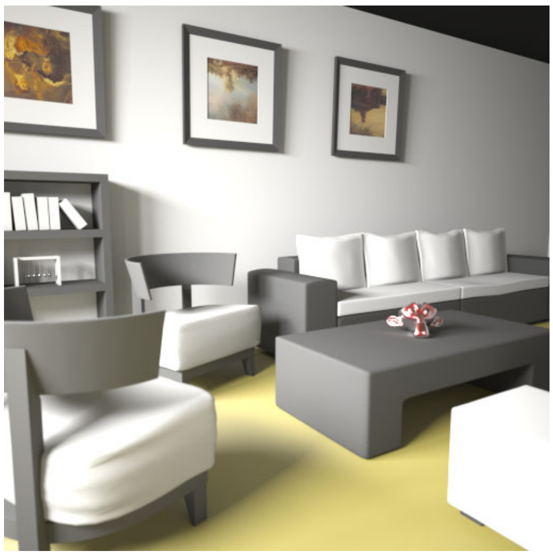
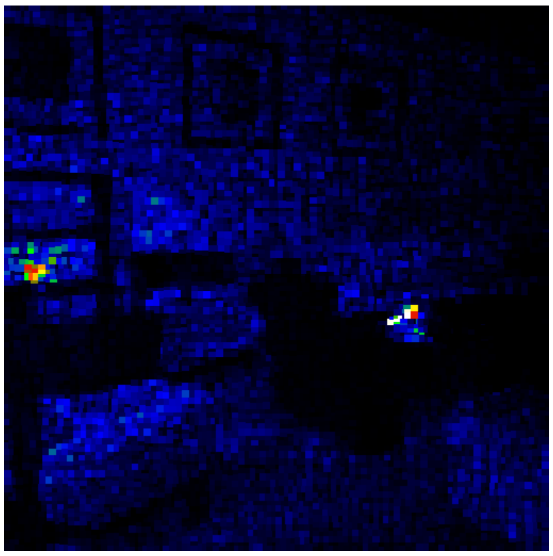


Figure 6.10: A simple sphere scene rendered with the proposed stopping condition (left) and the respective heat map (right) representing the number of samples required for a converged block. Most effort has to be spent in the penumbra regions. The graph shows the RMS error, with increasing number of samples, for normal and adaptive sampling.





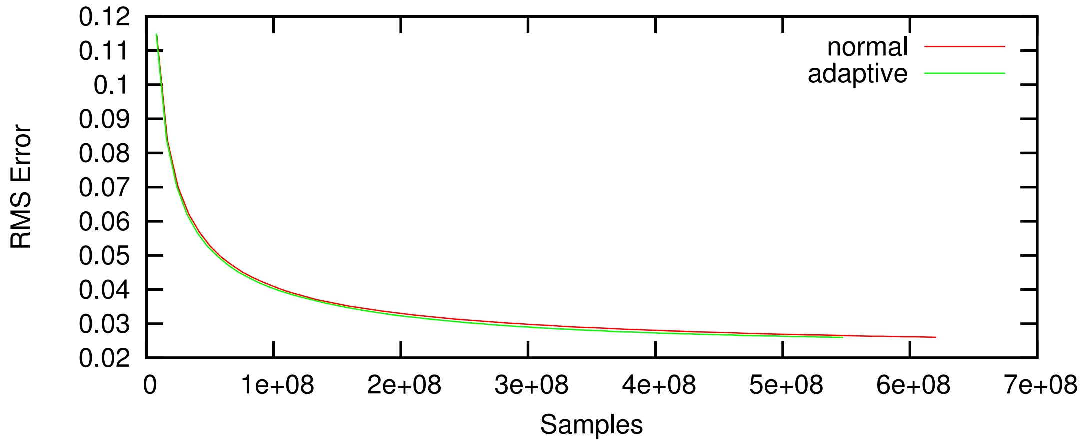


Figure 6.11: Final image (left) and sample density heat map (right) for the living room scene. The algorithm robustly finds the two spots with significantly higher variance than the rest of the image. The graph shows the RMS errors of normal and adaptive sampling with increasing number of samples.

6.3 Path Space Partitioning

In this section we present a full featured physically-based progressive global illumination system (parts of this are published in [Damm 10a]) that is capable of simulating complex lighting situations robustly by efficiently using both light and eye paths. Specifically, we combine three distinct algorithms: point-light-based illumination which produces low-noise approximations for diffuse inter-reflections, specular gathering for glossy and singular effects, and a caustic histogram method for the remaining light paths. The combined system efficiently renders low-noise production quality images with indirect illumination from arbitrary light sources including inter-reflections from caustics and allows for simulating depth of field and dispersion effects.

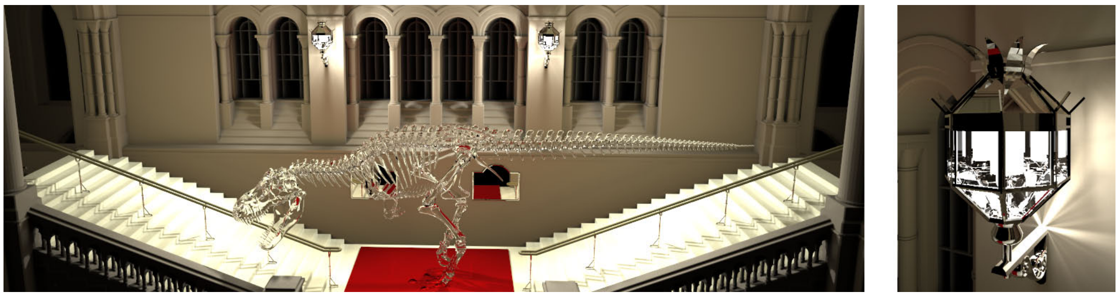


Figure 6.12: Our progressive rendering system is able to handle a wide range of complex light paths efficiently by combining three approaches. The shown scene contains diffuse and specular objects that are illuminated only by two lamps enclosed in glass. Our system not only handles the bright areas well but allows for smooth indirect illumination from this kind of light sources without the need for parameter tuning.

The problem of noise with full global illumination for production rendering was also addressed recently [Kriv 10] in the Arnold rendering system. They solve the problem of noisy caustics with two different methods. First, artists can manually disable certain light paths. Second, the specular exponent for secondary bounces is reduced automatically. The reduction of the specular exponent for higher order bounces, which is a broadening of the lobe, can be interpreted as a kind of low pass filter. It smooths out the high frequencies in the light transport that originate from very focused reflection lobes. Arnold also limits the $\frac{1}{r^2}$ geometric term in the illumination computation near light sources. This clamping was already used in [Kell 97] but reduces the energy and produces the wrong integral. A solution was presented in [Koll 04] where the missing energy is gathered correctly by additional ray tracing.

Figure 6.12 illustrates a scene with complex illumination problems. Our system computes progressive approximations by continuously refining the solution using a constant memory footprint without the need of pre-computations or optimizing parameters beforehand. The path space partitioning we use has the advantage of allowing the use of almost arbitrary BRDFs, which is important in production rendering in movies for example. Our rendering system progressively converges to a unique solution even if the BRDFs are not physically plausible, e.g. because they do not fulfill reciprocity. So complex shader networks could be used without change.

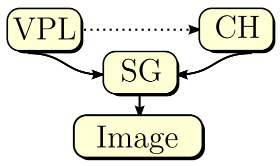


Figure 6.13: Overview of the connection of our three algorithms we combine to generate a full global illumination renderer. VPL is a point light based component that is generated first. The information that is provided by VPL is then used to generate the caustic paths in CH, the caustic histogram algorithm. Finally both outputs are used in a specular gathering algorithm to create the still missing light transport paths. Everything together generates the output image.

Each technique contributes a distinct part to the final image and together they are able to simulate for example caustics, specular surfaces, and depth of field effects. The benefit of the combination is that the techniques can take advantage of each other by reusing information where appropriate and that the advantages of each technique are retained:

- Virtual point light sources (VPL): smooth indirect illumination.
- Caustic histograms (CH): crisp, efficient caustics.
- Specular gathering (SG): robust handling of glossy and singular materials.

The caustic histogram method collects photons in bins that are created for each of the progressively sampled eye paths. Although we include approximating techniques and thus accept a certain small bias in the solution, the progressive rendering algorithm consistently converges to a unique solution, especially in complex illumination situations. We argue that a fully progressive system allows for much easier handling of the renderer by artists because the choice of parameters such as number of samples and photons does not change the final image quality (they may affect only rendering time). In addition, the proposed system decouples memory consumption and image quality (i.e. higher quality renderings do not require more memory at run-time). Compared to many previous approaches our system renders low-noise production quality images with indirect illumination from arbitrary light sources including inter-reflections from caustics and allows for simulating depth of field and dispersion effects.

Related Work

Physically-based rendering is a sub-problem of image synthesis, which aims to simulate light transport on a computer in a physically correct way. The problem has been studied in great detail [Phar 04, Dutr 06] and as it stands, ray tracing is the only method that allows to solve complex illumination problems consistently. Even though ray tracing performance has been improved greatly in the recent years [Shir 06], only few of the achievements directly help Monte Carlo based global illumination algorithms [Boul 08, Wald 08]. For that reason we developed the acceleration techniques presented in Chapter 3 to 5. Several hundreds of million paths need to be ray traced for a final image and thus Monte Carlo and quasi-Monte Carlo rendering algorithms have a very high computational demand and long rendering times. Unbiased Monte Carlo-based rendering systems are progressive by nature and thus have the advantage of robustly handling even complex situations with the only limitation being the rendering time. The disadvantage is that even in simple situations unbiased Monte Carlo algorithms have a high variance perceivable as noise in the image. In [Tabe 04] the authors describe a full global illumination system for feature films. They stress the need for full artist control of the global illumination solution and thus ignore physical correct rendering. Their system is designed for movie production and thus allow only for a fixed time frame for each computed image. To meet this speed requirement they perform ray tracing only on proxy geometry and consider only single bounce diffuse inter-reflection. Kajiya [Kaji 86] introduced Monte Carlo path tracing algorithms to computer graphics. Since then, a lot of research headed to improve the efficiency of the basic algorithm. The family of bidirectional path tracing algorithms using multiple importance sampling [Lafo 93, Veac 94] and the Metropolis light transport algorithm [Veac 97b] belong to the most powerful algorithms to date. While these algorithms are unbiased and can deal with complex lighting situations, they suffer from variance, which becomes visible as noise in images. Our system approaches the global illumination from both light and eye paths using a Monte Carlo technique, however, it reduces variance through the use of correlated point light sources for diffuse inter-reflections. The basic approach of splitting the final solution to be computed by different algorithms was also described in [Chen 91].

The Instant Radiosity algorithm (see Section 6.1.4) uses point light sources and graphics hardware to quickly compute a global illumination solution for diffuse scenes. Since then, point-light-based global illumination has proven to be a useful and efficient way to approximate diffuse inter-reflection in real time systems [Bent 03] and preview systems [Havs 07]. In [Sego 07] an extension to the Instant Radiosity algorithm is presented that uses Metropolis sampling to increase efficiency. The use of VPLs is at the core of our rendering system, however, we extend it to simulate non-diffuse light paths as well.

Photon mapping (see Section 6.1.3) has been introduced as a solution to the problem that there exist paths that cannot be efficiently sampled by any unbiased technique. With the recent improvements [Hach 08b], many shortcomings of the original method have been removed. In our approach we remove the remaining memory bound on the number of eye path vertices that need to be stored, which for example allows one to simulate anti-aliased depth of field with any sampling rate required. These problems were also recently addressed in [Hach 09]. Other recent research focuses on GPU based methods to accelerate the illumination computation [Wang 09].

The Lightcuts rendering framework [Walt 05, Walt 06] is a powerful approach to the many-lights problem, which is entirely based on point-light sources. Due to the ability to deal with an enormous number of point light sources, glossy effects can be handled, however, the system cannot simulate caustics and is not progressive. As our approach is progressive there is no limitation on the number of light sources and even the simulation of caustics is integrated.

Partitioning of Light Contributions

Our physically-based rendering system is designed to produce production quality images of 3d scenes with global illumination in complex situations without the need for extensive parameter tweaking. We therefore assume physically plausible input: for example light sources are modeled as geometry that is part of the scene, glass objects always have a thickness, and surface shaders need to be energy conserving. Progressiveness enables the user to continue image computation until a satisfactory result is achieved. There is neither the need to restart computations nor to discard intermediate results that allow for previews of an illumination situation after a short amount of time.

The goal of our system is to provide a fast and smoother global illumination solution as compared to approaches using multiple importance sampling [Veac 95, Veac 97a, Veac 97b] without sacrificing too much quality and flexibility. We achieve this goal by combining three techniques, each of which simulates a disjoint subset of path space in such a way that the techniques benefit from each other. For ease of reading, we apply Heckbert's notation (see Section 6.1) to identify path subspaces: E is the eye or camera, D denotes a diffuse bounce (in Section 2.3.4 we defined what diffuse means for our rendering system), S is a non-diffuse (specular) bounce, and L represents the light sources. Figure 6.14 illustrates the light transport paths we are able to simulate. The techniques and the simulated light paths in our system are:

(A) EDL **and** $EDD(S|D)^*L$ are handled by the point-light-based algorithm (covered in Section 6.3.2). These paths are directly visible diffuse surfaces that are directly or indirectly illuminated by another diffuse surface.

(B) $EDS(S|D)^*L$ is created by the caustic histogram method (covered in Section 6.3.3). This is the classic caustic situation where a diffuse surface is illuminated indirectly from a specular surface.

(C) $ES^+D^+(S|D)^*L$ is handled by specular gathering (covered in Section 6.3.1) and uses the results of the previous two methods. These are light paths of directly or indirectly lit surfaces seen through a specular reflection or refraction.

(D) ES^*L is directly evaluated as in the path tracing algorithm that was described in Section 6.1.1.

Looking at the path notation it can be seen that all these contributions are disjoint and two methods can never generate a contribution for the same path. All four techniques

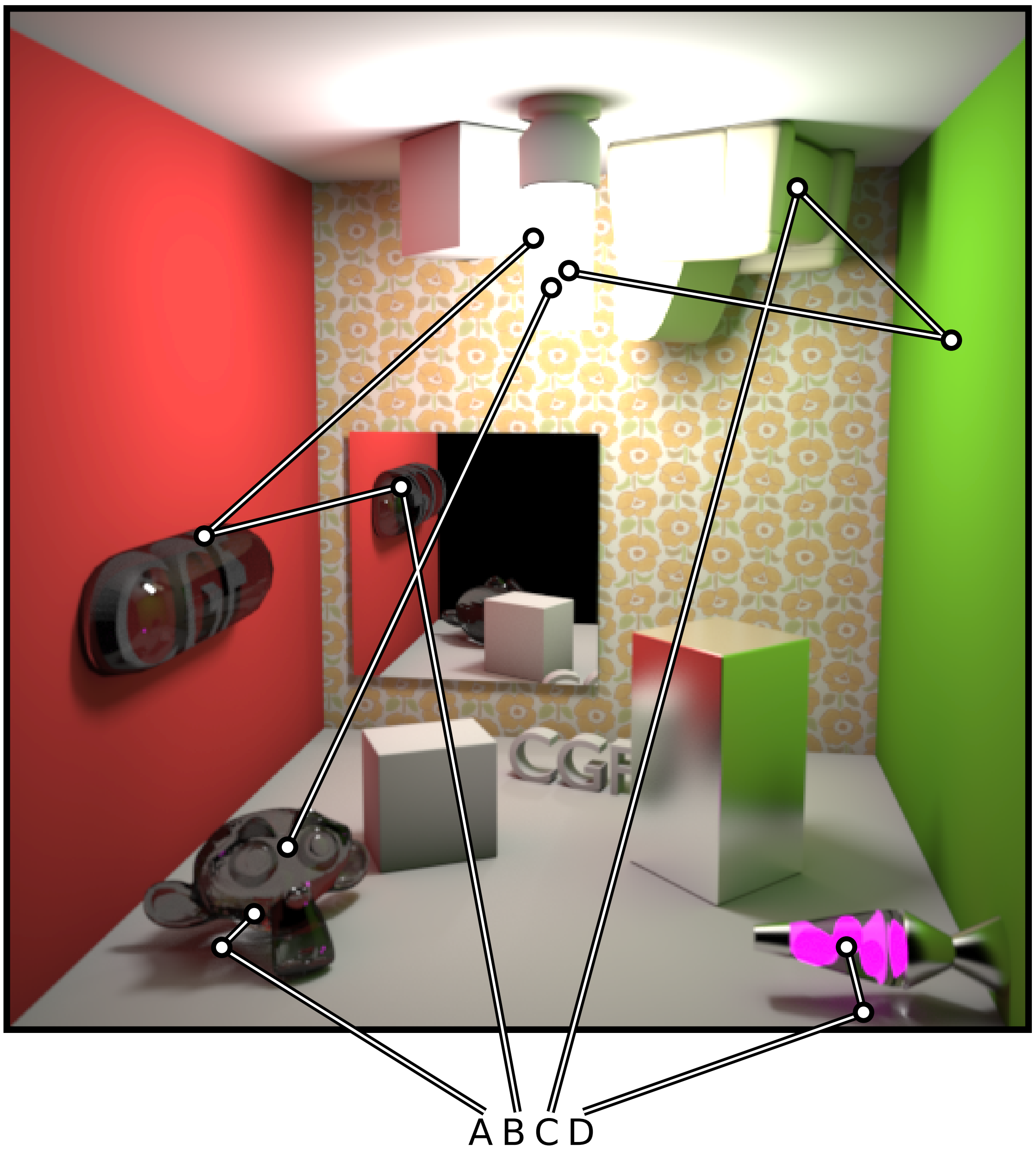


Figure 6.14: Illustration of selected light paths that we are able to simulate robustly: ’A’ is a classic caustic path ($EDSSL$), ’B’ shows the mirror image of a diffuse object behind a glass object ($ESSDSL$), ’C’ is diffuse indirect illumination from the green wall to the sofa ($EDDL$), and ’D’ shows a complex shaped light source (lava lamp) behind glass illuminating the scene ($EDSL$). Also note the caustic generated by the monkey head in the mirror ($ESDSSL$).

together can generate every possible path $E(D|S)^*L$. The final solution is simply obtained by summation.

Proof of completeness: To show the completeness of the above partitioning we start with with the complete set of possible paths

$$E(S|D)^*L \tag{6.2}$$

and factor out individual parts:

$$\begin{aligned} E(S|D)^*L &= ED(S|D)^*L + ES(S|D)^*L + EL \\ &= EDL + ED(S|D)^+L + ES(S|D)^*L + EL \end{aligned} \tag{6.3}$$

Taking the second term of Equation 6.3.

$$ED(S|D)^+L = EDD(S|D)^*L + EDS(S|D)^*L \tag{6.4}$$

This gives us already the first two techniques (A) and (B). What remains is the third and fourth term of Equation 6.3

$$\begin{aligned} ES(S|D)^*L + EL &= ES^+(S|D)^*L + EL \\ &= ES^+D(S|D)^*L + ES^*L \end{aligned} \tag{6.5}$$

which gives us (C) and (D).

Figure 6.15 shows the steps our rendering system performs in each rendering pass. A rendering pass is one iteration of computation that is finally added to the accumulation buffer containing the progressively refined result image. In the following sections each of the steps is explained in detail.

6.3.1 Eye Path Generation and Glossy Effects

The first step of our rendering system is the generation of eye path vertices. This is illustrated in Figure 6.16 (a). The eye paths are created by a random walk starting from the camera and are terminated via Russian roulette or when they hit a diffuse surface. The vertices are stored only on diffuse surfaces. These points are similar to the gather points in the Lightcuts [Walt 06] algorithm (but we store only the end points of a path) or the hit-points stored in the progressive photon map algorithm [Hach 08b]. The main difference is that we create a new set of eye path vertices each rendering pass instead of keeping them for the whole image generation process. This allows for progressive refinement of spatial anti-aliasing and complex specular effects. In the standard setting one eye path is started per pixel for each rendering pass. When such a path hits a light source (either directly or via an arbitrary number of specular bounces: ES^*L) the contribution is directly added into the accumulation buffer but the path is not necessarily terminated as the surface of the light source might also reflect light.

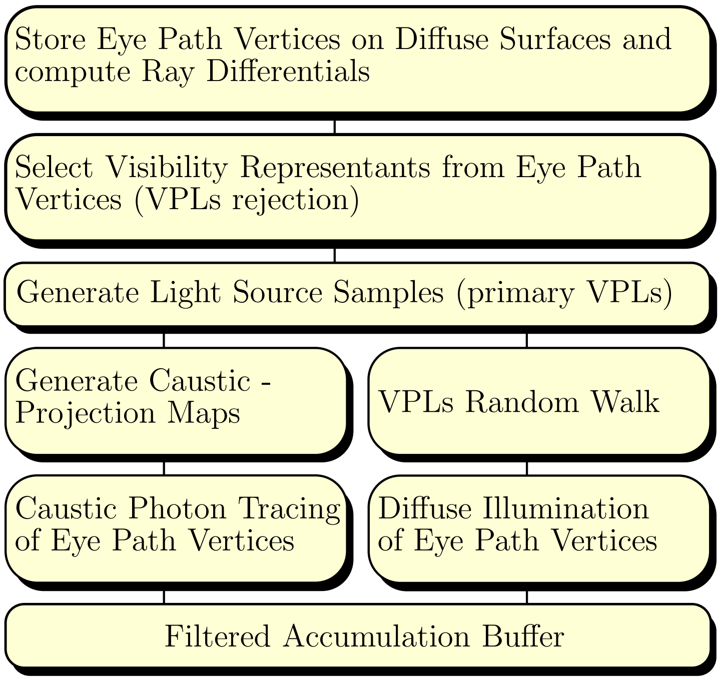

Store Eye Path Vertices on Diffuse Surfaces and compute Ray Differentials
Select Visibility Representants from Eye Path Vertices (VPLs rejection)
Generate Light Source Samples (primary VPLs)
Generate Caustic - Projection Maps
VPLs Random Walk
Caustic Photon Tracing of Eye Path Vertices
Diffuse Illumination of Eye Path Vertices
Filtered Accumulation Buffer

Figure 6.15: This figure describes the order of the basic operations our rendering system performs each rendering pass.

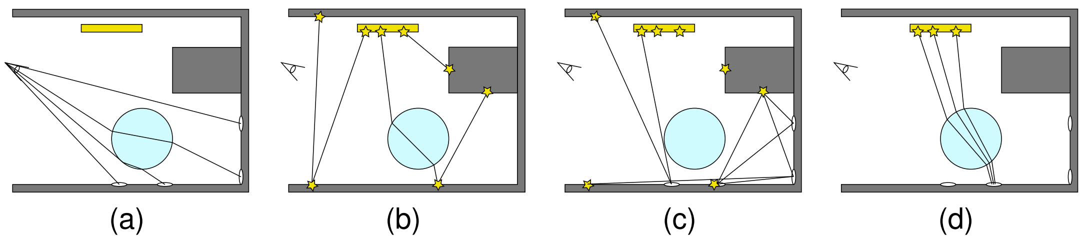


Figure 6.16: (a) Generation of the eye path vertices by performing a random walk into the scene starting from the eye. (b) Generation of the VPLs by a random walk starting from light sources. (c) Evaluation of the VPLs to compute an approximation of the diffuse illumination at the eye path vertices. (d) Accumulation of caustic photons into eye path vertices.

Using the generated eye path vertices we will gather diffuse illumination (Section 6.3.2), specular contributions, and caustics (Section 6.3.3).

Glossy and singular effects need to be simulated when the BRDF is not sufficiently diffuse (see Section 2.3.4). The idea of specular gathering is to sample the BRDF according to its probability density function by extending the eye path. This extension of the eye path is repeated until either it is terminated by Russian roulette or the path hits a diffuse surface. At this point the illumination is computed by using the point light sources and the caustic histogram (see Section 6.3.3). So, per rendering pass we gather the illumination for specular eye paths by using the current global illumination approximation given by the two other techniques. As this approximation is recomputed in each pass, the specular part becomes progressively refined, too.

6.3.2 Point-Light-Based Diffuse Illumination

We approximate the diffuse illumination by virtual point light sources (VPL) similar to the Instant Radiosity algorithm [Kell 97]. In order to be consistent with existing literature we use the term VPL (Virtual Point Light-source) for a light path vertex even though we prefer the term DIAL (Diffuse Illumination-Approximating Light) in the context of our rendering system. Note that the notion of diffuse illumination not only includes perfectly Lambertian surfaces, but also slightly glossy surfaces as specified in Section 2.3.4.

The first step in the Instant Radiosity algorithm generates samples on the light sources. As our scenes contain arbitrarily shaped light sources modeled as triangles, we first compute a PDF (probability density function) according to the area and intensity of each emitting triangle. This PDF is created once and can be reused for all subsequent rendering passes. Using three random numbers, a position on the light sources (stored as primary VPL) is sampled. Then a random walk terminated by Russian roulette is performed in the manner of a particle tracer, where on each diffuse surface another VPL is stored. Unlike eye paths, the light paths are not terminated on the first diffuse hit. This is illustrated in Figure 6.16 (b). These VPLs represent a point-wise approximation of the diffuse inter-reflection. In contrast to [Kell 97], the contribution of each VPL is then accumulated by sampling the direct visibility of the VPLs to each pixel via the stored eye path vertices, i.e. possibly incorporating even multiple specular bounces (see next section). Figure 6.16 (c) illustrates this. This accumulation is inherently progressive as each series of VPLs, generated by one random walk, is independent of the other random walks and the final image is independent of the number of VPLs created per rendering pass.

Stochastic Culling of VPLs. As we want to be able to efficiently handle complex scenes, we use an additional optimization to discard VPLs that are likely to have no contribution. For this purpose we choose a stratified subset of 256 of the eye path vertices and check the visibility from these locations prior to VPL storage. This can be either done based on an acceptance probability to not introduce an additional bias, or fully deterministic. To guarantee a good coverage and a changing subset for each rendering pass we use the Halton sequence over the image plane for the selection [Kell 00]. The culling is recreated each frame with a new set of samples to guarantee that even unlikely paths that might contribute significantly are not missed during the progressive rendering.

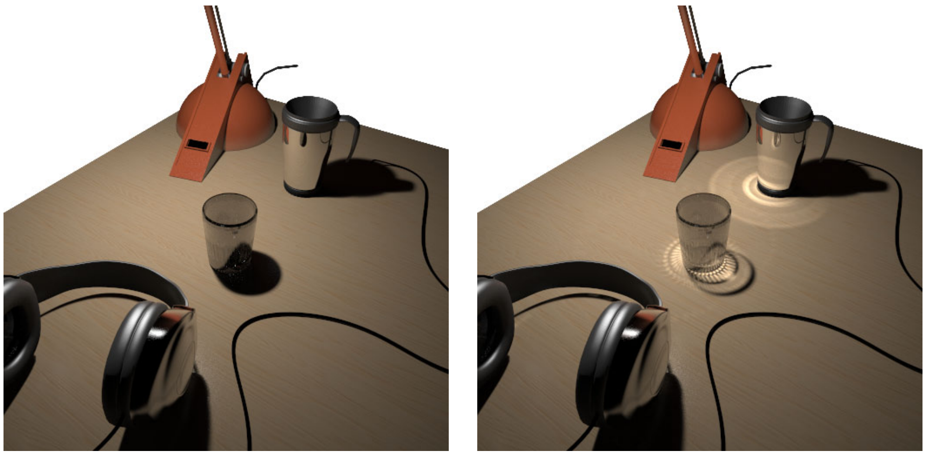


Figure 6.17: Illustration of the visual importance of caustic paths when reflecting or transmitting objects are in the scene. The invisible desk lamp is the only light in the scene.

6.3.3 Caustic Histogram Method

The final technique of our rendering system is the histogram method that provides the caustic paths (LS^+D). Figure 6.17 illustrates these paths in a scene where they are only enabled in the right hand image.

The technique can be understood from different perspectives. We describe it here from the viewpoint of histogram methods. The basic histogram method is an easy way of estimating the illumination in a scene and is related to finite element methods. In a pre-processing step, the scene is subdivided into surface patches that are called sample bins. In a second pass a light tracer generates random walks from the light sources and the number of photons hitting each bin is counted. This method can already be seen as a progressive method because each light paths improves the estimation. In its original form it is also a view independent solution, because the whole scene is subdivided into bins. While this method is very easy to implement, it suffers from the same problems as other finite element methods. First the quality of the solution depends on the granularity of the sample bins and artifacts are likely to be visible at the borders. Additionally, this method requires an enormous amount of memory when finely tessellated bins are used and, as other finite element methods, does not scale well to highly detailed scenes with millions of polygons. For our application of caustic paths we need high local detail to capture the fine details, however, only at very few places in the scene.

Our method is a two step method that is performed for each rendering pass. The two steps are illustrated in in Figure 6.16 (a) and Figure 6.16 (d). In the first step, the eye path vertices are created. Each eye path vertex has an associated bin size. To estimate the bin size we trace ray differentials [Igeh 99, Suyk 01] starting with half the pixel size using the same rationale as [Chri 03] used for geometry tessellation. This provides a good estimate of projected pixel footprint even after specular bounces.

This step produces the bins that are filled with photons in the second step. The bins are stored in a *k*d-tree the same way as the photons in a photon map (see Section 6.1.3). Now for each primary VPL (which are just samples on the light source) we trace photon random walks into the scene. If they do not hit a specular material they are discarded, otherwise they are continued until a diffuse material is found (or they are terminated by Russian roulette). The contribution to the image is computed by adding the value to the accumulation buffer using the associated pixel from each bin. Each photon is only accumulated into a single bin each bounce by choosing the closest. This way bins do never overlap and due to the euclidean distance calculation the bins are sphere shaped.

This procedure is illustrated in Figure 6.16 (d) for a glass sphere. In our implementation we trace photons until we have accumulated a given number for each rendering pass (in our case the parameter was set to 100000 for all images). Using the previously computed VPL as emitting light sources has the advantage of not requiring a new set of light source samples. The number 100000 provides a good balance between the caustics and the other parts of the illumination solution. Changing this number does not change the converged result, it can only change the convergence speed of caustics vs. the rest of the global illumination solution.

This method is conceptually similar to the algorithms described in [Dutr 93, Suyk 00, Hach 08b]. However, first, we only use the caustic histogram for the $EDS(S|D)^*L$ paths, and second, constant radius per eye path vertex is used for the bins based on the projected pixel size, which speeds up the photon collection. Furthermore, we recreate the bins in each rendering pass and thus in addition are able to support progressive per pixel anti-aliasing, glossy surfaces, depth of field, and motion blur. The fixed radius selection may of course introduce the usual photon mapping artifacts and introduces a bias, but as the (projected) bin size is not larger than half the pixel size the error is not larger than the error introduced by image space filters [Suyk 00]. The recreation has also been recently developed in [Hach 09].

Projection Maps

As explained above the caustic histogram gathers photons from reflected from specular surfaces. Hitting diffuse surfaces just absorbs the generated photons and it is wasted computation time. To avoid these paths we present an optimization based on projecting specular surfaces onto a hemisphere. Even for arbitrarily complex light sources only relatively few VPLs are generated per rendering pass. As we emit caustic photons from these VPLs to accumulate them in the histogram method we can make efficient use of a projection map per VPL as opposed to previous approaches of using projection maps with photon emission [Jens 09].

Projection maps are a pre-computation to direct caustic photons only in areas of glossy/specular objects. This increases the efficiency in simple scenes (with few glossy objects) significantly without slowing down complex scenes.

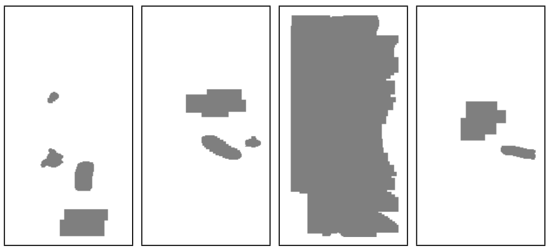


Figure 6.18: Illustration of four of the projection maps of the scene shown in Figure 6.27. The third map is from a VPL inside the lava lamp.

A discretized version of the hemisphere around each point light source is generated. A basic θ, ϕ discretization is used. In our experiments we used a resolution of 256×128 for each map with a single bit per entry. This projection map is generated conservatively. An entry of 0 means that in all directions contained in the discretization rectangle of (θ, ϕ) no specular object can be hit. An entry of 1 indicates a specular object in some of the covered directions.

The conservative projection map is generated by projecting each corner of the bounding box of a specular triangle onto the hemisphere. Triangles that lie completely behind the hemisphere are ignored. The result of this projection are eight pairs (θ_i, ϕ_i). From these angle pairs a new bounding box in spherical coordinates is computed. Care has to be taken when θ wraps around. Now each element of the projection map that touches the bounding box or is covered by it is set to 1. During construction the occupation of a projection map is computed by counting the number of elements that are set to 1. This number is then used to reject point light sources completely for caustic generation when they do not cover any specular objects. Figure 6.18 shows some projection maps created in the Box scene.

Such a projection map is generated for each point light source. A simple and practical way to use the projection map during light path generation is via rejection sampling. A random direction (θ_r, ϕ_r) on the hemisphere is generated using two random numbers ξ_1, ξ_2:

$$\begin{aligned} \theta_r &= \arccos\left(\sqrt{(1-\xi_1)}\frac{2}{\pi}\right) \\ \phi_r &= \xi_2 \end{aligned}$$

If the projection map contains 1 for the direction the light path is traced. Otherwise another pair of random numbers is generated. To assure correct energy for each generated photon it is important to count also the rejected photons.

During rendering the generation of projection maps was between 2% and 5% of the total time spent on the caustic histogram method. The speed-up achieved is linear to the coverage of the hemisphere, saving 85.2% of the photon rays in the Box scene, corresponding to 65.3% of the time required to compute the caustics and 4.6% of the total render time.

The utility and performance of this method depends of course on the scene setup. When all objects in the scene are specular and the scene is closed (i.e. from each point light source geometry is visible) this method does not provide any speedup. But practical experiments show that the expected slowdown is very small even for these cases. For the very few number of 40000 started photons the projection map generation of 8 projection maps is always below 2% of the total time needed to generated the caustic map. So this is a feature that can be always turned on and no user control is needed. Figure 6.19 shows the benefit of projection maps in a simple scene with three small caustic generators.

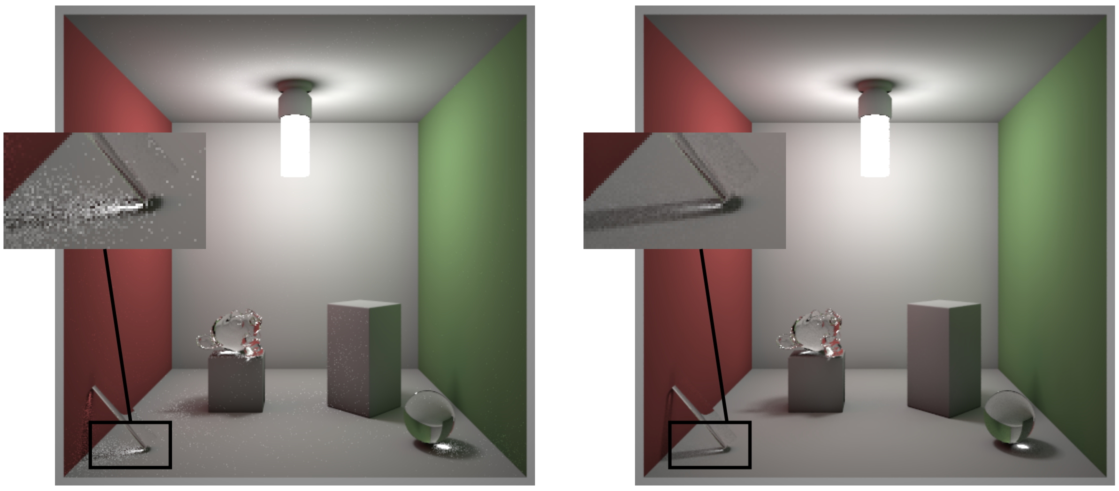


Figure 6.19: Advantage of using projection maps when there are only small caustic generators. Both images were computed with the equal computation time of 14 minutes on a single core. The right image is already much better converged in the caustic regions since much more photons actually contribute to the caustic histogram method when they are traced using a projection map. This scene also illustrates that our projection map implementation can easily handle arbitrary light sources.

6.3.4 Subsurface Scattering

We showed how a combination of three distinct techniques based on point light sources allows us to compute a global illumination solution including caustics, specular surfaces, depth of field, and dispersion effects in a single unified rendering system. Each of the techniques - VPLs, specular gathering, and the novel caustic histograms - efficiently works with the other two in combination. The progressive nature of our system allows the artist to start a rendering and be sure that it will converge to a unique solution independent of the parameter choice. At least in theory our system would produce the same image even if just a single VPL path or a single photon path would be generated per rendering pass. Additionally, our system uses only a constant amount of memory during rendering independent of the final image quality.

Subsurface scattering can be integrated in our system by allowing eye path vertices inside solid objects and tracing the photons also through these objects. An example of a simple integration can be found in figure 6.20 where the eye path vertices were created below the surface of a candle object. Additionally, participating media using the

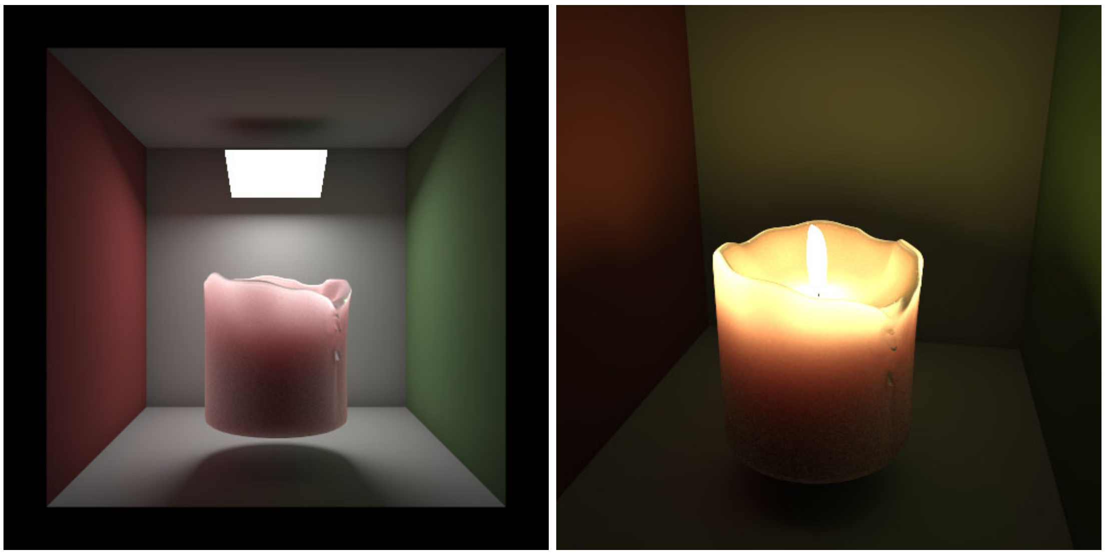


Figure 6.20: A simple subsurface scattering implementation using the caustic histogram method described in 6.3.3. An isotropic scattering material was used.

VPLs can be integrated [Raab 06]. Adding motion blur requires to sample just another dimension in our progressive renderer [Chri 06].

6.3.5 Results

In this section we discuss the global results of our rendering system. All images in this section were computed using the same default parameter settings of 16 VPL paths, 1 eye path per pixel, and 100000 caustic paths per rendering pass. All measurements were done on a Intel®Core$^{TM}2$ Quad Q9550 @ 2.83GHz.

Figure 6.26 shows the statistics for the different scenes used to present different aspects of our system. Figure 6.27 shows the image generation process with the inverse difference image scaled by 8 compared to a converged image with 11456 render passes. In Figure 6.21 you can find the associated graph showing the mean square error.

Diffuse Illumination. The diffuse illumination (direct and indirect) is approximated by the use of VPLs and thus very robust. This method was already discussed in many

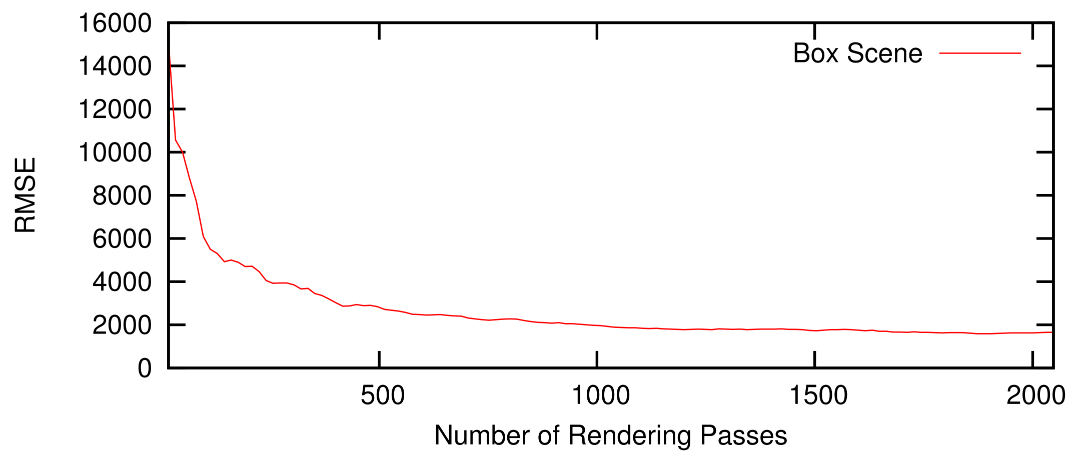


Figure 6.21: This graph shows the errors of the images (computed at 1024×1024) shown in Figure 6.27 from 16 to 2048 rendering passes in steps of 16 compared to a converged image with 11456 rendering passes. The steep descent at the beginning is because the indirect illumination is quickly approximated by the point light sources.

publications and is known to produce good results with a sufficient number of point light sources. As our system is progressive, the only limiting factor for the number of point light sources is rendering time. Figure 6.22 shows an equal time comparison of a bidirectional path tracer (BDPT) and point light based global illumination. The difference is most noticeable in the darker areas, where the detail is still obstructed by noise in the BDPT, while the VPLs approximation already shows each feature clearly. This robustness of indirect illumination in darker areas can also be seen in Figure 6.12 where the far left and right sides are only illuminated by indirect light. Note that the use of point lights for diffuse illumination also allows for efficient handling of diffuse transparent objects like curtains by creating VPLs also on the exit points (two sided) during the random walk.

Glossy Effects. Figure 6.23 shows a simple single layered material where κ is varied from 0 to 0.9. The first three spheres in the last row have a $\kappa \in \{0, 0.1, 0.199\}$ and are thus illuminated by VPLs directly (the specular threshold is $\kappa \geq 0.2$). The remaining spheres are illuminated by specular gathering, i.e. by extending the eye paths. Note, how these two techniques generate a consistent transition.

The top right image in Figure 6.26 shows a scene with many glossy objects and high variance situations where glossy objects are reflected in other glossy objects. In addition, the scene features depth of field. The illumination comes from a sky model only through the two small windows.

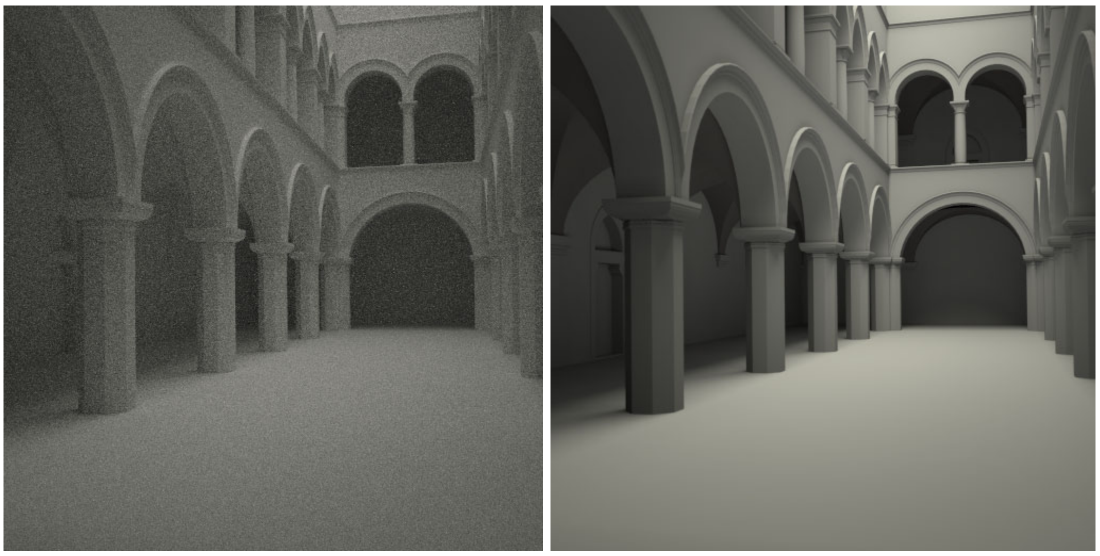


Figure 6.22: Equal time comparison (512×512, 3 minutes) of a BDPT (left) and the point light based renderer (right) in a simple diffuse scene. The noise in the BDPT solution is especially noticeable in the dark areas where none of the details are yet visible while in the other image the door and ceiling is clearly visible on the left side.

Caustics and other SDS paths. Caustics and SDS paths are efficiently treated by the caustic histogram method using projection maps (Figure 6.12 and Figure 6.14). The SDS paths are especially difficult in BDPT-based renderers and take a long time to converge even with Metropolis sampling. Without them the ground below the feet of the glass dinosaur in Figure 6.12 would be black. Section 6.3.3 further demonstrates how spectral effects can be integrated in our rendering system. The effect of spectral rendering is most noticeable in caustic areas, however, the presented method integrates well with the other proposed parts of the rendering system.

Depth of Field. Since we create a new set of eye path vertices in each rendering pass we can easily simulate e.g. a thin lens camera that renders depth of field effects. As our system is progressive we do not need to adjust any of the parameters for the illumination computation and, in many cases, the number of rendering passes needed for the illumination to converge is sufficient to robustly estimate depth of field effects (Figure 6.26).

Comparison to Progressive Photon Mapping and Metropolis Light Transport. Figures 6.24 and 6.25 show a direct equal-time comparison of our proposed rendering method to progressive photon mapping and to Metropolis light transport. For the first image we used a render time of 8 minutes. For the second image, 15 minutes were

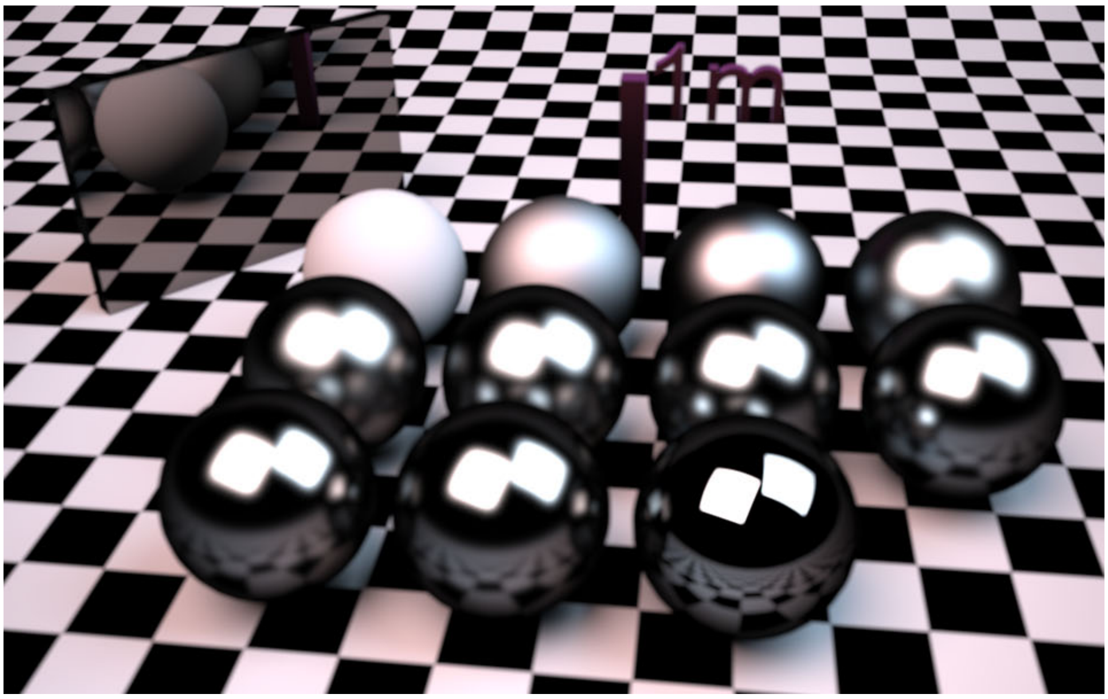


Figure 6.23: This image shows the smoothness of changing from VPLs based illumination for diffuse illumination ($\kappa < 2$, Section 6.3.2) to specular surfaces ($\kappa \geq 2$, Section 6.3.1). Due to the progressive rendering both methods converge to a final solution where no difference is noticeable without the need to set the correct number of samples or number of point light sources.

used. All images were created at a resolution of 512×512. Figure 6.24 is a simple box setting with diffuse colored walls. As it can be seen in the images our method produces much smoother results while still retaining a high quality caustic. Figure 6.25 shows more complex light paths through a glass object. The object is placed on a large diffuse plane. Here as well, our method produces much smoother results without sacrificing image detail. The progressive photon mapping algorithm still exhibits some low-frequency noise because it cannot gather enough photons within the given render time. In contrast, the Metropolis algorithm still has considerable high-frequency noise. Note that also progressive per pixel anti-aliasing is not possible in the progressive photon map algorithm since the gather-points are fixed for the whole image generation process. For the Metropolis comparison we used the open source luxrender v0.6rc5 as it contains a well optimized implementation. This explains also the slight difference in image appearance as this rendering system handles light sources and especially materials a little differently from our system. The last row shows the difference image to the path traced solution. It can be observed that each method has unique areas where the difference is still quite high after the 15 minutes of computation time. The progressive

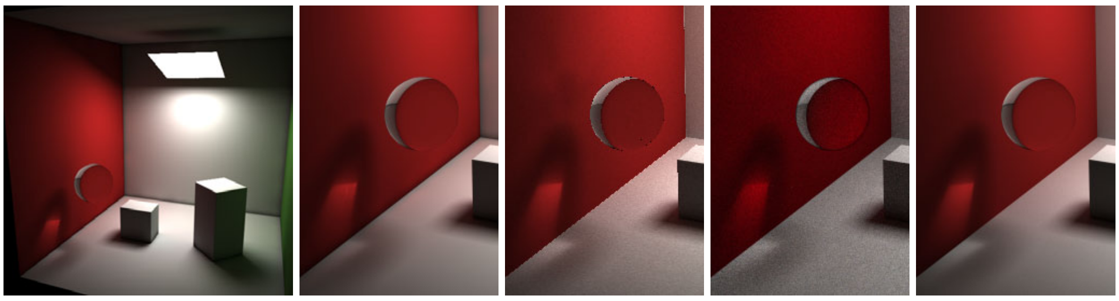


Figure 6.24: Equal time comparison (512×512, 8 minutes) between our proposed algorithm (first), progressive photon-mapping (second), and Metropolis light transport (third). The fourth image column shows a path traced image with 300000 samples per pixel.

photon map shows especially at the geometry silhouettes the lack of anti-aliasing but also quite high differences in the area, where few photons are transmitted and thus the gathering radius is still large. This is the same area where the Metropolis also has some higher noise. In contrast to this our algorithm spreads the error more evenly and shows an almost noise free error image.

6.3.6 Discussion

Indirect caustics ($EDS^{+}D^{+}L$) are created from non-primary VPLs. This works well when diffuse surfaces are brightly lit, but in many scenes this increases the variance too drastically for the final visible effect. So we chose to make them artist-controllable instead of enabling them by default. Furthermore, the depth of indirect caustics can be selected prior to rendering. Generally, a desirable feature for the whole rendering system would be to automatically balance the three parameters (number of VPLs, number of specular gathering rays, and the number of photons for the histogram) for optimal convergence speed with respect to a given scene.

The bias of bounding the geometric term in the Instant Radiosity method can be removed by the method proposed in [Koll 04]. The clamped contribution of point lights is compensated by gathering the missed illumination through additional rays. This comes at the cost of increasing the variance, especially in corners for an effect that is only seen in few scenes. We again made this an artist-controllable option.

Using a material model that cannot be split easily into a diffuse and a specular part (for example when using measured BRDFs like in [Matu 03]) might increase the variance and thus the time to a high quality image. One could use a simple heuristic to classify a BRDF as more diffuse or more specular but a wrong choice results in longer computation times. Note that this does not change the result image, only the time required to compute it. While our proposed rendering system is progressive by design, it is not yet adaptive in any way (except that in highly glossy scenes automatically fewer VPLs are stored). Especially in diffuse areas the resulting image quickly converges to

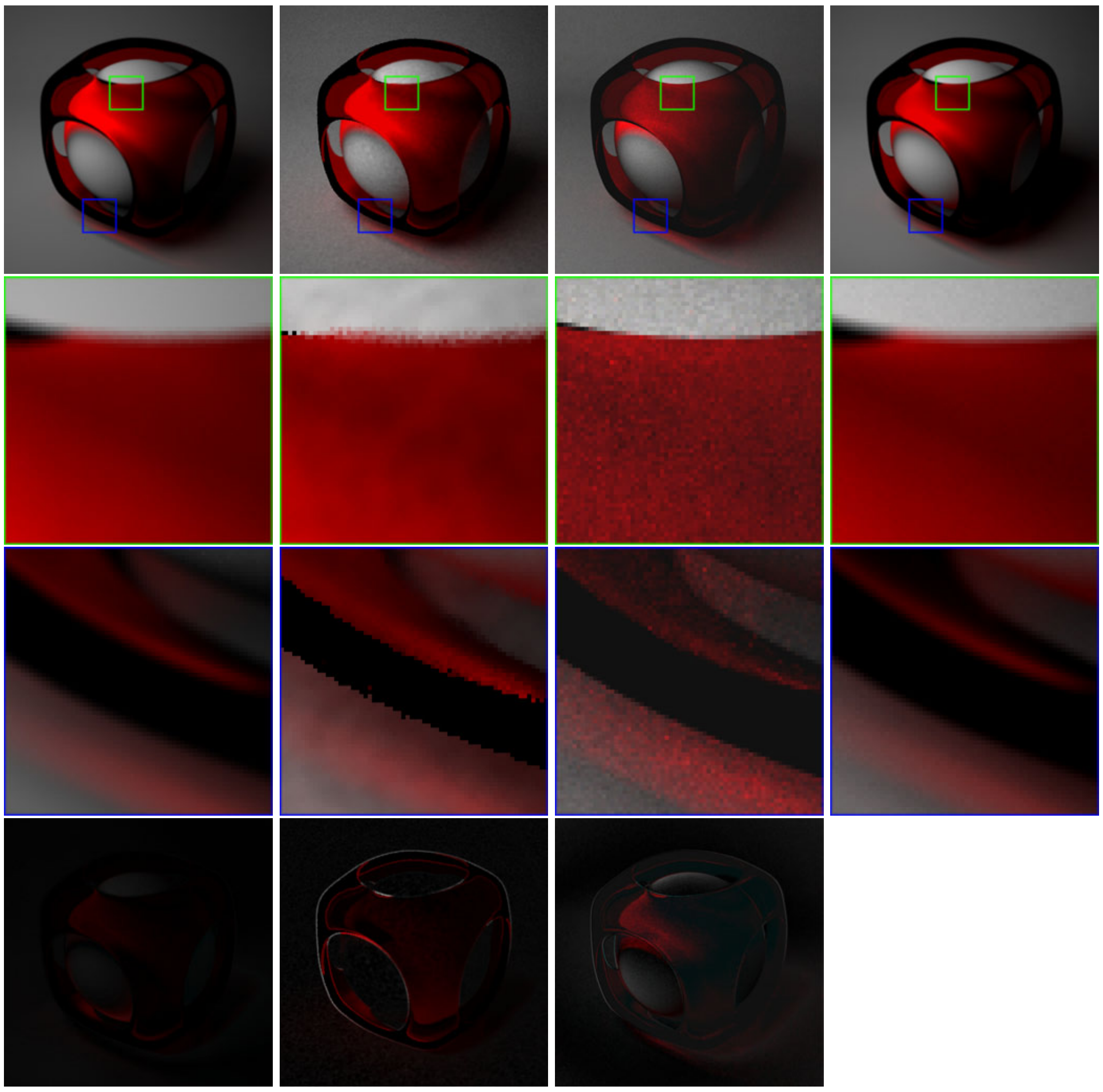
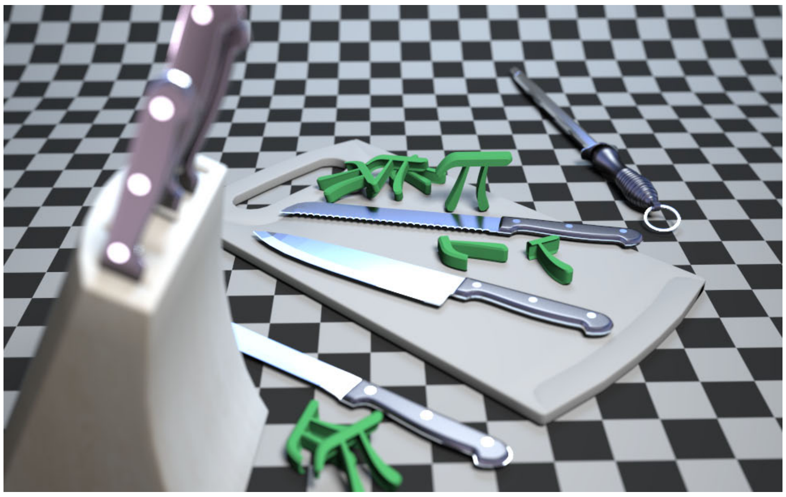
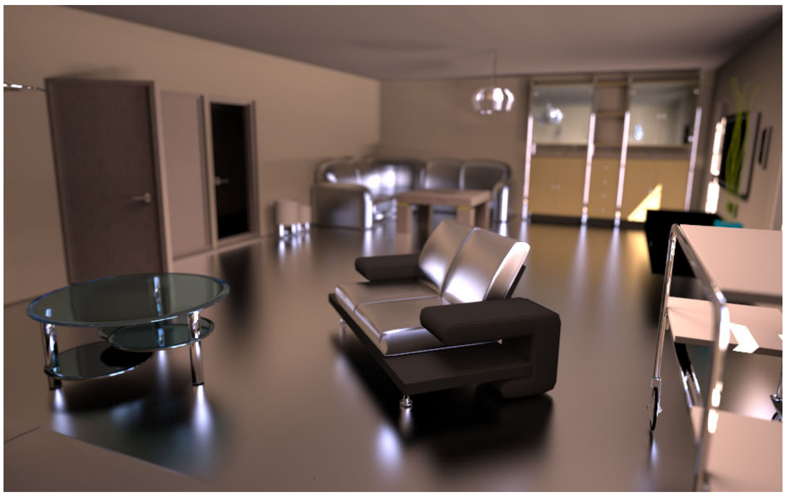


Figure 6.25: Equal time comparison (512×512, 15 minutes) between our proposed algorithm (first column), progressive photon-mapping (second column), and Metropolis light transport (third column). The fourth image shows a path traced image with 300000 samples per pixel. The last row shows the difference to the path traced image.

an acceptable solution while in areas of SDS paths there is still noticeable variance. By intuition one would like to sample only in areas of high variance. But this would remove some of the advantage of a real progressive renderer as it cannot be guaranteed that the under-sampled region is no area of higher variance (for example light through a key hole) that was not yet sampled. Of course knowledge about a given scene allows an artist to increase convergence speed by tuning the number of point light sources or caustic photons per rendering pass accordingly.



Knives (960×600 1.7 hours): thin lens depth of field simulation



Interior (960×600 2.8 hours): many glossy materials and depth of field

Scene	Res.	Polygons	Avg. VPL	% Diffuse	% Eye	% Caustic	per Pass
Box	1024×1024	49344	35.7	76.5%	5.7%	13.4%	4.8s
Museum	1200×520	331843	7.9	35.6%	21.7%	$37.6.8\%$	1.2s
Interior	960×600	909441	3.8	43.4%	35.3%	9.3%	0.9s
Knives	960×600	95472	22.1	52.0%	6.2%	39.2%	3.1s

Figure 6.26: The statistics show the relative time spent on the three techniques. The remaining time to 100% is due to management overhead. The projection map generation was between 2% and 5% of the total caustic time. $\%$ Eye is the relative time spent generating the eye paths. The last column shows the average time per rendering pass (resolution dependent). It can be seen that in scenes with many glossy or singular materials the number of average VPL per pass drops while the time spent for the eye paths automatically increases.

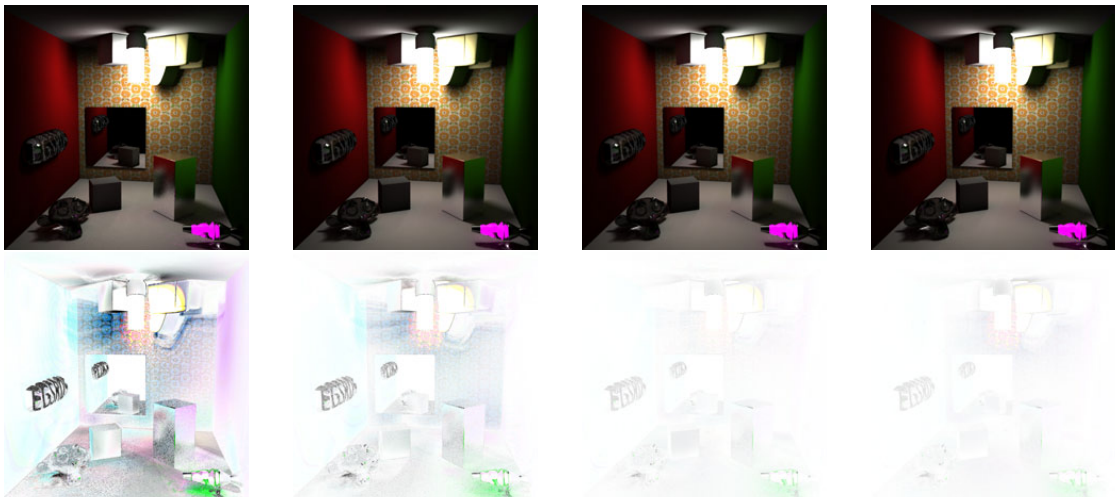


Figure 6.27: Evolution of the error with increasing number of rendering passes compared to a converged image with 11456 rendering passes (resolution: 1024×1024). The top row shows the results after $16, 128, 512$, and 2048 rendering passes. The bottom row shows the inverse difference image scaled by 8. Figure 6.21 shows the graph of the errors for the full image series.

6.4 High Dynamic Range Environment Map Lighting

In this section we present a novel method to start light paths on an emitting environment map in the context of Monte Carlo global illumination with image-based lighting or physical sky models. While there have been previous methods to use environment maps in a physically based renderer, none of these methods could start correctly weighted light paths. We solve this problem by generating projected planes that have a known area and can be sampled easily. These planes basically represent parallel light sources that have the correct lighting energy taken from the environment map. Parts of this technique are published in [Damm 09a]. This method was developed to efficiently generate the VPLs needed for the rendering system described in Section 6.3. But with this technique we can also efficiently render unbiased caustics from environment maps using bi-directional path tracing with multiple importance sampling, photon mapping, and the classic instant radiosity algorithm. When simulating light transport using Monte Carlo methods, all kinds of paths connecting light sources and the sensor must be generated. Even though all paths will eventually be found by a simple path tracer (see Section 6.1.1), some effects are best captured with some specialized sampling technique. A light tracer for example, starting the random walk at the light sources and deterministically connecting to the sensor, is well suited to render caustics. Other global illumination methods, such as photon mapping (see Section 6.1.3), bi-directional path tracing [Veac 94], or point light source-based methods as instant radiosity (see Section 6.1.4) rely on paths started from the light sources. While starting paths on geometric light sources is common practice (see Section 2.3.5), it is not as easy to start a path from an emitting environment map with a correct sampling density.

6.4.1 Domains of Integration

Analytic sky models like [Pree 99] or environment maps in the form of captured images [Debe 98] approximate distant illumination by an infinitely large sphere around the scene. Thus the radiance $L(\omega)$ emitted from the environment only depends on the direction ω for any point in the scene.

In a classic path tracing setting, the direct illumination is evaluated by sampling a direction at a point x and evaluating the visibility of the environment map (see Figure 6.29, left). In a Monte Carlo renderer this direction is picked according to a carefully chosen probability density function in order to minimize variance.

In a bi-directional algorithm this is not as straight forward as we need to start a path from outside the scene with a correct and known probability density function. For example in an instant radiosity-based renderer the indirect illumination is approximated by distributing point lights via a random walk from the light sources. When using environment maps the direct illumination can be evaluated as described above. For indirect illumination, sample points to start the light paths have to be generated with a correct probability distribution. This is illustrated on the right hand side of Figure 6.29 where the desired sampling point is marked as y. The problem is similar to the sampling of parallel light sources with a correct sampling density that was described in [Phar 04].

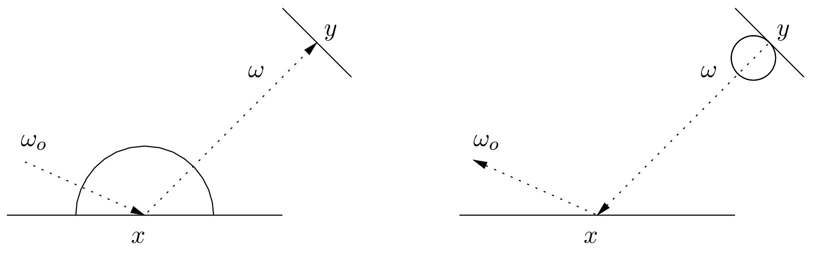


Figure 6.28: Sampling densities for area light sources: path tracing as in Equation 6.6 (left), and light tracing as in Equation 6.7 (right).

Monte Carlo methods for light transport estimate integrals using the expected value of some constructed random variable, e.g. to evaluate how much light emitted from a surface patch A_y is received by a surface patch A_x (see for example [Dutr 06]):

$$\begin{aligned}
F &= \int_{A_x}\int_{A_y} L(y \leftrightarrow x)\underbrace{\frac{\cos_x \cos_y}{||x-y||^2}}_{=:G} V(x \leftrightarrow y)dydx \\
&= \int_{A_x}\int_{S^2_-} L(x \leftarrow \omega)\cos_x d\omega dx \qquad (6.6)\\
&= \int_{A_y}\int_{S^2} L(y \rightarrow \omega)V(x \leftrightarrow y)\cos_y d\omega dy, \qquad (6.7)
\end{aligned}$$

which can be evaluated by corresponding Monte Carlo estimators, each sampling a combination of x, y, and ω for the respective integrals. The first line would sample two points and directly evaluate the geometric term G. Equation 6.6 describes a backward path tracer, sampling an outgoing direction from x (see Figure 6.28). Equation 6.7 describes a light tracer, starting with a point y on the light source. This last variant is especially useful to render caustics and the starting point for many algorithms (e.g. photon mapping). The corresponding estimator using Equation 6.7 is

$$F = E\left(\frac{1}{N}\sum_{k=1}^{N}\frac{L(y_k \rightarrow \omega_k)V\cos_{y_k}}{p(y_k)p(\omega_k)}\right) = E\left(\frac{1}{N}\sum_{k=1}^{N}\pi|A_y|L(y_k \rightarrow \omega_k)V\right)$$

with $p(y_k) = 1/|A_y|$ a uniform distribution on the surface of the light source and $p(\omega_k) = \cos_{y_k}/\pi$ following a cosine distribution.

For image-based lighting [Debe 98], it is not as straightforward to find these probability densities. While $p(\omega)$ can easily be modeled (e.g. by using $p(\omega) = 1/(4\pi)$ uniformly on the sphere), the question remains where to put y.

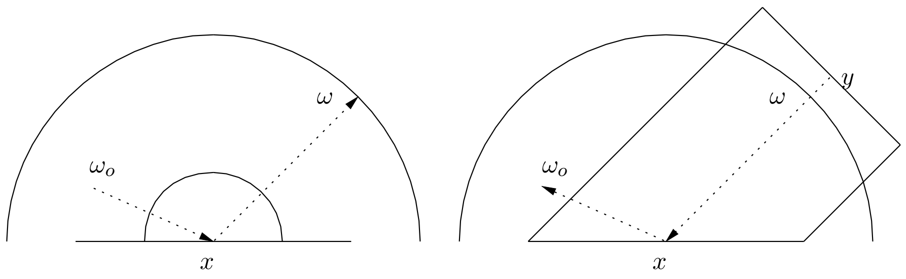


Figure 6.29: Two ways to sample distant illumination: on the left, ω is obtained by prolonging an eye path ending in x. On the right, a point y outside the scene is needed to trace a direction ω towards x.

6.4.2 Plane Sampling for Starting Light Paths from Environment Maps

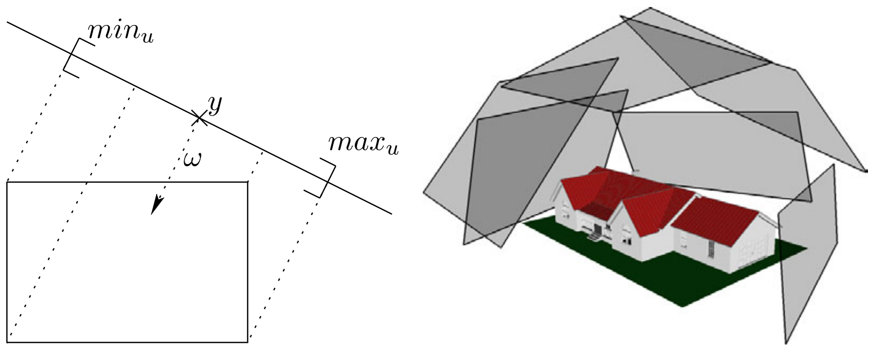


Figure 6.30: Illustration of the sampled plane in 2d (left), and visualization of six sampled planes for a 3d scene.

To allow for light tracing from an environment map, we start by reformulating a similar expression as in Section 6.4.1, in order to evaluate how much light emitted from an environment map is received by a surface patch A_x:

$$\begin{aligned} F_{env} &= \int_{A_x}\int_{S^2_-} L(x \leftarrow \omega)\cos_x d\omega dx & (6.8)\\ &= \int_{S^2_-}\int_{A_x} L(x \leftarrow \omega)\underbrace{\cos_x dx}_{=:dy} d\omega \\ &= \int_{S^2}\int_{A_y} L(y \rightarrow \omega)V(x \leftrightarrow y)dyd\omega & (6.9)\\ &= E\left(\frac{1}{N}\sum_{k=1}^{N}\frac{L(y_k \rightarrow \omega_k)V}{p(y_k)p(\omega_k)}\right). \end{aligned}$$

Equation 6.9 in fact describes a light tracing algorithm. The newly introduced variable $y = f(x) + C$ with $|J_f| = \cos_x$ can be found geometrically on a plane perpendicular to ω (see Figure 6.29). The vector C is used to shift the plane out of the bounding box, so the visibility term V will be equivalent to the one evaluated for the environment map.

This principle is easily extended from one surface patch A_x to the whole scene. The random variable y remains the same, even for different surface normals resulting in different $\cos_x$, as the substitution explicitly removes this term. The visibility term $V(x \leftrightarrow y)$ takes care of occlusion and even allows for a too large integration domain A_y, i.e. which covers more than just A_x after the transformation.

To start paths on the environment map, the densities for y and ω are separated, as for geometry light source sampling. Assuming for the moment that $p(\omega) = \frac{1}{4\pi}$ uniformly on the sphere will be sufficient. So at this stage we have a directional light source with direction ω and need to pick a starting point y. As such a light source has infinite extent or is infinitely far away, the first idea is to sample the end point x for the light ray. It is very hard for complex scenes to find the correct density for such an end point. We thus use virtual planes of finite extent just outside the bounding box of the scene to represent all of the directional light which is of interest for the scene (see Figure 6.28, right).

Environment maps can be interpreted as an infinite collection of parallel light sources (one for each direction). To start a particle path from the environment we divide the problem into first sampling a direction ω and then sampling an appropriate starting point y outside the scene for the directional light source associated with ω.

Algorithmically, we sample the direction ω on the environment map, according to a probability density $p(\omega)$, for example $p(\omega) = 1/(4\pi)$. Then we create an orthonormal basis (u, v, ω) around ω, and project the eight corners of the bounding box of the scene onto the plane through the origin spanned by (u, v), as illustrated in Figure 6.30. From these projections, we only keep a 2-dimensional bounding box in the coordinate system of the plane (`minu, maxu, minv, maxv`). This quad is pushed outside the bounding box of the scene. The exact distance does not matter, as the quad is used as a parallel light source. In practice, we project one vertex of the scene bounding box (or the center) onto $-\omega$ and shift this point in the direction $-\omega$ by the sum of the three bounding box widths. Pseudo-code for the plane generation can be found in Listing 6.1. Such a quad is easy to sample. The starting point y can be chosen uniformly on this quad, which results in the simple probability density $p(y) = 1/|A_y|$, with $|A_y|$ being the area of the quad. See Listing 6.2 for pseudo-code of this operation.

```
omega = envmap_sample();
inv_p = 1/p(omega);      // account for p(omega)
L     = envmap_radiance(omega);
(u,v) = get_onb(omega); // create orthonormal basis (u,v,omega)
for(int c=0;c<8;c++)    // project scene bounding box
{
  dotu = dotproduct(aabb.corner[c], u);
  dotv = dotproduct(aabb.corner[c], v);
  minu = min(minu, dotu); maxu = max(maxu, dotu);
  minv = min(minv, dotv); maxv = max(maxv, dotv);
}
float dist = dotproduct(aabb.center, -omega)
           + aabb.width[0] + aabb.width[1] + aabb.width[2];
```

Listing 6.1: Pseudo-code for the generation of the quad from a sampled environment direction. This computation can be reused for several samples.

```
y = - dist * omega + u*(minu + (maxu-minu)*rand1)
                   + v*(minv + (maxv-minv)*rand2);
inv_p *= (maxu-minu)*(maxv-minv); // compensate for p(y) = 1/A
particle.energy = L*inv_p;
particle.y = y;
particle.omega = omega;
```

Listing 6.2: Random sampling of the particle starting point on the precomputed quad

The estimator remains unbiased as long as the quad is large enough to cover all of the scene. In order to maximize the number of rays actually intersecting the scene, the quad should be as small as possible. Apart from enabling the use of algorithms relying on light paths (instant radiosity, photon maps, etc.), the advantage of the new technique becomes apparent when the environment map has a high dynamic range and importance sampling is used, i.e. $p(\omega) \sim L(\omega)$. We use sample warping [Clar 05] to sample ω according to the luminance of the environment map.

6.4.3 Results and Discussion

With the new technique it is now possible to create caustic paths from the environment efficiently. Other unbiased techniques generate these paths as well, but at a much lower convergence rate. For an illustration of this significant difference, see Figure 6.31. The environment map contains a sun with pixels 1000 times brighter as in the rest of the image. All three path samplers use the same importance sampling [Clar 05] to evaluate direct light, thus the same sharp shadow appears in all images. However, using this kind of importance sampling for direct light only fails to capture caustics. Metropolis sampling helps, but will waste a lot of samples in the sun, as the probability is proportional to the radiance contributed to the image. Of course Metropolis sampling will perform better

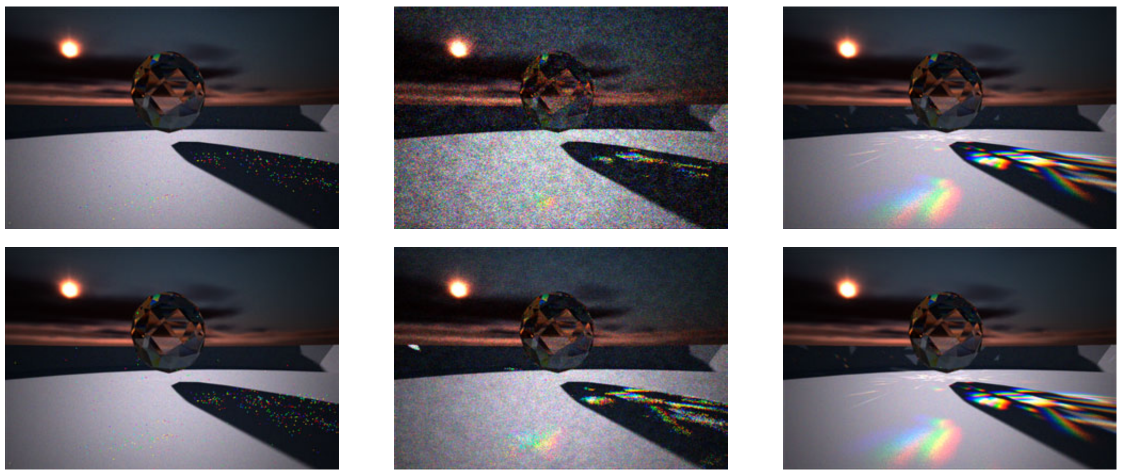


Figure 6.31: Comparison of plain backward path tracing from the eye (left), the same path tracer with Metropolis sampling (middle), and bi-directional path tracing with a light tracing pass using plane sampling (right). Each example on the top row has equal computation time (about 5 minutes on an AMD Opteron). The path tracing achieves 342 samples in that time, the Metropolis version can only evaluate 105, the bi-directional method has 231 samples per pixel. The bottom row shows a comparison of equal sample count (500 samples per pixel).

when a bi-directional path sampler is used and when the sun is not directly visible, especially in difficult areas of the image such as caustics from multiple inner reflections seen through a mirror.

The plane sampling technique is a specialized solution to efficiently capture prominent effects, such as caustics, when image-based lighting is used. It is also most useful when bi-directional algorithms are used, which rely on paths started from the light sources, such as instant radiosity or photon mapping. A bi-directional path tracer could work around this by employing path tracing for these effects, but efficiency will be much lower if the environment is non-diffuse.

However, there are cases in which other techniques are better. For example a mostly diffuse environment such as a cloudy sky is best sampled by a path tracer, as importance sampling by surface reflection can be done. If only direct lighting is of interest, combined sampling of incoming radiance and surface reflection [Clar 08] is the way to go.

The simple method described above to sample a quad, which is the 2d-bounding box of a projected 3d-bounding box, can easily be extended to choose (u, v) to minimize the size of the quad, or be replaced by sampling for example a disc (the projected bounding sphere) as was done in [Phar 04]. We chose quads because most render systems have the axis-aligned bounding box of the scene readily available, and it is then straightforward to extend the technique to projection maps (see Section 6.3.3) to be used with photon mapping or instant radiosity. Since the parametrization of each plane is implicitly given only a discretization resolution has to be chosen and the rest of the projection map algorithm is the same.

6.5 Interactive Preview for Design and Editing

This section presents a fast and simple screen space filtering method designed for ray traced Monte Carlo global illumination images which achieves real-time rates. Parts of this are published in [Damm 10b]. The rendering system described in Section 6.3 was designed for offline high quality rendering and generated noise free images by separating the light contributions. For interactive or preview applications it is only partly suitable since the error is not distributed uniformly over the screen when instant radiosity is used. For preview applications a simple path tracer is better suited, but even on modern hardware only few samples can be traced for interactive applications, resulting in very noisy outputs. Taking advantage of the fact that Monte Carlo computes hemispherical integrals that may be very similar for neighboring pixels, we derive a fast edge-avoiding filtering method in screen space using the À-Trous wavelet transform that operates on the full, noisy image and produces a result that is close to a solution with many more samples per pixel. In contrast to other real-time preview systems like lightspeed [Raga 07], we do not store any precomputed data and use the same algorithm for preview as is used for final-frame rendering. This avoids the complexity of relighting approaches, where different rendering algorithms are used for preview and for the final rendering as was for example done in [Pell 05].

As discussed in Section 6.1, Monte Carlo-based global illumination ray tracing is one of the most flexible and robust light transport simulation algorithms. Although there exist many different approaches, they all use random path sampling to solve the integral in the rendering equation. Usually hundreds of paths per pixel are necessary to achieve a smooth image. But even modern computers and algorithms can only provide a few paths per pixel in the time-frame needed for interactive applications or a fast preview. Limiting the number of randomly sampled paths results in an image that still contains highly visible noise. Even in conceptually simple illumination situations like a diffuse uni-colored wall, the noise is visible. But in this case the correct solution varies smoothly from pixel to pixel and the result can be improved by combining several neighboring samples into one integral estimate. Edge-avoiding wavelets provide an efficient approach for filtering such samples, provided that a good edge-stopping functions is available. However, as we show in Section 6.5.3, the standard decimated wavelet bases are poorly suited for the Monte Carlo noise and show visible artifacts. On the other hand the bilateral filter avoids these artifacts but can be quite slow for large filter sizes.

Therefore we extend the fast, un-decimated À-Trous wavelet transform to incorporate multiple edge-stopping functions and show that it can robustly filter highly noisy Monte Carlo global illumination images and produce visually pleasing results at interactive rates. We use a hybrid technique that supplements the output of a CPU based ray tracer with information from additionally rasterized images. Our GPU shader applies the Edge-Avoiding À-Trous filter combining the different buffers to produce the output image.

Edge-Avoiding Wavelets and the Bilateral Filter

Naive computation of the discrete wavelet-transform requires convolutions with increasing filter sizes per level. The *Mallat algorithm* [Mall 89] alleviates this problem by introducing *decimation* (i.e. down-sampling) and illustrates the relationship to multiresolution analysis. Efficient computation of the un-decimated transform has been proposed by [Hols 89, Mall 98] adapting *Fast Filter Transforms* [Burt 81] for the use with wavelets. These filters contain a constant number of coefficients on each level of the analysis, by spreading the initial filter by a factor of 2^{level} and filling in zeroes. In the wavelet context this is known as the *algorithme À-Trous* ("with holes"). A formal analysis of the relationship between the Mallat algorithm and the À-Trous algorithm is given in [Hols 89].

The *bilateral filter* is a non-linear filter, proposed by [Toma 98] (among others), to smooth images. Since it is computationally expensive, methods for its acceleration have been developed (e.g. [Dura 02, Pari 09]). [Fatt 07] uses the À-Trous algorithm to compute a *fast multiscale bilateral decomposition*. Edge-avoiding wavelets [Fatt 09] adopt this for the decimated case by extending second-generation wavelets [Swel 97] and introducing an edge-crossing function. Edge-avoiding wavelets (computed via the Mallat-algorithm) are the decimated counterpart of the fast bilateral decomposition (computed via the À-Trous algorithm). In contrast to the many specialized and selective wavelet bases, the shape of each edge-avoiding wavelet itself changes data dependently similar to a data dependent windowing in the short time Fourier-transform.

The bilateral filter has been extended for different applications to use more information in the edge-stopping function. The *trilateral filter* [Chou 05] uses an adaptive neighborhood function and the image gradient. The *joint* and *cross bilateral filter* [Pets 04, Eise 04] use the color information from multiple images taken under different lighting conditions to combine their frequency components. [Sloa 07] uses bilateral up-sampling from a low resolution low frequency light buffer for *real-time soft global illumination*. But a low resolution buffer lacks the positional precision necessary to decide on which side of an edge a sample lies thus removing the possibility of retaining high resolution details like sharp shadows. For this reason our *edge-avoiding À-Trous* filter operates on a full resolution buffer. It combines edge-stopping functions from multiple input images (noisy ray tracing image, normal buffer, position buffer) and is efficiently computed on the GPU.

Fast and Smooth Global Illumination

Real time display of smooth global illumination solutions can be achieved using precomputation. The most prominent algorithms are methods based on Precomputed Radiance Transfer [Sloa 02] and Meshless Radiosity [Leht 08]. Based on interactive ray tracing on distributed systems and interleaved sampling in [Wald 02b] an approach for interactive global illumination was presented that uses the discontinuity buffer [Kell 98] to avoid filtering across edges.

Another approach builds on the Instant Radiosity algorithm (see Section 6.1.4) that approximates the global illumination with many point lights and extends it to interactive display [Dach 05, Dach 06, Lain 07, Nich 09]. [Rits 08] also approximate the visibility of the point light sources. These methods, like most GPU methods, have restrictions like the shape of the light sources and lack the generality of a full ray tracing solution. Finally the advancement in GPU computing power allows to implement previous offline algorithms like photon mapping [Jens 96a] or final gathering in realtime [Fabi 09, McGu 09, Rits 09, Wang 09].

There exist a vast mount of general (offline) techniques for Monte Carlo global illumination (see Section 6.1). Since Monte Carlo integration always comes with noise, several filtering techniques have been developed that exploit piecewise continuity in surface irradiance. Irradiance caching [Ward 88] for example is an early method that interpolated samples stored in a word space octree. This technique has been improved [Ward 92b] and was also adapted for use in movie production [Tabe 04]. A simple screen space filtering method is presented in [Kont 06] and recently the use of wavelets in the context of adaptive sampling has proved to be useful [Over 09].

6.5.1 Edge-Avoiding À-Trous Irradiance Filtering

Our method takes a noisy Monte Carlo path traced full resolution image and produces a smooth output image by taking into account the input image itself as well as a normal and a position buffer. Using the ray traced image as input allows to retain high frequency detail like sharp shadows that are not available in the other buffers. The resulting image shows sharp edges around geometry borders and successfully smoothes the illumination where needed. Using a general path tracing algorithm allows for various different light transport effects like glossy materials and different light source types and shapes (e.g. simple point lights, arbitrary (deformed) area lights, image based lighting), without special handling.

In the following we describe the stages of our algorithm and give the necessary theoretical background. Section 6.5.2 discusses our implementation with optimizations and in Section 6.5.3 we present the performance and discuss the results.

Figure 6.32 illustrates the basic steps of our algorithm. The input is the path-traced (noisy) image (rt buffer), containing in the simplest case both direct and indirect illumination. Additionally, the edge-stopping function of the À-Trous filter takes into account the normal and the position buffer. The number of applications of the À-Trous filter is determined by the total desired filter size (in our case 5 iterations).

In the absence of edges (e.g. all buffers uni-colored) it applies an even smoothing with increasing filter size per iteration. In the presence of edges the influence of neighboring samples is diminished by the edge-stopping function. This is closely related to the bilateral filter and was used with the À-Trous algorithm in [Fatt 07] and with decimated wavelet bases in [Fatt 09].

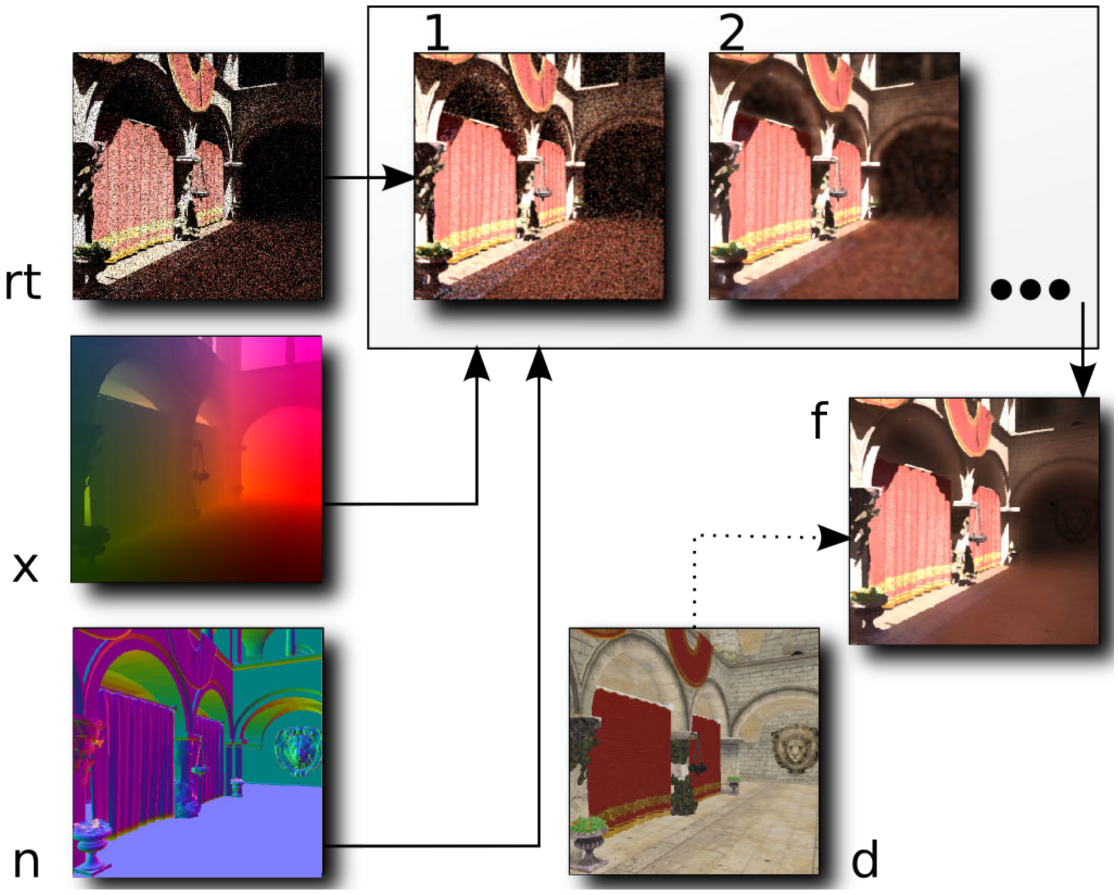


Figure 6.32: *rt), n) and x)* are the input buffers (ray tracing, normal, world space position). *1) and 2)* show two steps of the repeated application of our edge-avoiding À-Trous filter. *f)* is the final output image after tone mapping. *d)* shows the optional diffuse buffer

Monte Carlo Light Transport

The Monte Carlo Method for light transport solves the integral of the rendering equation (see Section 6.2.1) by constructing random paths that connect the camera with the light sources in the scene. This is the most general known solution for the full rendering equation and is well studied in literature. One problem with Monte Carlo image synthesis though, is the high variance that is visible even in simple scenes. To reduce this variance quadratically more samples are needed for a linear improvement.

To evaluate the À-Trous filtering algorithm we use a path tracer (PT) with next event estimation as our basic image formation algorithm (see Section 6.1.1). This path tracer starts by tracing rays from the camera into the scene as a random walk. At each interaction point a connection to a randomly chosen position on a light source is made. If it is not occluded the contribution of this path is added to the image accumulation buffer.

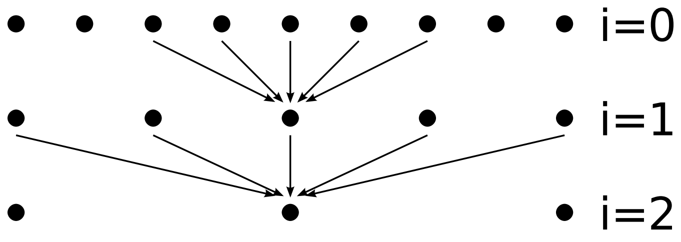


Figure 6.33: Illustration of the usual decimation step when using decimated wavelets. Each iteration the input signal is reduced. In contrast to this see Figure 6.34 where each iteration produces a signal with the same number of elements.

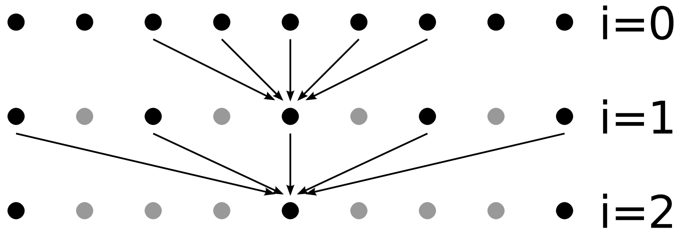


Figure 6.34: Three levels of the À-Trous wavelet transform. Arrows indicate pixels that correspond to non-zero entries in the filter h_i and are used to compute the center pixel at the next level. Gray dots are positions that the full undecimated wavelet transform would take into account but that are skipped by the À-Trous algorithm.

Algorithme À-Trous

We will describe a smoothing process to reduce the variance of the final ray traced image. This is motivated by the following observation: The incident irradiance at a single point on a surface is described by the integral over the hemisphere. Under interactive or real-time constraints a path tracer can only trace a single path per pixel thus estimating the integral with a single sample only. But if neighboring hemispheres are similar one would expect similar integrals. Therefore the smoothing tries to average samples with a similar hemisphere.

We will now first describe the (unweighted) À-Trous wavelet transform and then extend it afterwards to incorporate our edge-stopping function. [Burt 81] shows that wide Gaussian filters can be well approximated by repeated convolution with *generating kernels*. [Hols 89] uses this construction to compute the discrete wavelet transform, known as the *algorithme À-Trous* :

1. At level $i = 0$ we start with the input signal $c_0(p)$
2. $c_{i+1}(p) = c_i(p) * h_i$, where $*$ is the discrete convolution. The distance between the entries in the filter h_i is 2^i.

3. $d_i(p) = c_{i+1}(p) - c_i(p)$,
   where d_i are the detail or wavelet coefficients of level i.

4. if $i < N$ (number of levels to compute):
   increment i, go to step 2

5. $\{d_0, d_1, ..., d_{N-1}, c_N\}$ is the wavelet transform of c.

The reconstruction is given by

$$c = c_N + \sum_{i=N-1}^{0} d_i \tag{6.10}$$

Filter h is based on a B_3 spline interpolation $(\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16})$ [Murt 97]. At each level $i > 0$ the filter doubles its extent by filling in 2^{i-1} zeros between the initial entries. Thus the number of non-zero entries remains constant (see Figure 6.34).

Data-Dependent Weights to Preserve Edges

Edge-avoiding filtering is achieved by introducing a data-dependent weighting function. We extend the intensity-based edge-stopping function of the bilateral filter to combine multiple edge-stopping functions. The discrete convolution in step 2 becomes

$$c_{i+1}(p) = \frac{1}{k} \sum_{q \in \Omega} h_i(q) \cdot w(p,q) \cdot c_i(p) \tag{6.11}$$

with weight function w, pixel positions p, q, and Ω as the positions of non-zero coefficients in h_i. k is the normalization factor

$$k = \sum_{q \in \Omega} h_i(q) \cdot w(p,q) \tag{6.12}$$

and w are the combined edge-stopping functions from the ray-traced input image (rt), normal buffer (n) and position buffer (x)

$$w(p,q) = w_{rt} \cdot w_n \cdot w_x \tag{6.13}$$

$$w_{rt}(p,q) = e^{-\frac{||I_p - I_q||}{\sigma_{rt}^2}} \tag{6.14}$$

where I_p and I_q are the color values of the rt buffer at positions p and q. w_n and w_x are computed in the same way with their own σ_n and σ_x respectively.

We implement this filter in GLSL (see Listing 6.3) passing the step-width (i.e. number of zeroes between filter entries), filter h, parameters σ_{rt}, σ_n, σ_x as uniforms and images rt, n, x as textures.

Since it is hard to make assumptions about the noise contained in the ray-traced image, we choose our initial σ_{rt} to include variations of the scale of the maximum intensity of the ray-traced image. Note that the intensities can be bigger than 1 in our full-HDR environment. Thus the edge-stopping function depends at the first level on w_n and w_x only. At each pass we set σ^i_{rt} to $2^{-i}\sigma_{rt}$ thus allowing for smaller illumination variations to be smoothed.

Edges that are preserved by our edge-stopping function are still present at coarser levels of the transformation. Therefore we discard the finer levels of the wavelet transform, rendering step 3 of the À-Trous algorithm unnecessary, and directly use level N as output image.

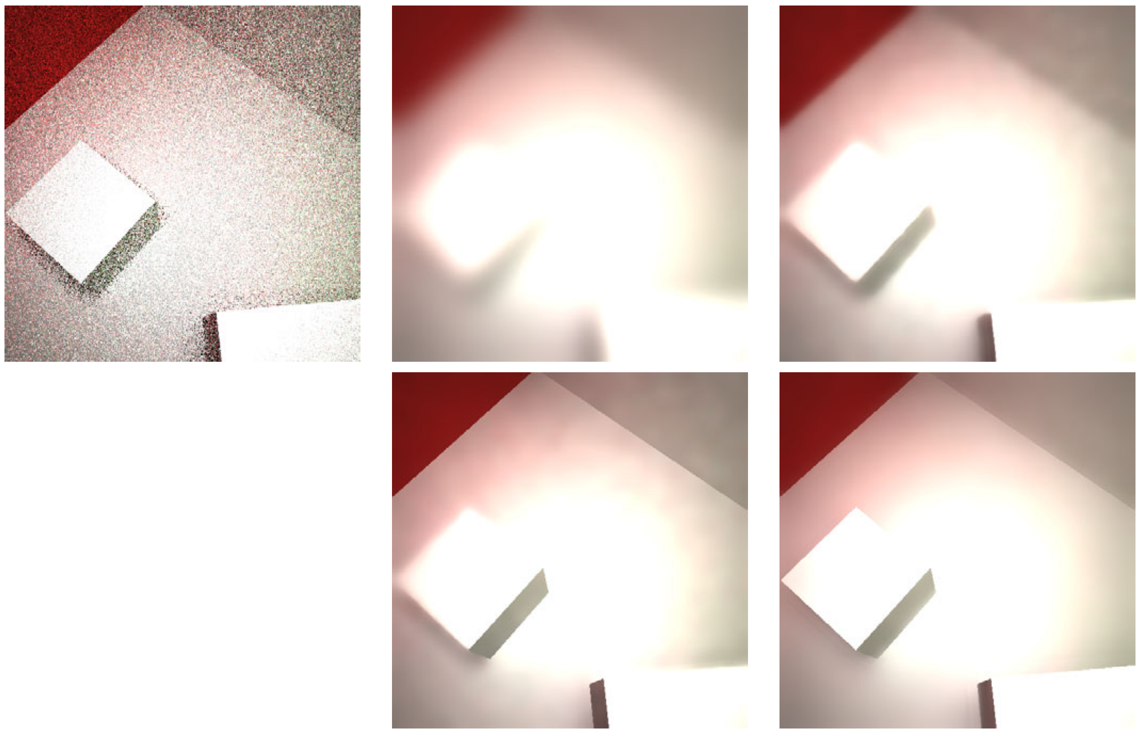


Figure 6.35: À-Trous filter with increasing number of edge weights: unfiltered rt buffer (input), no additional weights: (“pure” À-Trous), rt buffer only: σ_{rt} (bilateral approx.), two buffers: σ_{rt}, σ_n and all three buffers: $\sigma_{rt}, \sigma_n, \sigma_x$ (our result).

6.5.2 Implementation

We implemented the edge-avoiding À-Trous filter in GLSL which can be seen in Listing 6.3. After computing the input buffers (rt, normal, position) the filter is applied multiple times to the rt buffer. Each pass uses the previously smoothed result as input. All our buffer textures use a 16 bit floating point format.

The normal and position buffers are rasterized with OpenGL using frame buffer objects and passed to the smoothing shader via textures. Even though these buffers could easily be computed using the ray tracer we conserve bandwidth by computing them directly on the GPU. We directly display the final smoothed rt output buffer, or in the case of deferred shading, multiply the result with an OpenGL generated diffuse color buffer. As ray tracer we use a rather slow but very flexible CPU implementation. As Table 6.1 shows this is the bottleneck of our system and could be improved by using a more specialized ray tracer. Nevertheless we achieve interactive frame rates with our system. If not otherwise noted, performance measurements and comparisons were computed in this section on an Intel®Core$^{TM}2$ CPU (E6850) @ 3.00GHz with an NVIDIA GeForce GT 9500 GPU.

Deferred Shading

Deferred shading [Hagr 04] postpones the illumination computation at each pixel until a final rendering pass per light source. This pass takes a position, normal and diffuse buffer as input. We can apply this to our approach by using the ray tracer to compute the incident illumination at each visible pixel without the final material evaluation. This light buffer can then be smoothed using our À-Trous filter and be applied by multiplying it with the diffuse buffer. This allows to retain high detail in textures that would otherwise be blurred but can only be applied to diffuse surfaces.

Optimizations

We now discuss some possibilities to speed up a full rendering system that uses our filter by specializing the rendering algorithm.

While it would be possible to use a dense low resolution ray tracing buffer, one of the main advantages of our À-Trous filtering would be lost. Namely that in the full resolution edge buffer each sample location knows on which side of the edge it lies. We can however reduce the number of rays by sampling only every n-th pixel. Now the first steps of the filtering are related to an in-painting problem. This of course increases the variance of the final image even more and some features like hard shadows may be lost. Since in our system the major bottleneck is the ray tracer we get an almost linear speedup with the reduction of traced rays. Figure 6.36 shows results for sub-sampling. Note that for correct in-painting, the rt buffer should not be used as edge weight for the first iterations but only when the image is fully in-painted. For 2×2 sub-sampling this is after one iteration, for 4×4 after two.

Depending on bandwidth usage one can avoid tracing primary rays by downloading the depth or position buffer from the GPU. This might give a speedup when the path depth is low and the bandwidth is not a problem. In our case the resulting render times were slower by about 15%. This of course changes when a GPU ray tracer is used. In that case all data would already be available on the GPU.

```
uniform sampler2D colorMap, normalMap, posMap;
uniform float c_phi, n_phi, p_phi, stepwidth;
uniform float kernel[25];
uniform vec2 offset[25];

void main(void) {
  vec4 sum = vec4(0.0);
  vec2 step = vec2(1./512., 1./512.); // resolution
  vec4 cval = texture2D(colorMap, gl_TexCoord[0].st);
  vec4 nval = texture2D(normalMap, gl_TexCoord[0].st);
  vec4 pval = texture2D(posMap, gl_TexCoord[0].st);

  float cum_w = 0.0;
  for(int i = 0; i < 25; i++) {
    vec2 uv = gl_TexCoord[0].st + offset[i]*step*stepwidth;

    vec4 ctmp = texture2D(colorMap, uv);
    vec4 t = cval - ctmp;
    float dist2 = dot(t,t);
    float c_w = min(exp(-(dist2)/c_phi), 1.0);

    vec4 ntmp = texture2D(normalMap, uv);
    t = nval - ntmp;
    dist2 = max(dot(t,t)/(stepwidth*stepwidth),0.0);
    float n_w = min(exp(-(dist2)/n_phi), 1.0);

    vec4 ptmp = texture2D(posMap, uv);
    t = pval - ptmp;
    dist2 = dot(t,t);
    float p_w = min(exp(-(dist2)/p_phi),1.0);

    float weight = c_w * n_w * p_w;
    sum += ctmp * weight * kernel[i];
    cum_w += weight*kernel[i];
  }
  gl_FragData[0] = sum/cum_w;
}
```

Listing 6.3: This code shows the full implementation of our À-Trous filter in GLSL. Each step uses 5×5 texture look-ups. The uniform parameters are initialized by the main program to the parameters described previously. The first four lines are the declaration of the uniform variables. These are set by the calling program to contain the required values like the texture buffers (rt, normal and position) that are sampled or the kernel weight for each sample. The main function contains the actual filtering operation. This function is called for each output pixel and computes the resulting pixel value that is written in `gl_FragData[0]`. The code is for the 5×5 filter version and thus iterates over 25 samples. Inside the loop the buffers are sampled and the weight computed according to Section 6.5.1.

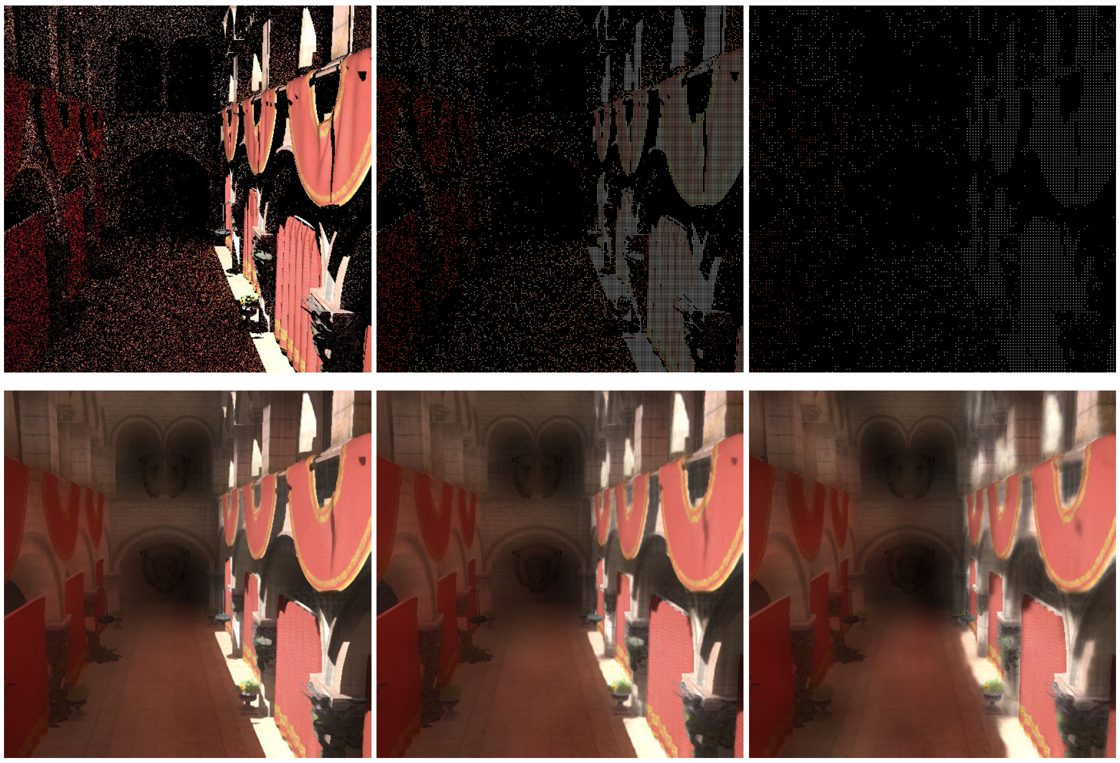


Figure 6.36: Sub-sampling the rt buffer. The top row shows the input using one sample per pixel (1.1 fps), one sample every 2×2 pixels (3.8 fps) and one sample every 4×4 pixels (11.0 fps). The bottom row shows the result after filtering where the loss of detail is clearly visible.

When the flexibility of arbitrary shaped light sources is not needed, the direct light computation can also be performed on the GPU using e.g. shadow maps or shadow volumes. This optimization is related to the previous one, but now the first shadow ray can also be omitted. In our case the speed-up was almost a factor of 2 when using a single point light source in the Sponza scene with one indirect bounce.

6.5.3 Results

Table 6.1 shows the performance of each individual part of our system for all scenes depicted in Figure 6.43. This measurement was done at a resolution of 512×512 with a single path per pixel and one indirect bounce. For each path, 4 rays are traced (primary and secondary plus two shadow rays). The measurements were averaged over 16 iterations for the shown camera perspective. The À-Trous filter was applied five times. As expected the À-Trous filter performance is independent of the input data and very fast. The filter can be adapted to different kinds of applications by specifically extending it to evaluate more input weight buffers. This is illustrated in Figure 6.35 for zero to three weights. The impact on the performance is shown in Table 6.2. This table also shows the performance of the À-Trous GPU filter when changing the resolution. Figure 6.37 shows a sharp shadow of a pole being preserved correctly while smoothing the other parts.

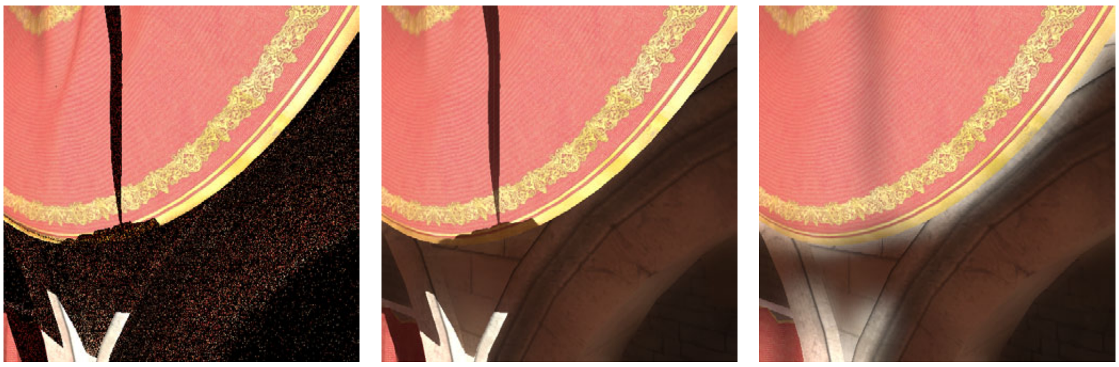


Figure 6.37: From left to right: Close up of the rt-input of a pole shadow in the Sponza scene. Result with our presented weighting approach. Result without rt buffer: the shadow is blurred.

The time spent on the OpenGL rendering varies extensively with scene complexity and number of textures. Depending on application and input data it might be more efficient to produce the required normal, position and diffuse buffer with the ray tracer and upload them to the GPU.

Since we use a general ray tracer it is easy to change for example the glossiness of materials or the size and shape of light sources. The effects of this are illustrated in the Figures 6.39 and 6.40.

Scene	RT (ms)	GL (ms)	Upload (ms)	À-Trous (ms)	FPS
Box	307.6	2.9	5.8	78.5 (5.6)	3.2
Sponza	835.2	35.5	4.4	78.6 (5.6)	1.13
Sibenik	510.2	8.5	5.2	78.6 (5.6)	1.9
Outdoor	316.4	2.6	6.3	78.6 (5.6)	3.0

Table 6.1: Individual timings for the test scenes shown in Figure 6.43 at a resolution of 512×512. RT is the total time to produce the input ray tracing buffer, GL the time to generate the position, normal and diffuse buffer using OpenGL. Upload is the time it takes to copy the ray tracing buffer from main memory to the GPU. À-Trous gives the time of five repeated applications of our filter on a GT 9500. The number in brackets is the time on an NVIDIA GTX 285. FPS are the final frames per second achieved by our testing system.

Res.	512×512			1024×1024			1920×1080		
# Iter	1	5	10	1	5	10	1	5	10
# W									
0	0.3	1.5	2.9	1.0	5.0	11.3	1.9	9.6	22.4
1	0.5	2.4	4.7	1.8	8.7	17.2	3.5	16.9	33.6
2	0.8	3.5	6.9	2.7	12.9	26.6	5.0	25.1	52.1
3	1.2	5.6	11.0	4.2	20.9	41.6	8.2	42.6	86.0

Table 6.2: Time in ms to compute the À-Trous filter on a NVIDIA GTX 285 at different resolutions (Res.). $5 \cdot 2^{\#Iter}$ corresponds to the total filter-size and # W is the number of weights. The original À-Trous algorithm does not use weights. Our implementation uses 3 weights. See Figure 6.35 for results with fewer weights.

Increasing Bounces and Samples per Pixel

The number of bounces can easily be increased. While slowing down the ray tracing it allows for a more correct global illumination solution. The first three images in Figure 6.38 show results with more bounces. To reduce the variance in the input buffer, more than one path per pixel can be traced. This is illustrated in the last two image columns in Figure 6.38 with 4 and 16 paths per pixel.

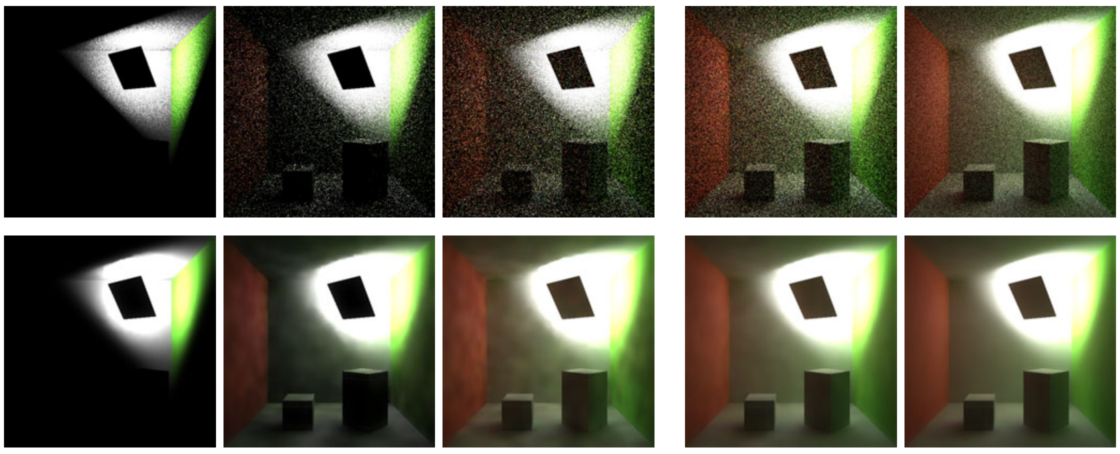


Figure 6.38: The top row shows the input, the bottom row the output. The first three columns show increasing number of bounces from 0 (direct light) to 1 and 5. The last two columns show the improvement when tracing more than one path per pixel (4 and 16 paths).

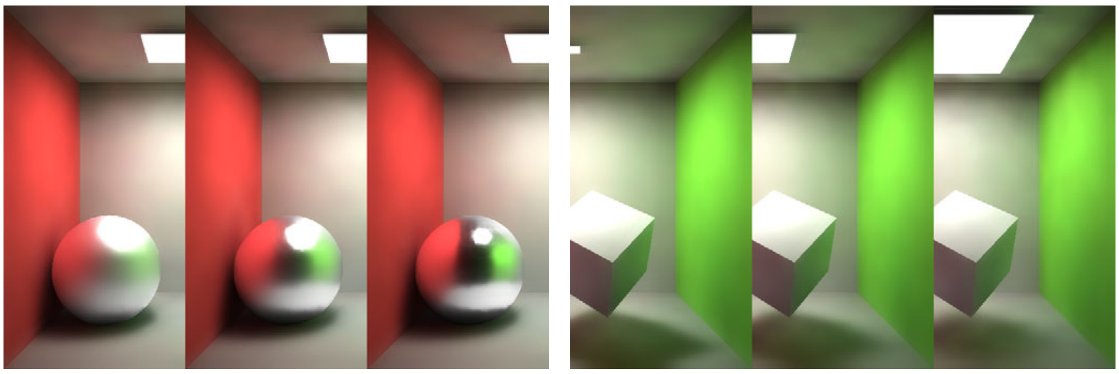


Figure 6.39: Varying the glossiness of the sphere and the size of the light source. The three images on the left show increasing glossiness on the sphere. The three images on the right show an increasing light source. Note the change in the shadow of the cube.

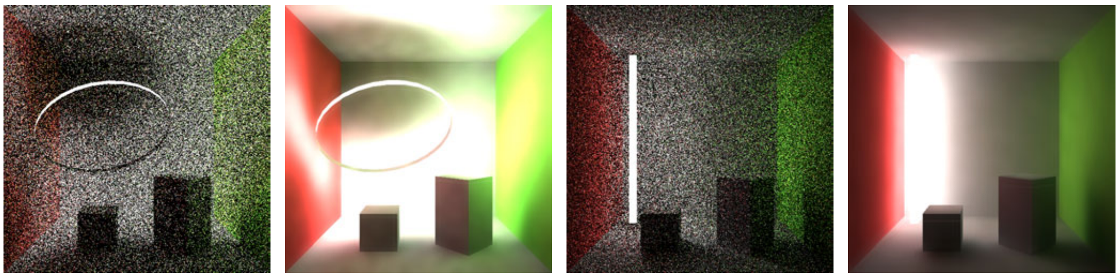


Figure 6.40: Arbitrarily shaped light sources. The images show the rt input buffer and the filtered output with two bounces. This shows the advantage of using a general ray tracer and performing image space filtering.

Comparison to Other Wavelet Bases

As compared to standard de-noising techniques our method allows one to discard the detail coefficients instead of shrinking them. In a high variance ray traced image a single bright pixel in a dark region may indicate a large smooth patch with the same mean irradiance. Such a pixel would result in a detail coefficient of very large magnitude which would not be sufficiently smoothed by wavelet shrinkage.

Figure 6.41 shows the comparison of our edge-avoiding À-Trous filter to other bases. As can be seen in the standard decimated wavelet transform (using CDF(2,2) and Red-Black wavelets [Uytt 98]) filtering out detail coefficients of large magnitude leads to visible artifacts resulting from the decimation. The edge avoiding versions suffer from the same artifacts. See for example the EAW CDF(2,2) image where it fails to preserve the two edges between the light source and the wall.

Adaptive wavelet rendering [Over 09] takes advantage of the hierarchical nature of decimated wavelet bases. They adaptively refine the wavelet representation and sam-

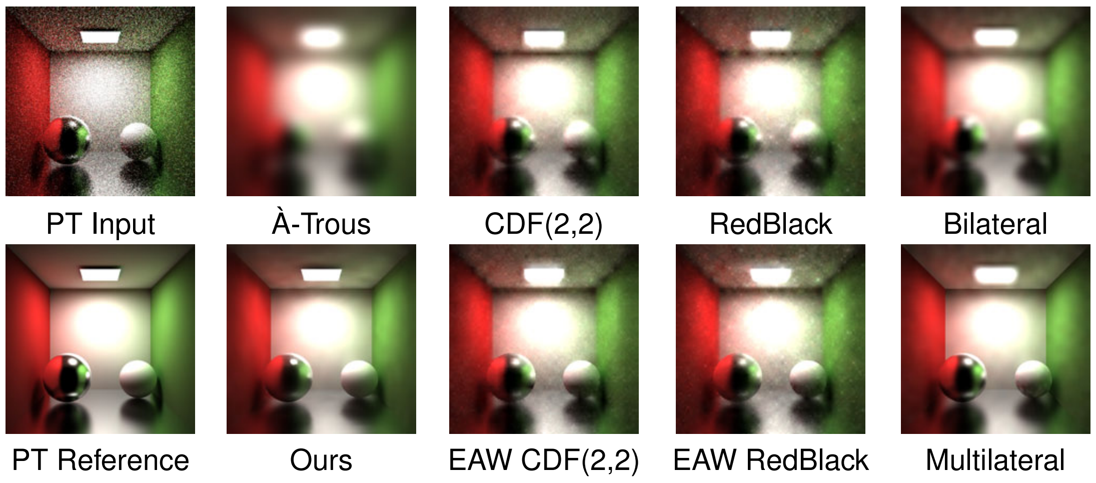


Figure 6.41: Comparison to other wavelet bases for filtering noisy Monte Carlo images. The first row shows the result of filtering with the standard wavelet transform. The second row shows the edge avoiding versions (using our edge-stopping function). We also include the results of the bilateral and multilateral filter, where the multilateral version is the extension of the standard bilateral filter with our multiple edge-stopping function.

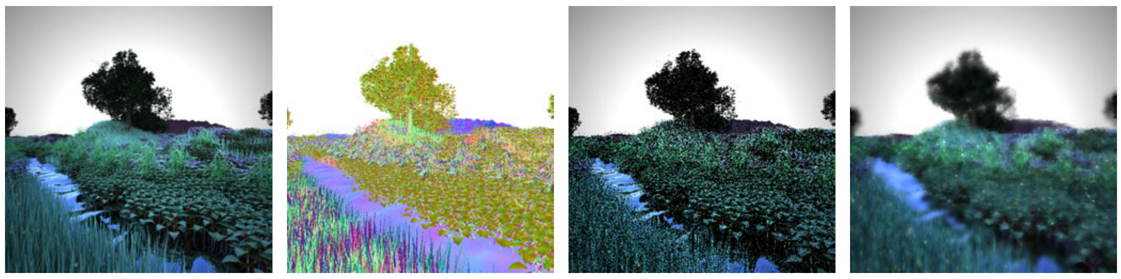


Figure 6.42: Failure case for our algorithm: A nature scene with 20 Million triangles. Due to many complex shadow casting objects the illumination variance is very high and a single sample per pixel is insufficient. Additionally there is severe aliasing in the normal buffer.

ple the support of the wavelets. While it would be interesting to carry over this approach to un-decimated bases, this is not straight forward because there is no implicit hierarchy anymore. It would also be interesting to apply our filtering technique to other global illumination algorithms like micro-rendering [Rits 09] or image space photon mapping [McGu 09].

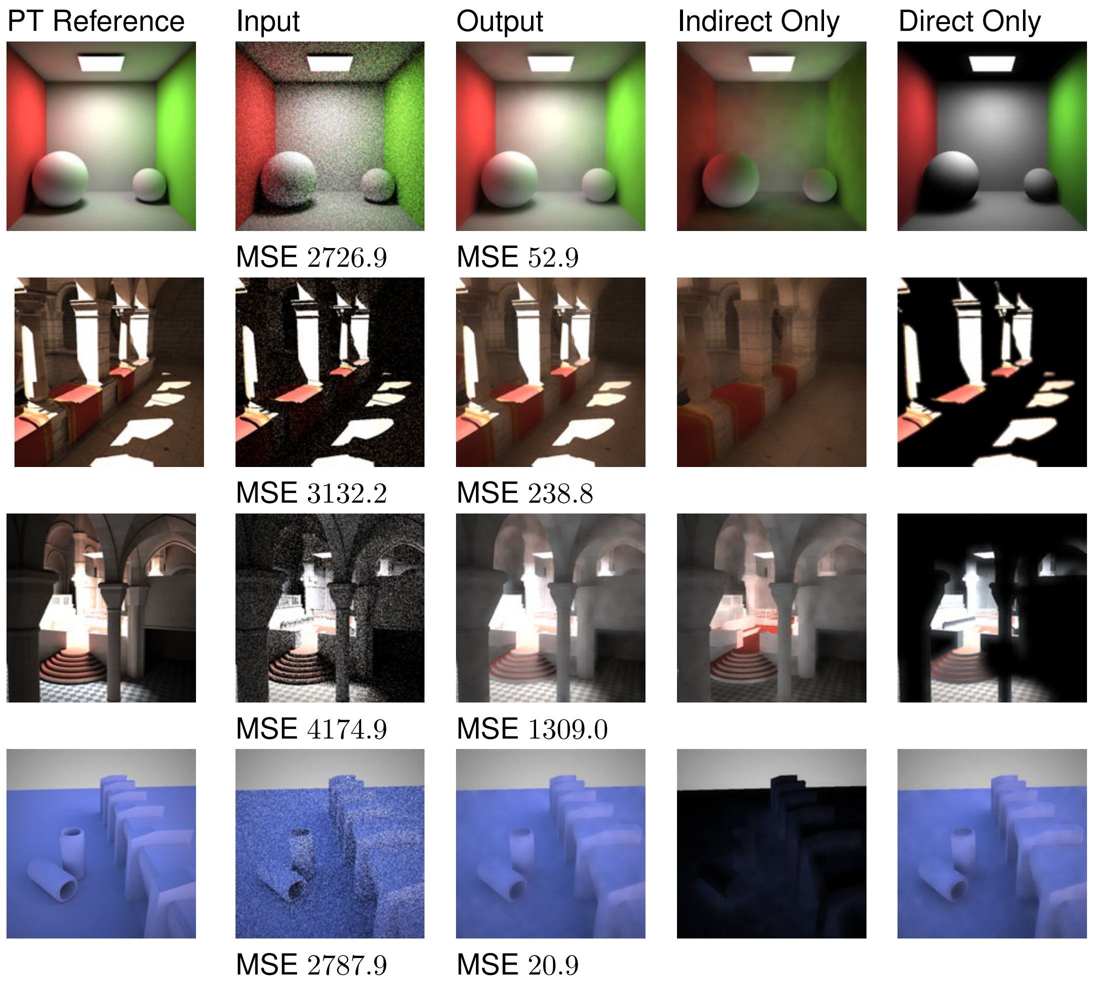


Figure 6.43: Test scenes for our filtering algorithm. Each image was computed at a resolution of 512×512 with one path per pixel and one indirect bounce. The reference image was computed with 4096 paths per pixel. MSE shows the mean square error to the reference solution. The last row shows an outdoor scene illuminated with a diffuse skylight.

6.5.4 Discussion

In this section we presented a novel filtering technique for highly noisy Monte Carlo global illumination images that can be used in interactive applications. By extending the À-Trous wavelet transform to incorporate an edge stopping function taking multiple buffers into account, we can successfully eliminate the noise while preserving sharp details. This filtering assumes that the filtered incident radiance is reflected smoothly but we also showed that it can be successfully applied for scenes with slightly glossy surfaces. This is the same distinction that we made in Section 6.3 with our path space partitioning method and in Section 2.3.4 for our material model. For higher quality output the same partitioning could be used here and would allow to add for example correct reflections or refractions to this method.

However, highly complex scenes are problematic for our smoothing algorithm since one sample per pixel (rt buffer as well as position and normal buffer) no longer suffice to capture the necessary information. This results in the loss of details and smoothing for example over shadows or object boundaries. Both problems are illustrated in Figure 6.42. Another problem is that due to the edge-stopping function the À-Trous wavelet transform is no longer energy conserving. While this is acceptable in some situations it has to be considered for high quality offline rendering.

Since our approach is very general in future work we plan to apply it to more advanced Monte Carlo global illumination algorithms like bi-directional path tracing and to investigate how a more sophisticated edge-stopping function could improve the results even further. Another interesting extension would be to incorporate adaptive sampling to avoid sampling in areas where the À-Trous filtering already gives a good result. A possibility to fix the energy conservation would be to use an additional channel in the wavelet transform and post-normalize [Fatt 09] the output buffer. Using a ray tracer that operates fully on the GPU is likely to provide a significant speedup for a rendering system using our filter.

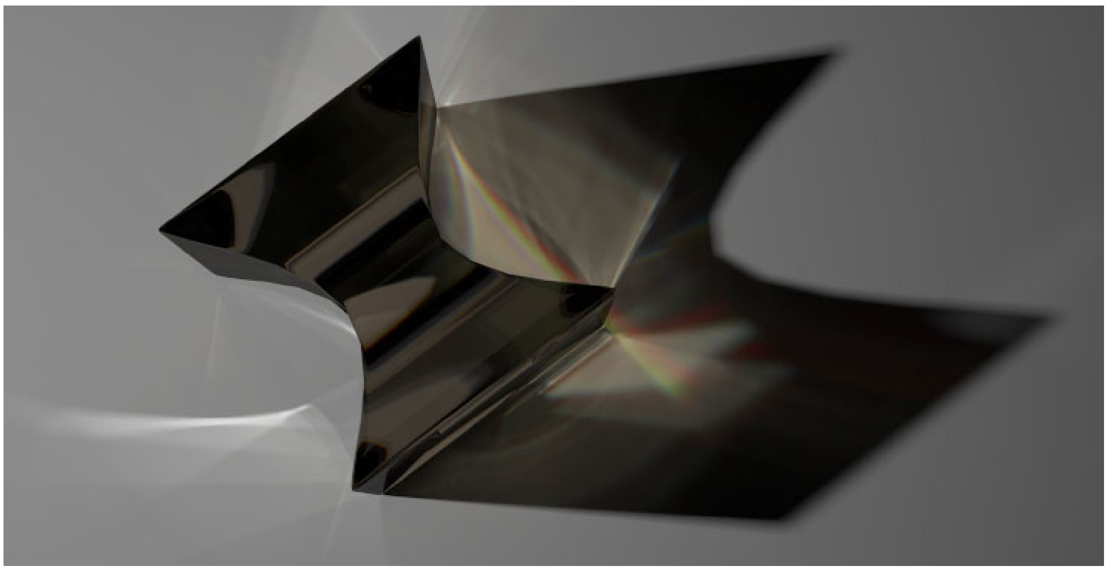


Figure 6.44: (745×380 0.8 hours) On demand spectral rendering allows to integrate dispersion into an RGB based render. The renderer itself performs all computations in RGB but a path may have an assigned wavelength that is used to create the correct dispersion effects.

6.6 On Demand Spectral Rendering

For efficiency reasons our system is based on RGB rendering. Yet it is possible to consistently integrate an approximation for spectral effects into our rendering system by on demand (or lazy) spectral rendering. In this chapter we first show a simple method to integrate wave-length dependent effects into our rendering system and then briefly present two applications that use spectral rendering and that benefit greatly from the acceleration of incoherent rays presented in this chapter. The first application is presented in Section 6.6.1. It is a complete simulation of realistic camera lenses. This requires a full global illumination renderer as presented in Chapter 6 and thus traces highly incoherent rays. The second application presented in Section 6.6.2 is a physically based simulation of fluorescent concentrators where physical processes are simulated by a Monte Carlo method that requires ray tracing. The ability to trace millions of incoherent rays per processor core greatly benefits the simulation times here as the ray tracing is the bottleneck.

Any rendering algorithm based on RGB can be augmented by a simple method to allow for basic spectral effects. This works by using the normal RGB transport until a ray hits for example a dispersive surface (e.g. glass). At this point the ray is assigned a randomly sampled wavelength from the original RGB color. In addition, we further assign a novel RGB color corresponding to the sampled wavelength. This RGB color is used as weight for all further color computations. It is important that, once a ray has an assigned wavelength, it will keep it until the ray terminates. The wavelength assignment is done for all rays generated in the system: eye-paths, photon-paths, and the random walks to generate the VPLs. This allows for the usual color caustics (photon paths), the chromatic aberration seen through refractive objects (eye-paths) and correct indirect illumination from colored caustics (VPLs). Figure 6.44 shows a rendering with a dispersive glass object.

6.6.1 Lens Simulation

Spectral effects have several applications in realistic rendering. Here we present some of the results of our camera lens simulation model. Parts of this have been published in [Stei 09]. It is a spectral example application that highlights the need for fast incoherent ray tracing. The approach incorporates insights from geometric diffraction theory, from optical engineering, and from glass science. It efficiently simulates all five monochromatic aberrations, spherical and coma aberration, astigmatism, field curvature, and distortion. Chromatic aberration, lateral color, and aperture diffraction are also considered. In this section we only shortly discuss the aberrations and flare effects to illustrate how incoherent rays are naturally generated by simulating lenses.

The camera model used in photo-realistic image synthesis plays an important role. The commonly used pinhole camera model creates unrealistically sharp and distortion free images that have infinite depth of field. So often, the approach to graphics is to simulate isolated effects separately. For example depth of field simulation [Cook 84] and fast approximations of depth of field [Kass 06] have been studied. The thick lens model [Kolb 95] provides a linear approximation with regard to the lens thickness, causing significant visual influence by optical path length manipulation [Pedr 06]. It originates from optical design analysis [Smit 07]. The linear transformation based on paraxial optics allows oneto express the imaging process of a whole system of lenses in a single transformation matrix. We used the thin lens model, a special case of the thick lens model to generate all the images shown in that chapter. Models like the thin and thick lens model are not able to simulate sophisticated optical effects like lens aberrations of higher order, or flare effects inside the system. Our approach based on Monte Carlo path generation methods extends this by certain important methods: wavelength dependent events, the use of real glass data and consideration of dielectric Fresnel interaction at every glass surface. The correct simulation of light interacting with glass surfaces is important for realistic results. [Devl 02] provides an overview over all necessary formulas. Based on this, we use Sellmaier definitions of typical optical glass types that can be found in the Schott glass catalog [Opti 09].

Aberrations

In the lens the varying thickness and incident angle due to the spherical surface causes a changing optical path length with varying ray height h (see Figure 6.45), i.e. distance of the ray to the optical axis. Consequently, the image suffers from so-called Seidel

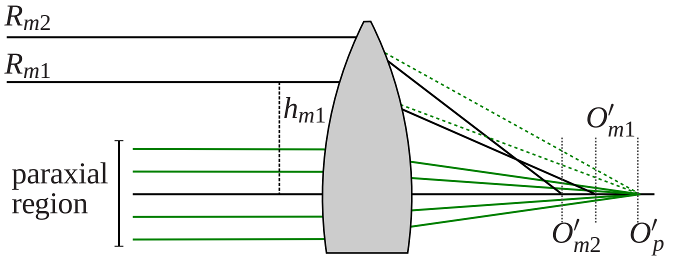


Figure 6.45: Comparison of Gaussian refraction (dashed) and Snell's refraction (solid) of marginal region rays.

aberrations [Ray 02, Smit 07]. Tracing rays through an optical system means applying Snell's law [Devl 02] repeatedly:

$$\eta(\lambda)\sin\theta = \eta'(\lambda)\sin\theta' \tag{6.15}$$

The Taylor expansion of $\sin\theta$ reveals that the incident angle appears in different orders.

$$\sin\theta = \sum_{n=0}^{\infty}(-1)^n\frac{\theta^{2n+1}}{(2n+1)!} = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} \pm \cdots \tag{6.16}$$

Discarding all terms of higher order than the first yields the paraxial approximation

$$\sin\theta \approx \theta, \tag{6.17}$$

which is valid for small angles θ. This linear approximation of Equation 6.16 is the central basis of all linear lens models like the thin or thick lens description. By truncating the higher order terms, aberrations for a larger ray height with increasing θ can not be simulated anymore. In contrast, we use Snell's law (Equation 6.15) for our computations and therefore the ray height is not limited. Figure 6.45 depicts the difference between linear Gaussian optics and reality for marginal rays with large height h. Seidel [Seid 57] classified five different forms of monochromatic aberrations introduced with the 3rd order. The effect that paraxial rays with large height are focused closer to the lens is called *spherical aberration*. For oblique rays, *coma aberration* causes comet-like tails to points. *Astigmatism* is caused by the fact that the ray height can vary in two directions, resulting in different image planes. *Field curvature* is the obvious consequence of using spherical surfaces. The image is not formed on a real plane, but on a curved surface. Distortion arises with marginal rays, becoming more or less magnified, and thus introducing either inward pincushion distortion or outward barrel distortion. *Chromatic aberrations* are caused by dispersion during transmission. Shorter wavelengths are focused closer to the lens because of stronger refraction compared to longer wavelengths.

Lens Design

Optical instruments usually consist of one or more spherically curved, axially symmetric glass surfaces. Using lenses in tandem, either cemented or with a gap in between, offers possibilities for aberration correction, imaging distance or effective focal length manipulation. A typical description of a lens design is outlined in Figure 6.46.

radius[mm]	thickness[mm]	material	radius[mm]
42.970	9.8	LAK9	19.2
-115.33	2.1	LLF7	19.2
306.840	4.16	air	19.2
aperture	4.0	air	15.0
-59.060	1.870	SF7	17.3
40.930	10.640	air	17.3
183.920	7.050	LAK9	16.5
-48.910	79.831	air	16.5

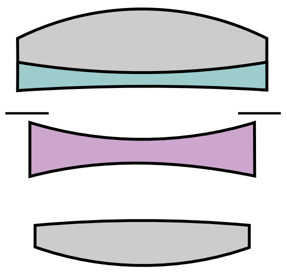


Figure 6.46: Tessar Design by Brendel (USP 2854889)[Smit 05], $f/2.8$, 100mm effective focal length (EFL).

Each row describes one spherical surface inside the device. The curvature follows a fundamental rule: A positive radius stands for a surface with center of curvature on the right side, and negative sign means that the center is on the left side. The thickness applies to the transmission distance along the optical axis. The material name is a code referencing an entry in the Schott glass catalog [Opti 09]. The last column represents the semi-diameter of the element. The aperture stop position and radius are defined in row 4. It is important to note that all values are scaled with the same factor when using the same lens design for a different focal length.

Flare Effects

Light in a lens housing is not only refracted towards the sensor. Internal reflection cannot be completely eliminated and become clearly visible in certain situations. A typical flare effect is oblique incoming light, that gets reflected at a surface and then reaches the image plane as can be seen in Figure 6.47.

In order to model the typical color variations of lens flares, absorption characteristics of material coatings have to be simulated. We approximate thin-film interference by including a look-up step right after Fresnel computation.

For the specific phenomenon of lens flares, we only need to consider direct light from the scene to obtain the light paths for reflection artifacts. Lens flares are best found by starting samples at light sources and deterministically connecting them to the lens. The resulting image delivers an independently viewable solution of the lens flare as shown in Figure 6.49 (left). The results are only correct for the predetermined setting of direct light. Mirror interactions are not accounted for in the light tracing pass but

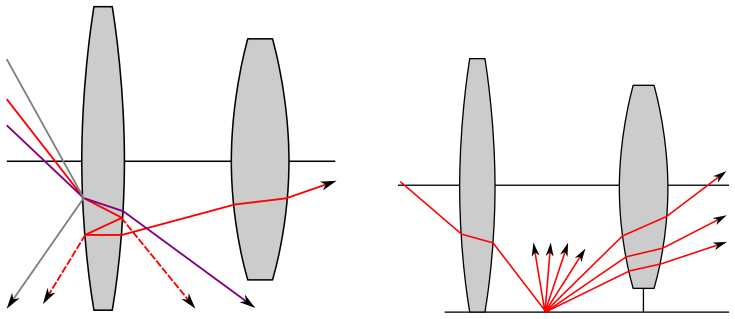


Figure 6.47: Secondary reflections inside the lens barrel can arise due to extreme angles of incidence, which cause internal reflections inside the elements, as seen on the left. The right schematic shows, that stray light occurs when the barrel has a certain reflectance.

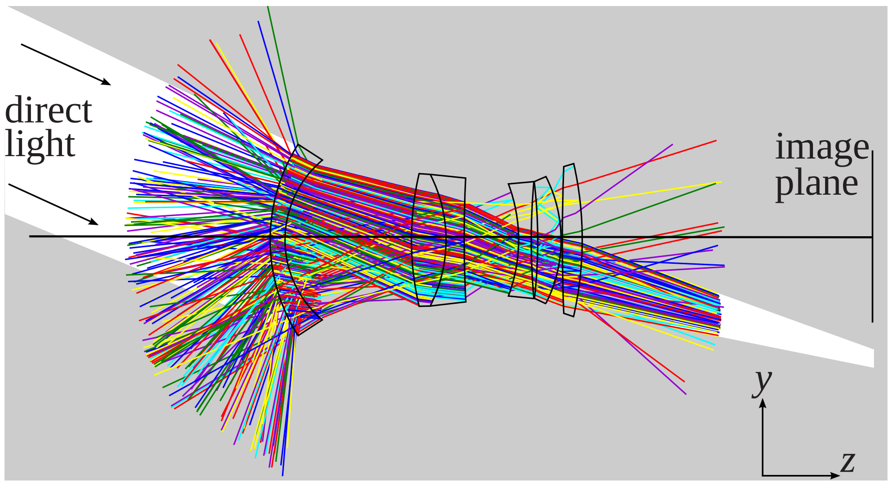


Figure 6.48: Lens tracing plot for an aperture-less Momiyama wide-angle [Smit 05] at 36mm with fix $f/2.8$, simulating lens flare either directly at the first surface or later. The image of the light source itself is not in the field of view. The reflected rays reaching the sensor create the flare effects. This image illustrates that simulating a lens creates highly incoherent rays.

could be easily included by extending the light paths and filtering out appropriately in the backward stage. Neglecting direct light in the backwards path tracing stage allows both results to be summed up to the final image in Figure 6.49 (right). The lens tracing plot in Figure 6.48 clearly shows regions on the sensor, where reflected light arrives. This plot also illustrates how incoherent the resulting rays are from the simulation.

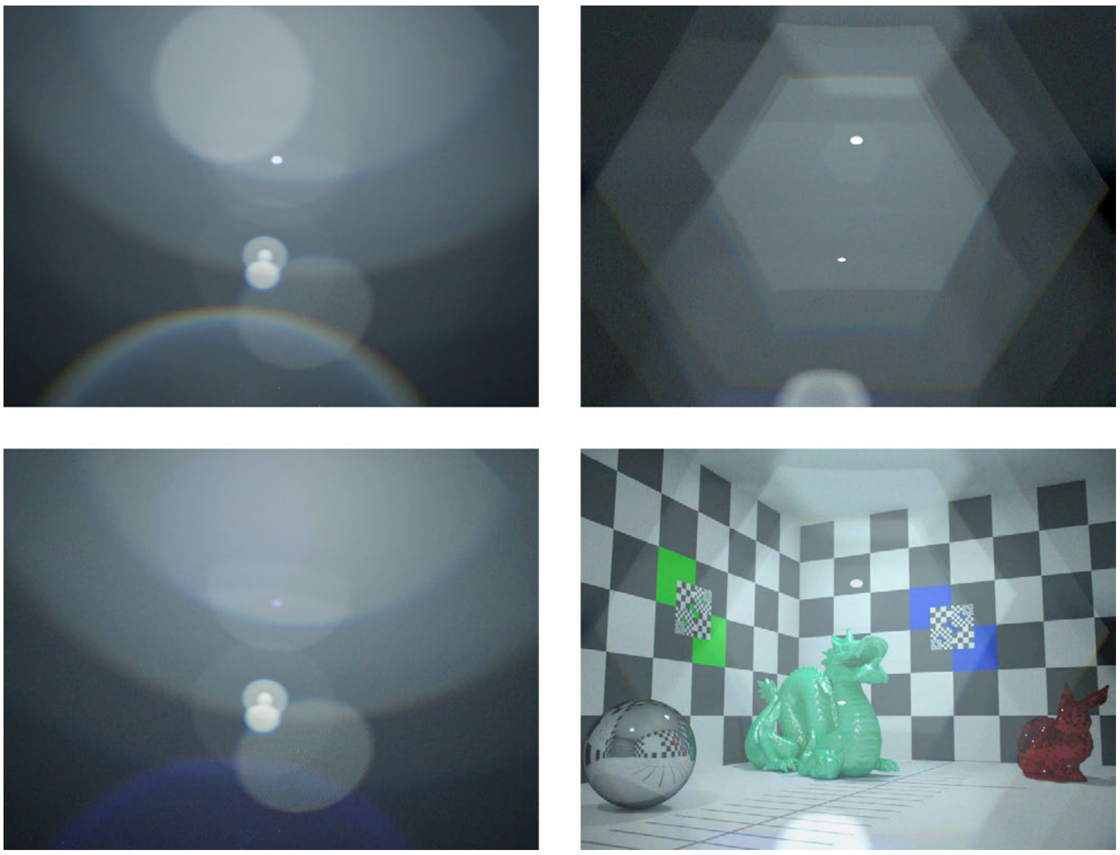


Figure 6.49: In the left images, a 33mm wide-angle lens by Mori [Smit 05] at $f/2.8$ was used. A lens surface was equipped with a modified reflection spectrum resembling a thin layer V-coating [Smit 07] (lower left). The right images show the lens flare result of Kimura wide-angle with $f/5.6$ clearly reproducing aperture ghosts. The combined image gives an impression of the resulting impact.

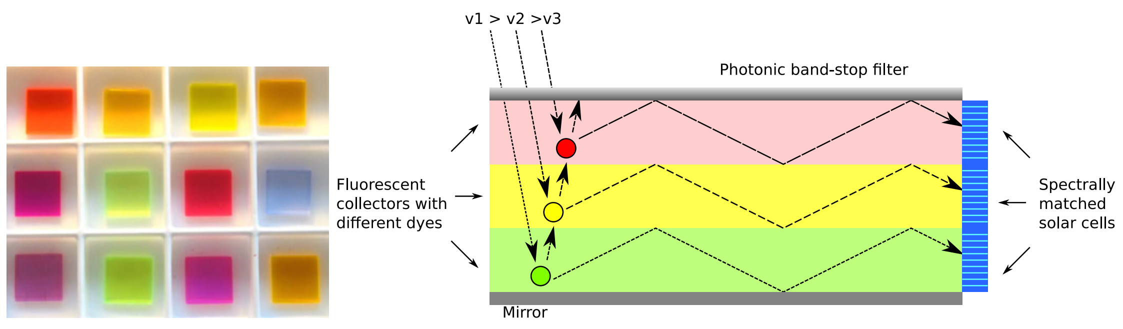

v1 > v2 >v3
Photonic band-stop filter
Fluorescent collectors with different dyes
Spectrally matched solar cells
Mirror

Figure 6.50: The left image shows a photograph of fluorescent concentrator samples used for the measurements. The right figure illustrates the concept of a fluorescent concentrator: the absorbed light is re-emitted at a different wavelength in the dye and then guided to the solar cell, with a very low probability of re-absorption. One can improve this by using a concentrator stack, as shown in the figure.

6.6.2 Fluorescent Concentrator Simulation

Spectral light transport has of course also applications for directly simulating physical processes without the need to produce images. One such application is the simulation of fluorescence. In this section we investigate fluorescence in the context of fluorescent concentrators. Fluorescent concentrators (see Figure 6.50 for a photograph) are developed in solar energy research in order to improve the efficiency and applicability of solar cells. This is achieved by concentrating even diffuse light on cloudy days on small solar cells, using a fluorescent dye enclosed in low-cost acrylic glass. One advantage is their ability to use also the diffuse light on cloudy days for solar cells and the reduction of the costs for photovoltaic systems. We developed a Monte Carlo simulation model for fluorescent concentrators that makes it possible to test and improve different setups and to gain further insight into the physical processes. This work was published in [Bend 08].

A fluorescent concentrator is made from PMMA (acrylic glass) and contains a dye. Figure 6.50 illustrates the basic setup for one application of a fluorescent concentrator: If light enters the concentrator and hits a dye molecule it will be absorbed and re-emitted at a different wavelength according to the photo-luminescence spectrum (PL-spectrum). Since the PL-spectrum is shifted to longer wavelengths as compared to the absorption spectrum, light is unlikely to be re-absorbed and therefore travels through the medium mostly undisturbed. By designing the fluorescent concentrator in a certain shape, light is trapped inside and can be guided to the solar cell due to total internal reflection.

Through Monte Carlo simulation, it is possible to check the existing physical models of fluorescent concentrators and to thoroughly test new ideas. Thus enhanced concepts can be invented and proved before their realization. The data and techniques acquired during our investigations can also be applied to Monte Carlo ray tracing applications to simulate fluorescence in the context of computer graphics.

Our work is based on the dissertation of Zastrow [Zast 81] and the publications of

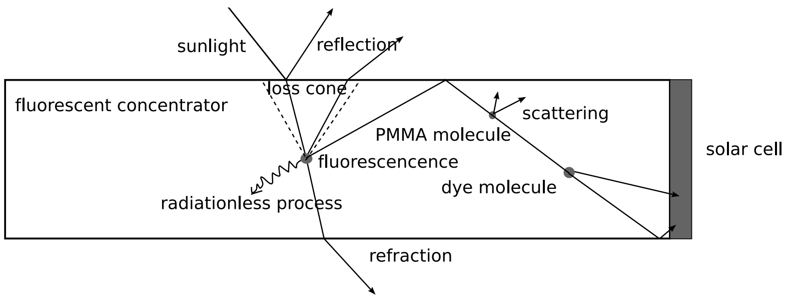


Figure 6.51: Processes in a fluorescent concentrator.

Peters, Goldschmidt et al. [Gold 06, Pete 07] about fluorescent concentrators. It focuses on the physically correct simulation of fluorescent concentrators in order to gain deeper knowledge about the physical processes involved and to optimize the concentrator. Heidler [Heid 82] developed a Monte Carlo model for fluorescent concentrators in 1982. His model was made for an efficiency analysis for the concentrator. Carrascosa et al. [Carr 83] were first to describe ray tracing of fluorescent concentrators. In the last few years Burgers et al. [Burg 05, Burg 06] and Schüler et al. [Schu 07] used Monte Carlo ray tracing for simulations of fluorescent concentrators and quantum dot solar concentrators, respectively. We conduct additional experiments, as e.g. the angular experiment and use the fast incoherent ray tracing developed in Chapter 3 which make a quick simulation possible and even allow for interactive previews for dynamically changing material parameters.

Simulation

Parts of the processes in a fluorescent concentrator are well known and can be described by analytical models. But for example the behavior of the dye is still not fully understood and difficult to measure in an experimental setup. In order to build a simulation we have to rely on existing models and experimental data. The required input parameters are reconstructed from the measured absorption and photo-luminescence spectrum.

We used a range from 300 to 800 nanometers of the AM 1.5 spectrum [Amer] for the input wavelengths. The dye in the concentrator we used to test our model was BA241. This data can easily be replaced to simulate different conditions, for instance a different dye.

We decided to simulate single particles (photons) with one wavelength instead of weighted paths. These particles are traced on their way through the fluorescent concentrator. Each of the interactions is modeled as independent event. This approach requires more rays to be shot than other methods, which transport differential energy per path, but on the other hand it is less error prone in the implementation and numerically more robust.

The use of the our ray tracing method not only allowed to quickly simulate physical measurements but to use the described method also for image generation of complex models build from the described material. For example, without fluorescence, the

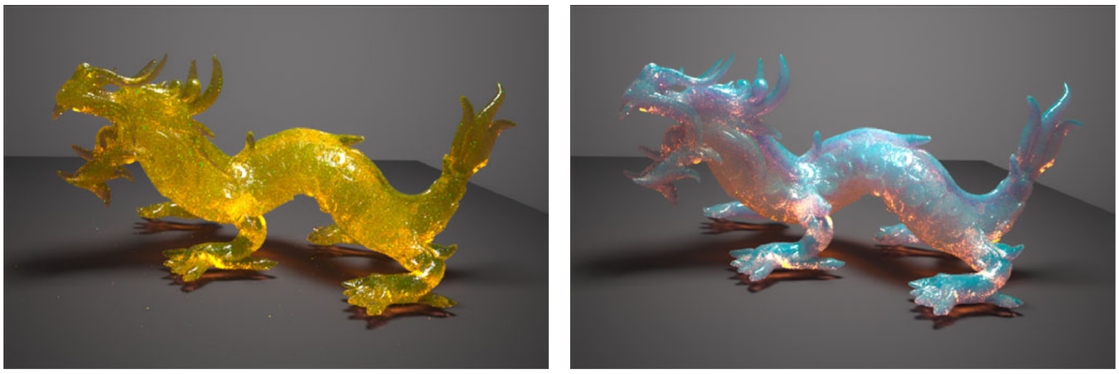


Figure 6.52: A dragon with fluorescent dye (left) and the same dragon rendered without fluorescence (right).

dragon model would look like the right image in Figure 6.52. Employing just an ordinary yellow dye would change the color of the caustics under the dragon. Fast ray tracing allows for interactive feedback when changing parameters like lighting conditions and material properties of the fluorescent concentrator. For simple shapes, as the original sample, even interactive changes to the geometry would be possible. The visualization system only draws one sample per pixel when parameters are changed (see Figure 6.53), but quickly accumulates more samples to a converging image if the parameters are left unchanged.

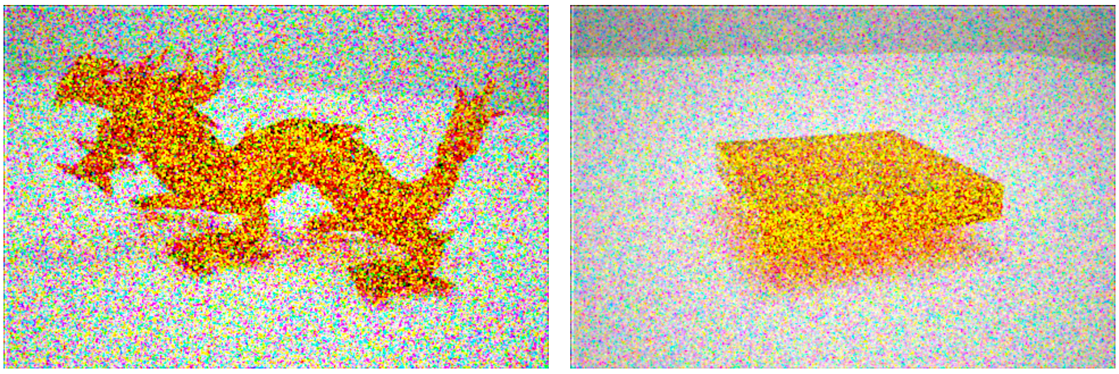


Figure 6.53: Interactive previews of a fluorescent dragon and the sample.

7 Conclusion

In this thesis we investigated high performance ray tracing and its application to physically based light transport simulation. One of the primary goals was to enable high quality production rendering using global illumination and physically based algorithms. With the architecture that was presented here, we achieved this goal. We provided solutions for offline high quality final frame rendering as well as fast interactive preview that achieves high quality within real-time constraints. This full rendering system was presented in Chapter 6, where we developed all the individual building blocks needed for a full production system. With our path space partitioning method, scenes with non-physical materials can be robustly rendered and the images are guaranteed to converge to a solution. We also showed that when using physically correct materials our system produces the correct result and we provided examples that it can be used directly for quantitative physical simulations.

To achieve the required performance for our light transport algorithms we developed a ray tracing acceleration structure in Chapter 3 that is not only faster than all previously available triangle based methods, but also uses a lot less memory. This data structure was specifically optimized for incoherent rays that occur in global illumination simulation. We also considered the scene representation in Chapter 4 and developed, among other strategies, a triangle splitting pre-process that further accelerates ray tracing. This pre-process is again fully transparent to the rendering system and is part of the tree construction. All the algorithms we developed also make use of the hardware features of modern processors and we show that a few low level optimizations can greatly improve the ray tracing performance. With our investigation of software caching in Chapter 5 we further sped up incoherent rays and the visibility computations without the need to explicitly construct coherent data access. All these methods combined provide the necessary foundation for our high quality production rendering system and the high performance preview system.

7.1 Future Work

Computer architecture is a moving target. In recent years we have seen some major architectural shifts in commodity hardware, for example, with the introduction of freely programmable GPUs. CPU architecture is also evolving at a quick rate. This means that many of the more hardware related details of implemented algorithms might need to be adjusted for future hardware generations. Especially the actual impact of SIMD units wider than 4 will be an interesting future research topic as soon as this hardware is actually available.

Another interesting aspect is the choice of scene representation. We chose the triangle representation because it allows for high performance and relatively complex scenes. But the strive for more complexity quickly reaches the limits of what can be stored in core. Out of core techniques and streaming were developed to cope with the needed complexity but are limited by bandwidth. In the future it will be interesting to see if the triangle representation remains the best choice. Procedural methods that are tightly coupled with the ray tracing method might allow for much higher complexity. But these techniques are not yet ready for a full production environment and a lot of research will be needed to develop the necessary data structures, algorithms, and editing tools.

The speed of the actual rendering is also an important research area. Even though we presented many improvements, photo realistic image generation is still a slow process, even on the fastest hardware available. One potential improvement is to find clever approximations that do not compromise the final quality of an image. However, defining image quality is a subjective problem. Thus the correct solution, in the sense of Monte Carlo integration of the full light transport equation, should always be of prime importance in the development of new algorithms. The improvement of ray tracing itself is not only important to accelerate existing techniques but, as shown in this thesis with the simulation for fluorescent concentrators, it can enable new applications that were previously impossible to achieve in a reasonable amount of time. Providing physically correct simulations at nearly interactive speeds will change the way physicists perform virtual experiments. These techniques are also not limited to light transport simulation and possible applications range from chemistry to medical applications.

List of Tables

List of Figures

Listings

Bibliography

[Aila 10] T. Aila and T. Karras. “Architecture Considerations for Tracing Incoherent Rays”. In: *Proc. High-Performance Graphics 2010*, pp. 113–122, 2010.

[Aman 87] J. Amanatides and A. Woo. “A Fast Voxel Traversal Algorithm for Ray Tracing”. In: *Eurographics, 87(3)*, pp. 3–10, 1987.

[Amer] American Society for Testing and Materials. “Reference Solar Spectral Irradiance: Air Mass 1.5”. http://rredc.nrel.gov/solar/spectra/am1.5/.

[Appe 68] A. Appel. “Some techniques for shading machine renderings of solids”. In: *AFIPS ’68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 37–45, 1968.

[Arik 05] O. Arikan, D. Forsyth, and J. O’Brien. “Fast and detailed approximate global illumination by irradiance decomposition”. *ACM Trans. Graph.*, Vol. 24, pp. 1108–1114, July 2005.

[Ashi 00] M. Ashikhmin and P. Shirley. “An Anisotropic Phong BRDF Model”. *Journal of Graphics Tools*, Vol. 5, No. 2, pp. 25–32, 2000.

[Ashi 06] M. Ashikhmin and J. Goyal. “A reality check for tone-mapping operators”. *ACM Trans. Appl. Percept.*, Vol. 3, No. 4, pp. 399–411, 2006.

[Atki 00] K. Atkinson. “The planar radiosity equation and its numerical solution”. *IMA Journal of Numerical Analysis*, Vol. 20, No. 2, pp. 303–332, 2000.

[Ball 93] T. Ball and J. Larus. “Branch prediction for free”. In: *PLDI ’93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pp. 300–313, 1993.

[Bend 08] M. Bendig, J. Hanika, H. Dammertz, J. C. Goldschmidt, M. Peters, and M. Weber. “Simulation of fluorescent concentrators”. In: *Proc. 2008 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 93–98, 2008.

[Bent 03] C. Benthin, I. Wald, and P. Slusallek. “A scalable approach to interactive global illumination”. *Computer Graphics Forum*, Vol. 22, No. 3, pp. 621–630, 2003. (Proceedings of Eurographics).

[Bent 06] C. Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006.

[Bent 75] J. Bentley. “Multidimensional binary search trees used for associative searching”. *Commun. ACM*, Vol. 18, No. 9, pp. 509–517, 1975.

[Berg 92] M. de Berg. *Efficient Algorithms for Ray Shooting and Hidden Surface Removal*. PhD thesis, Rijksuniversiteit te Utrecht, 1992.

[Blin 77] J. Blinn. “Models of light reflection for computer synthesized pictures”. *Computer Graphics (Proc. SIGGRAPH ’77)*, pp. 192–198, 1977.

[Boli 98] M. Bolin and G. Meyer. “A perceptually based adaptive sampling algorithm”. *ACM Transactions on Graphics (Proc. SIGGRAPH 1998)*, pp. 299–309, 1998.

[Boul 08] S. Boulos, I. Wald, and C. Benthin. “Adaptive ray packet reordering”. *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, Vol. , No. , pp. 131–138, Aug. 2008.

[Budg 08] B. Budge, D. Coming, D. Norpchen, and K. Joy. “Accelerated building and ray tracing of restricted BSP trees”. In: *Proc. 2008 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 167–174, 2008.

[Burg 05] A. Burgers, L. Slooff, R. Kinderman, and J. van Roosmalen. “Modelling of luminescent concentrators by ray-tracing”. In: *Proceedings of the 20th European Photovoltaic Solar Energy Conference and Exhibition*, 2005.

[Burg 06] A. Burgers, L. Slooff, A. Buchtemann, and J. van Roosmalen. “Performance of single layer luminescent concentrators with multiple dyes”. In: *Conference Record of the 2006 IEEE 4th World Conference on Photovoltaic Energy Conversion*, pp. 198–201, 2006.

[Burt 81] P. Burt. “Fast filter transform for image processing”. *Computer Graphics and Image Processing*, Vol. 16, No. 1, pp. 20 – 51, 1981.

[Carr 83] M. Carrascosa, F. Agullo-Lopez, and S. Unamuno. “Monte Carlo simulation of the performance of PMMA luminescent solar collectors”. *Applied Optics*, Vol. 22, pp. 3236–3241, 1983.

[Cell 09] *Cell Broadband Engine Programming Handbook Version 1.12*. IBM®, 2009.

[Chen 91] S. Chen, H. Rushmeier, G. Miller, and D. Turner. “A progressive multi-pass method for global illumination”. In: *SIGGRAPH ’91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pp. 165–174, 1991.

[Chou 05] P. Choudhury and J. Tumblin. “The trilateral filter for high contrast images and meshes”. In: *SIGGRAPH ’05: ACM SIGGRAPH 2005 Courses*, p. 5, 2005.

[Chri 02] P. Christensen. “Photon Mapping Tricks”. In: *A Practical Guide to Global Illumination using Photon Mapping. SIGGRAPH 2002 Course Note 43. ACM*, July 2002.

[Chri 03] P. Christensen, D. Laur, J. Fong, W. Wooten, and D. Batali. “Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes”. *Comput. Graph. Forum*, Vol. 22, No. 3, pp. 543–552, 2003.

[Chri 06] P. Christensen, J. Fong, D. Laur, and D. Batali. “Ray Tracing for the Movie ’Cars’”. In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 73–78, 2006.

[Clar 05] P. Clarberg, W. Jarosz, T. Akenine-Möller, and H. Jensen. “Wavelet importance sampling: efficiently evaluating products of complex functions”. *ACM Transactions on Graphics*, Vol. 24, No. 3, pp. 1166–1175, 2005.

[Clar 08] P. Clarberg and T. Akenine-Möller. “Practical product importance sampling for direct illumination”. In: *Computer Graphics Forum (Proc. of Eurographics 2008)*, pp. 681–690, 2008.

[Clin 06] D. Cline, K. Steele, and P. Egbert. “Lightweight Bounding Volumes for Ray Tracing”. *Journal of Graphics Tools*, Vol. 11, No. 4, pp. 61–71, 2006.

[Cook 84] R. Cook, T. Porter, and L. Carpenter. “Distributed ray tracing”. *Computer Graphics (Proc. SIGGRAPH ’84)*, pp. 137–145, 1984.

[Cort 09] *Cortex-A8 Technical Reference Manual*. ARM®, 2009.

[Dach 05] C. Dachsbacher and M. Stamminger. “Reflective shadow maps”. In: *I3D ’05: Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pp. 203–231, 2005.

[Dach 06] C. Dachsbacher and M. Stamminger. “Splatting indirect illumination”. In: *I3D ’06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 93–100, 2006.

[Daly 93] S. Daly. “The visible differences predictor: an algorithm for the assessment of image fidelity”. In: *Digital Images and Human Vision*, pp. 179–206, MIT Press, 1993.

[Damm 06] H. Dammertz and A. Keller. “Improving Ray Tracing Precision by World Space Intersection Computation”. In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 25–32, 2006.

[Damm 08a] H. Dammertz, J. Hanika, and A. Keller. “Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays”. In: *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering)*, pp. 1225–1234, 2008.

[Damm 08b] H. Dammertz and A. Keller. “Edge Volume Heuristic - Robust Triangle Subdivision for Improved BVH Performance”. In: *Proc. 2008 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 155–158, 2008.

[Damm 09a] H. Dammertz and J. Hanika. “Plane Sampling for Light Paths from the Environment Map”. *Journal of Graphics, GPU and Game Tools*, Vol. 14, No. 2, pp. 25–31, 2009.

[Damm 09b] H. Dammertz, J. Hanika, A. Keller, and H. Lensch. “A Hierarchical Automatic Stopping Condition for Monte Carlo Global Illumination”. In: *Proc. of the WSCG 2009*, pp. 159–164, 2009.

[Damm 09c] S. Dammertz, H. Dammertz, A. Keller, and H. Lensch. “Textures on Rank-1 Lattices”. *Comput. Graph. Forum*, Vol. 28, No. 7, pp. 1945–1954, 2009.

[Damm 10a] H. Dammertz, A. Keller, and H. Lensch. “Progressive Point-Light-Based Global Illumination”. *Computer Graphics Forum*, Vol. 29, No. 8, pp. 2504–2515, 2010.

[Damm 10b] H. Dammertz, D. Sewtz, J. Hanika, and H. Lensch. “Edge-Avoiding A-Trous Wavelet Transform for fast Global Illumination Filtering”. In: *Proc. High Performance Graphics 2010*, pp. 67–75, 2010.

[Debe 98] P. Debevec. “Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography”. *Proc. of SIGGRAPH ’98*, pp. 189–198, 1998.

[Devl 02] K. Devlin, A. Chalmers, A. Wilkie, and W. Purgathofer. “STAR: Tone Reproduction and Physically Based Spectral Rendering”. In: *State of the Art Reports, Eurographics 2002*, pp. 101–123, 2002.

[Dipp 85] M. Dippé and E. Wold. “Antialiasing through stochastic sampling”. *Computer Graphics (Proc. SIGGRAPH ’85)*, pp. 69–78, 1985.

[Dmit 04] K. Dmitriev and H.-P. Seidel. “Progressive Path Tracing with Lightweight Local Error Estimation”. In: B. Girod, M. Magnor, and H.-P. Seidel, Eds., *Vision, modeling, and visualization 2004 (VMV-04)*, pp. 249–254, Akademische Verlagsgesellschaft Aka, Stanford, USA, 2004.

[Drep 07] U. Drepper. “What Every Programmer Should Know About Memory”. Tech. Rep., Red Hat, Inc., 2007.

[Drie 05] T. Driemeyer and R. Herken. *Programming Mental Ray (Mental Ray Handbooks) (mental ray®Handbooks)*. Springer-Verlag New York, Inc., 2005.

[Dura 02] F. Durand and J. Dorsey. “Fast bilateral filtering for the display of high-dynamic-range images”. In: *SIGGRAPH*, pp. 257–266, 2002.

[Dutr 06] P. Dutré, K. Bala, and P. Bekaert. *Advanced Global Illumination*. AK Peters, Ltd., 2006.

[Dutr 93] P. Dutré, E. Lafortune, and Y. Willems. “Monte Carlo light tracing with direct computation of pixel intensities”. In: *Proceedings of Compugraphics ’93*, pp. 128–137, 1993.

[Edwa 06] D. Edwards, S. Boulos, J. Johnson, P. Shirley, M. Ashikhmin, M. Stark, and C. Wyman. “The halfway vector disk for BRDF modeling”. *ACM Transactions on Graphics*, Vol. 25, No. 1, pp. 1–18, 2006.

[Eise 04] E. Eisemann and F. Durand. “Flash photography enhancement via intrinsic relighting”. In: *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, pp. 673–678, 2004.

[Erns 07] M. Ernst and G. Greiner. “Early Split Clipping for Bounding Volume Hierarchies”. In: *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 73–78, 2007.

[Erns 08] M. Ernst and G. Greiner. “Multi bounding volume hierarchies”. In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pp. 35 –40, 2008.

[Fabi 09] B. Fabianowski and J. Dingliana. “Interactive Global Photon Mapping”. *Computer Graphics Forum*, Vol. 28, No. 4, pp. 1151–1159, 2009.

[Farr 04] J. Farrugia and B. Péroche. “A progressive rendering algorithm using an adaptive perceptually based image metric”. *Computer Graphics Forum*, Vol. 23, pp. 605–614, 2004.

[Fatt 07] R. Fattal, M. Agrawala, and S. Rusinkiewicz. “Multiscale shape and detail enhancement from multi-light image collections”. In: *SIGGRAPH ’07: ACM SIGGRAPH 2007 papers*, p. 51, 2007.

[Fatt 09] R. Fattal. “Edge-avoiding wavelets and their applications”. *ACM Trans. Graph.*, Vol. 28, No. 3, pp. 1–10, 2009.

[Fole 94] J. Foley, R. L. Phillips, J. Hughes, A. v. Dam, and S. Feiner. *Introduction to Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[Furt 07] T. Furtak, J. Amaral, and R. Niewiadomski. “Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms”. In: *SPAA ’07: Proc. of the 19th annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 348–357, 2007.

[Geim 06] M. Geimer. *Interaktives Ray Tracing*. PhD thesis, Koblenz-Landau University, Germany, 2006.

[Glas 89] A. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.

[Glas 94] A. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers Inc., 1994.

[Gold 06] J. Goldschmidt, S. Glunz, A. Gombert, and G. Willeke. “Advanced Fluorescent Concentrators”. In: *Proceedings of the 21st European Photovoltaic Solar Energy Conference*, 2006.

[Gold 87] J. Goldsmith and J. Salmon. “Automatic Creation of Object Hierarchies for Ray Tracing”. *IEEE Computer Graphics & Applications*, Vol. 7, No. 5, pp. 14–20, 1987.

[Gold 91] D. Goldberg. “What every computer scientist should know about floating-point arithmetic”. *ACM Comput. Surv.*, Vol. 23, No. 1, pp. 5–48, 1991.

[Hach 08a] T. Hachisuka, W. Jarosz, P. Weistroffer, K. Dale, G. Humphreys, M. Zwicker, and H. Jensen. “Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing”. *ACM Transactions on Graphics (Proceedings of SIGGRAPH ’08)*, p. , 2008.

[Hach 08b] T. Hachisuka, S. Ogaki, and H. Jensen. “Progressive photon mapping”. In: *SIGGRAPH Asia ’08: ACM SIGGRAPH Asia 2008 papers*, pp. 1–8, 2008.

[Hach 09] T. Hachisuka and H. Jensen. “Stochastic progressive photon mapping”. In: *SIGGRAPH Asia ’09: ACM SIGGRAPH Asia 2009 papers*, pp. 1–8, 2009.

[Haeb 90] P. Haeberli and K. Akeley. “The accumulation buffer: hardware support for high-quality rendering”. In: *SIGGRAPH ’90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pp. 309–318, 1990.

[Hagr 04] S. Hagreaves. “Deferred Shading, Game Developers Conference”. 2004.

[Havr 01] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001.

[Havr 02] V. Havran and J. Bittner. “On Improving *k*d-trees for Ray Shooting”. *Journal of WSCG*, Vol. 10, No. 1, pp. 209–216, 2002.

[Havr 05] V. Havran, R. Herzog, and H.-P. Seidel. “Fast Final Gathering via Reverse Photon Mapping”. *Computer Graphics Forum (Proceedings of Eurographics 2005)*, Vol. 24, No. 3, pp. 323–333, August 2005.

[Havr 07] V. Havran and J. Bittner. “Ray Tracing with Sparse Boxes”. In: *Spring Conference on Computer Graphics (SCCG 2007)*, pp. 49–54, 2007.

[Havs 07] M. Hašan, F. Pellacini, and K. Bala. “Matrix row-column sampling for the many-light problem”. *ACM Trans. Graph.*, Vol. 26, No. 3, p. 26, 2007.

[Heck 90] P. Heckbert. “Adaptive radiosity textures for bidirectional ray tracing”. In: *SIGGRAPH ’90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pp. 145–154, 1990.

[Heid 82] K. Heidler. *Wirkungsgraduntersuchung zur Solarenergiekonversion mit Fluoreszenzkollektoren*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 1982.

[Hols 89] M. Holschneider, R. Kronland-Martinet, J. Morlet, and P. Tchamitchian. *A real-time algorithm for signal analysis with the help of the wavelet transform*. Springer-Verlag, 1989.

[Hubo 06] E. Hubo, T. Mertens, and P. Bekaert. “The quantized kd-tree: compression of huge point sampled models”. *ACM SIGGRAPH 2006 Sketches*, p. 179, 2006.

[Hunt 06] W. Hunt, W. Mark, and G. Stoll. “Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic”. In: *2006 IEEE Symposium on Interactive Ray Tracing*, Sept. 2006.

[IEEE 85] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute. *IEEE standard for binary floating-point arithmetic*. *ANSI/IEEE Std 754-1985.*, IEEE Computer Society Press, 1985.

[Igeh 99] H. Igehy. “Tracing ray differentials”. In: *SIGGRAPH ’99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 179–186, 1999.

[Inte 09] *Intel®64 and IA-32 Architecture Optimization Reference Manual*. Intel®, 2009.

[Ize 06] T. Ize, I. Wald, C. Robertson, and S. G. Parker. “An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes”. In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 27–55, 2006.

[Ize 08] T. Ize, I. Wald, and S. Parker. “Ray Tracing with the BSP Tree”. In: *Proc. 2008 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 159–166, 2008.

[Jaro 08] W. Jarosz, H. Jensen, and C. Donner. “Advanced global illumination using photon mapping”. In: *SIGGRAPH ’08: ACM SIGGRAPH 2008 classes*, pp. 1–112, ACM, New York, NY, USA, 2008.

[Jens 09] H. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2009.

[Jens 96a] H. Jensen. “Global illumination using photon maps”. In: *Rendering Techniques ’96 (Proc. of the Seventh Eurographics Workshop on Rendering)*, pp. 21–30, 1996.

[Jens 96b] H. Jensen. “Rendering Caustics on Non-Lambertian Surfaces”. *Computer Graphics Forum*, Vol. 16, pp. 116–121, 1996.

[Kaji 86] J. Kajiya. “The rendering equation”. *SIGGRAPH Comput. Graph.*, Vol. 20, No. 4, pp. 143–150, 1986.

[Kalo 09] J. Kalojanov and P. Slusallek. “A parallel algorithm for construction of uniform grids”. In: *HPG ’09: Proceedings of the Conference on High Performance Graphics 2009*, pp. 23–28, ACM, 2009.

[Kalo 86] M. Kalos and P. Whitlock. *Monte Carlo Methods, Volume I: Basics*. J. Wiley & Sons, 1986.

[Kamm 07] R. Kammaje and B. Mora. “A Study of Restricted BSP Trees for Ray Tracing”. In: *RT ’07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pp. 55–62, IEEE Computer Society, 2007.

[Kass 06] M. Kass, A. Lefohn, and J. Owens. “Interactive Depth of Field Using Simulated Diffusion on a GPU”. Tech. Rep., Pixar Animation Studios, 2006.

[Kell 00] A. Keller and I. Wald. “Efficient Importance Sampling Techniques for the Photon Map”. In: *Proceedings of the Vision Modeling and Visualization Conference*, pp. 271–279, 2000.

[Kell 01] A. Keller. “Hierarchical Monte Carlo Image Synthesis”. *Mathematics and Computers in Simulation*, Vol. 55, No. 1-3, pp. 79–92, 2001.

[Kell 97] A. Keller. “Instant Radiosity”. *Proc. of SIGGRAPH ’97*, pp. 49–56, 1997.

[Kell 98] A. Keller. *Quasi-Monte Carlo Methods for Photorealisitic Image Synthesis*. PhD thesis, Universität Kaiserslautern, 1998.

[Kess 08] M. Kesson. “Pixar’s RenderMan”. In: *SIGGRAPH Asia ’08: ACM SIGGRAPH ASIA 2008 courses*, pp. 1–138, 2008.

[Kin 97] J. Kin and K. Choi. “Ray Tracing Triangular Meshes”. *In Western Computer Graphics Symposium*, pp. 43–52, 1997.

[Kirk 91] D. Kirk and J. Arvo. "Improved Ray Tagging For Voxel-Based Ray Tracing". In: J. Arvo, Ed., *Graphics gems II*, pp. 264–266, Academic Press Professional, Inc., 1991.

[Klim 97] K. Klimaszewski and T. Sederberg. "Faster Ray Tracing Using Adaptive Grids". *IEEE Comput. Graph. Appl.*, Vol. 17, No. 1, pp. 42–51, 1997.

[Knut 81] D. Knuth. *The Art of Computer Programming Vol. 2: Seminumerical Algorithms*. Addison Wesley, 1981.

[Kolb 95] C. Kolb, D. Mitchell, and P. Hanrahan. "A realistic camera model for computer graphics". *Proc. of SIGGRAPH '95*, pp. 317–324, 1995.

[Koll 04] T. Kollig and A. Keller. "Illumination in the Presence of Weak Singularities". In: *Proc. Monte Carlo and Quasi-Monte Carlo Methods 2002*, pp. 245–257, Springer, 2004.

[Kont 06] J. Kontkanen, J. Räsänen, and A. Keller. "Irradiance Filtering for Monte Carlo Ray Tracing". In: *Monte Carlo and Quasi-Monte Carlo Methods 2004*, pp. 259–272, Springer Berlin Heidelberg, 2006.

[Kriv 10] J. Křivánek, M. Fajardo, P. Christensen, E. Tabellion, M. Bunnell, D. Larsson, and A. Kaplanyan. "Global Illumination Across Industries". In: *ACM SIGGRAPH 2010 Course Notes*, 2010.

[Lafo 93] E. Lafortune and Y. Willems. "Bi-directional path tracing". In: *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93*, pp. 145–153, 1993.

[Laga 08a] A. Lagae and P. Dutré. "Accelerating Ray Tracing using Constrained Tetrahedralizations". *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, Vol. 27, No. 4, pp. 1303–1312, June 2008.

[Laga 08b] A. Lagae and P. Dutré. "Compact, Fast and Robust Grids for Ray Tracing". *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, Vol. 27, No. 4, pp. 1235–1244, June 2008.

[Lain 07] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, and T. Aila. "Incremental Instant Radiosity for Real-Time Indirect Illumination". In: *Proceedings of Eurographics Symposium on Rendering 2007*, pp. 277–286, Eurographics Association, 2007.

[Laut 07] C. Lauterbach, S.-E. Yoon, and D. Manocha. "Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing". In: *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 19–26, 2007.

[Laut 09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. "Fast BVH Construcion on GPUs". In: *Computer Graphics Forum (Proc. of Eurographics 2009)*, p. to appear, 2009.

[Leht 08] J. Lehtinen, M. Zwicker, E. Turquin, J. Kontkanen, F. Durand, F. X. Sillion, and T. Aila. "A meshless hierarchical representation for light transport". In: *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pp. 1–9, 2008.

[Lext 00] J. Lext, U. Assarsson, and T. Möller. "BART: A benchmark for animated ray tracing". Tech. Rep., Chalmers University of Technology, 2000.

[Lofs 05] M. Löfsted and T. Akenine-Möller. "An Evaluation Framework for Ray-Triangle Intersection Algorithms". *Journal of Graphics, GPU, and Game Tools*, Vol. 10, No. 2, pp. 13–26, 2005.

[Maho 05] J. Mahovsky. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, 2005.

[Mall 89] S. Mallat. "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, pp. 674–693, 1989.

[Mall 98] S. Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 1998.

[Mant 05] R. Mantiuk, S. Daly, K. Myszkowski, and H.-P. Seidel. "Predicting Visible Differences in High Dynamic Range Images - Model and its Calibration". In: B. E. Rogowitz, T. N. Pappas, and S. J. Daly, Eds., *Human Vision and Electronic Imaging X, IS&T/SPIEś 17th Annual Symposium on Electronic Imaging*, pp. 204–214, 2005.

[Marr 02] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, A. Miller, and M. Upton. "Hyper-Threading Technology Architecture and Microarchitecture". *Intel Technology Journal*, Vol. 6, No. 1, pp. 4–15, February 2002.

[Mats 98] M. Matsumoto and T. Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". *ACM Trans. Model. Comput. Simul.*, Vol. 8, No. 1, pp. 3–30, 1998.

[Matu 03] W. Matusik, H. Pfister, M. Brand, and L. McMillan. "A Data-Driven Reflectance Model". *ACM Transactions on Graphics*, Vol. 22, No. 3, pp. 759–769, July 2003.

[McGu 09] M. McGuire and D. Luebke. "Hardware-accelerated global illumination by image space photon mapping". In: *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pp. 77–89, 2009.

[Meht 04] D. Mehta and S. Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004.

[Mitc 87] D. Mitchell. “Generating antialiased images at low sampling densities”. *Computer Graphics (Proc. SIGGRAPH ’87)*, pp. 65–72, 1987.

[Moll 97] T. Möller and B. Trumbore. “Fast, Minimum Storage Ray/Triangle Intersection”. *Journal of Graphics Tools*, Vol. 2, No. 1, pp. 21–28, 1997.

[More 01] H. Moreton. “Watertight tessellation using forward differencing”. In: *HWWS ’01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pp. 25–32, 2001.

[Murt 97] F. Murtagh. “Multiscale Transform Methods in Data Analysis”. 1997.

[Mysz 98] K. Myszkowski. “The visible differences predictor: applications to global illumination problems”. In: *Rendering Techniques ’98 (Proc. of the Sixth Eurographics Workshop on Rendering)*, pp. 233–236, 1998.

[Nich 09] G. Nichols and C. Wyman. “Multiresolution splatting for indirect illumination”. In: *I3D ’09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 83–90, 2009.

[Nied 92] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, Philadelphia, 1992.

[Opti 09] *Optical Glass - Data Sheets*. Schott AG, 2009.

[Over 08] R. Overbeck, R. Ramamoorthi, and W. Mark. “Large ray packets for real-time Whitted ray tracing”. In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pp. 41 –48, 2008.

[Over 09] R. Overbeck, C. Donner, and R. Ramamoorthi. “Adaptive wavelet rendering”. In: *SIGGRAPH Asia ’09: ACM SIGGRAPH Asia 2009 papers*, pp. 1–12, 2009.

[Pain 89] J. Painter and K. Sloan. “Antialiased ray tracing by adaptive progressive refinement”. *Computer Graphics (Proc. SIGGRAPH ’89)*, pp. 281–288, 1989.

[Pari 09] S. Paris and F. Durand. “A Fast Approximation of the Bilateral Filter Using a Signal Processing Approach”. *International Journal of Computer Vision*, Vol. 81, No. 1, pp. 24–52, 2009.

[Park 99] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. “Interactive Ray Tracing for Volume Visualization”. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 5, No. 3, pp. 238–250, 1999.

[Pedr 06] F. Pedrotti and L. Pedrotti. *Introduction to Optics*. Pearson Education (US), 2006.

[Pell 05] F. Pellacini, K. Vidimče, A. Lefohn, A. Mohr, M. Leone, and J. Warren. "Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography". *ACM Trans. Graph.*, Vol. 24, No. 3, pp. 464–470, 2005.

[Pete 07] M. Peters, J. Goldschmidt, P. Loeper, A. Gombert, and G. Willeke. "Application of Photonic Structures on Fluorescent Concentrators". In: *Proceedings of the 22nd European Photovoltaic Solar Energy Conference*, 2007.

[Pets 04] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama. "Digital photography with flash and no-flash image pairs". *ACM Trans. Graph.*, Vol. 23, No. 3, pp. 664–672, 2004.

[Phar 04] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 2004.

[Pixa 05] Pixar. "The RenderMan Interface". 2005.

[Pree 99] A. Preetham, P. Shirley, and B. Smits. "A practical analytic model for daylight". In: *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 91–100, 1999.

[Pres 92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.

[Quin 03] M. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[Raab 06] M. Raab, D. Seibert, and A. Keller. "Unbiased Global Illumination with Participating Media". In: *Proc. Monte Carlo and Quasi-Monte Carlo Methods 2006*, pp. 591–605, Springer, 2006.

[Raga 07] J. Ragan-Kelley, C. Kilpatrick, B. W. Smith, D. Epps, P. Green, C. Hery, and F. Durand. "The lightspeed automatic interactive lighting preview system". In: *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, p. 25, ACM, 2007.

[Rama 99] M. Ramasubramanian, S. Pattanaik, and D. Greenberg. "A perceptually based physical error metric for realistic image synthesis". *ACM Transactions on Graphics (Proc. SIGGRAPH '99)*, pp. 73–82, 1999.

[Ray 02] S. Ray. *Applied Photographic Optics*. Focal Press, 3rd Ed., 2002.

[Rein 00] E. Reinhard, B. E. Smits, and C. Hansen. "Dynamic Acceleration Structures for Interactive Ray Tracing". In: *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pp. 299–306, 2000.

[Rein 02] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. “Photographic tone reproduction for digital images”. In: *SIGGRAPH ’02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 267–276, ACM, 2002.

[Resh 05] A. Reshetov, A. Soupikov, and J. Hurley. “Multi-Level Ray Tracing Algorithm”. *ACM Transactions on Graphics (Proc. SIGGRAPH 2005)*, pp. 1176–1185, 2005.

[Resh 07] A. Reshetov. “Faster Ray Packets - Triangle Intersection through Vertex Culling”. In: *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 105–112, 2007.

[Riga 03] J. Rigau, M. Feixas, and M. Sbert. “Refinement Criteria for Global Illumination using Convex Functions”. In: *Compositional Data Analysis Workshop*, 2003.

[Rits 08] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz. “Imperfect shadow maps for efficient computation of indirect illumination”. In: *SIGGRAPH Asia ’08: ACM SIGGRAPH Asia 2008 papers*, pp. 1–8, 2008.

[Rits 09] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, and C. Dachsbacher. “Micro-rendering for scalable, parallel final gathering”. In: *SIGGRAPH Asia ’09: ACM SIGGRAPH Asia 2009 papers*, pp. 1–8, 2009.

[Sait 08] M. Saito and M. Matsumoto. “SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator”. In: A. Keller, S. Heinrich, and H. Niederreiter, Eds., *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pp. 607–622, Springer Berlin Heidelberg, 2008.

[Same 89] H. Samet. “Implementing ray tracing with octrees and neighbor finding”. *Computers And Graphics*, Vol. 13, pp. 445–460, 1989.

[Schl 94] C. Schlick. “An inexpensive BRDF model for physically-based rendering”. *Computer Graphics Forum*, Vol. 13, pp. 233–246, 1994.

[Scho 03] D. Schouten, X. Tian, A. Bik, and M. Girkar. “Inside the Intel compiler”. *Linux Journal*, Vol. 2003, No. 106, 2003.

[Schu 07] A. Schüler, A. Kostro, C. Galande, M. Valle del Olmo, E. de Chambrier, and B.Huriet. “Principles of Monte-Carlo Ray-Tracing Simulations of Quantum Dot Solar Concentrators”. In: *Proceedings of the ISES solar world congress 2007*, 2007.

[Sedg 92] R. Sedgewick. *Algorithms in C++*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[Seet 04] H. Seetzen, W. Heidrich, W. Stuerzlinger, G. Ward, L. Whitehead, M. Trentacoste, A. Ghosh, and A. Vorozcovs. “High dynamic range display systems”. *ACM Trans. Graph.*, Vol. 23, No. 3, pp. 760–768, 2004.

[Sego 06] B. Segovia, J.-C. Iehl, R. Mitanchey, and B. Péroche. “Bidirectional instant radiosity”. In: *Proceedings of the 17th Eurographics Workshop on Rendering*, 2006.

[Sego 07] B. Segovia, J.-C. Iehl, and B. Péroche. “Metropolis Instant Radiosity”. *Computer Graphics Forum*, Vol. 26, No. 3, pp. 425–434, 2007.

[Seid 57] P. von Seidel. “Über die Theorie der Fehler, mit welchen die durch optische Instrumente gesehenen Bilder behaftet sind, und über die mathematischen Bedingungen ihrer Aufhebung”. *Abhandlungen der naturwissenschaftlich-technischen Commission bei der Königlichen Bayerischen Akademie der Wissenschaften in München*, Vol. 1, pp. 227–267, 1857.

[Seil 08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. “Larrabee: a many-core x86 architecture for visual computing”. *ACM Transactions on Graphics (Proc. SIGGRAPH 2008)*, pp. 1–15, 2008.

[Shev 07] M. Shevtsov, A. Soupikov, and A. Kapustin. “Highly Parallel Fast *k*d-tree Construction for Interactive Ray Tracing of Dynamic Scenes”. In: *Computer Graphics Forum (Proc. Eurographics 2007)*, pp. 395–404, 2007.

[Shir 00] P. Shirley. *Realistic Ray Tracing*. AK Peters, Ltd., 2000.

[Shir 02] P. Shirley. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., 2002.

[Shir 06] P. Shirley. “State of the art in interactive ray tracing”. In: *ACM SIGGRAPH 2006 Courses*, 2006.

[Sloa 02] P.-P. Sloan, J. Kautz, and J. Snyder. “Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments”. In: *SIGGRAPH ’02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 527–536, 2002.

[Sloa 07] P.-P. Sloan, N. Govindaraju, D. Nowrouzezahrai, and J. Snyder. “Image-Based Proxy Accumulation for Real-Time Soft Global Illumination”. In: *PG ’07: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, 2007.

[Smit 05] W. Smith. *Modern Lens Design*. McGraw-Hill, 2nd Ed., 2005.

[Smit 07] W. Smith. *Modern Optical Engineering*. McGraw-Hill Professional, 4th Ed., 2007.

[Smit 81] J. Smith. “A study of branch prediction strategies”. In: *ISCA ’81: Proceedings of the 8th annual symposium on Computer Architecture*, pp. 135–148, 1981.

[Smit 98] B. Smits. “Efficiency Issues for Ray Tracing”. *Journal of Graphics Tools*, Vol. 3, No. 2, pp. 1–14, 1998.

[Stei 09] B. Steinert, H. Dammertz, J. Hanika, and H. Lensch. “General Spectral Camera Lens Simulation”. 2009. Poster at Pacific Graphics, Jeju, Korea.

[Step 06] A. Stephens, S. Boulos, J. Bigler, I. Wald, and S. G. Parker. “An Application of Scalable Massive Model Interaction using Shared Memory Systems”. In: *Proceedings of the 2006 Eurographics Symposium on Parallel Graphics and Visualization*, pp. 19–26, 2006.

[Sund 04] V. Sundstedt, A. Chalmers, K. Cater, and K. Debattista. “Top-Down Visual Attention for Efficient Rendering of Task Related Scenes”. In: *In Vision, Modeling and Visualization*, pp. 209–216, 2004.

[Sund 07] V. Sundstedt, D. Gutierrez, O. Anson, F. Banterle, and A. Chalmers. “Perceptual rendering of participating media”. *ACM Trans. Appl. Percept.*, Vol. 4, No. 3, p. 15, 2007.

[Suyk 00] F. Suykens and Y. Willems. “Adaptive Filtering for Progressive Monte Carlo Image Rendering”. In: *Eighth International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG 2000)*, 2000.

[Suyk 01] F. Suykens and Y. D. Willems. “Path Differentials and Applications”. In: *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pp. 257–268, 2001.

[Swel 97] W. Sweldens. “The lifting scheme: A construction of second generation wavelets”. *SIAM J. Math. Anal.*, Vol. 29, No. 2, pp. 511–546, 1997.

[Szir 02] L. Szirmay-Kalos, V. Havran, B. Balázs, and L. Szćsi. “On the efficiency of ray-shooting acceleration schemes”. In: *Spring Conference on Computer Graphics (SCCG 2002)*, pp. 97–106, 2002.

[Szir 98] L. Szirmay-Kalos and G. Márton. “Worst-Case Versus Average Case Complexity of Ray-Shooting”. *Journal of Computing*, Vol. 61, No. 2, pp. 103–133, 1998.

[Tabe 04] E. Tabellion and A. Lamorlette. “An approximate global illumination system for computer generated films”. In: *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, pp. 469–476, 2004.

[Toma 98] C. Tomasi and R. Manduchi. “Bilateral Filtering for Gray and Color Images”. In: *ICCV ’98: Proceedings of the Sixth International Conference on Computer Vision*, p. 839, IEEE Computer Society, Washington, DC, USA, 1998.

[Tsak 09] J. Tsakok. “Faster incoherent rays: Multi-BVH ray stream tracing”. In: *HPG ’09: Proceedings of the Conference on High Performance Graphics 2009*, pp. 151–158, 2009.

[Tull 95] D. Tullsen, S. Eggers, and H. Levy. “Simultaneous Multithreading: Maximizing On-Chip Parallelism”. In: *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, 1995.

[Uytt 98] G. Uytterhoeven and A. Bultheel. “The Red-Black wavelet transform”. In: *Signal Processing Symposium (IEEE Benelux)*, pp. 191–194, 1998.

[Veac 94] E. Veach and L. Guibas. “Bidirectional estimators for light transport”. In: *Rendering Techniques ’94 (Proc. of the Fifth Eurographics Workshop on Rendering)*, pp. 147 – 161, 1994.

[Veac 95] E. Veach and L. Guibas. “Optimally Compining Sampling Techniques for Monte Carlo Rendering”. *Proc. of SIGGRAPH ’95*, pp. 419–428, 1995.

[Veac 97a] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997.

[Veac 97b] E. Veach and L. Guibas. “Metropolis Light Transport”. *Proc. of SIGGRAPH ’97*, pp. 65–76, 1997.

[Wach 06] C. Wächter and A. Keller. “Instant Ray Tracing: The Bounding Interval Hierarchy”. In: *Rendering Techniques 2006 (Proc. 17th Eurographics Symposium on Rendering)*, pp. 139–149, 2006.

[Wach 07] C. Wächter and A. Keller. “Terminating Spatial Partition Hierarchies by A Priory Bounding Memory”. In: *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 41–46, 2007.

[Wald 01] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. “Interactive Rendering with Coherent Ray Tracing”. In: *Computer Graphics Forum (Proc. Eurographics 2001)*, pp. 153–164, 2001.

[Wald 02a] I. Wald, C. Benthin, and P. Slusallek. “OpenRT – A Flexible and Scalable Rendering Engine for Interactive 3D Graphics”. Tech. Rep. TR-2002-01, Saarland University, 2002.

[Wald 02b] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. “Interactive global illumination using fast ray tracing”. In: *EGRW ’02: Proceedings of the 13th Eurographics workshop on Rendering*, pp. 15–24, 2002.

[Wald 04a] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[Wald 04b] I. Wald, J. Günther, and P. Slusallek. "Balancing Considered Harmful - Faster Photon Mapping using the Voxel Volume Heuristic -". In: *Computer Graphics Forum*, p. 2004, 2004.

[Wald 06a] I. Wald and V. Havran. "On building fast *k*d-trees for Ray Tracing, and on doing that in O(N log N)". In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 18–20, 2006.

[Wald 06b] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. "Ray tracing animated scenes using coherent grid traversal". In: *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pp. 485–493, 2006.

[Wald 07a] I. Wald. "On fast Construction of SAH based Bounding Volume Hierarchies". In: *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 33–40, 2007.

[Wald 07b] I. Wald, S. Boulos, and P. Shirley. "Ray tracing deformable scenes using dynamic bounding volume hierarchies". *ACM Trans. Graph.*, Vol. 26, No. 1, p. 6, 2007.

[Wald 07c] I. Wald, W. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. Parker, and P. Shirley. "State of the Art in Ray Tracing Animated Scenes". In: D. Schmalstieg and J. Bittner, Eds., *STAR Proceedings of Eurographics 2007*, pp. 89–116, The Eurographics Association, Sep. 2007.

[Wald 08] I. Wald, C. Benthin, and S. Boulos. "Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-branching BVHs". In: *Proc. 2008 IEEE/EG Symposium on Interactive Ray Tracing*, pp. 49–57, 2008.

[Walt 05] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. "Lightcuts: a scalable approach to illumination". In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pp. 1098–1107, 2005.

[Walt 06] B. Walter, A. Arbree, K. Bala, and D. P. Greenberg. "Multidimensional lightcuts". In: *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pp. 1081–1088, 2006.

[Wang 09] R. Wang, R. Wang, K. Zhou, M. Pan, and H. Bao. "An efficient GPU-based approach for interactive global illumination". In: *SIGGRAPH '09: ACM SIGGRAPH 2009 papers*, pp. 1–8, 2009.

[Ward 88] G. Ward, F. M. Rubinstein, and R. D. Clear. "A ray tracing solution for diffuse interreflection". *SIGGRAPH Comput. Graph.*, Vol. 22, pp. 85–92, June 1988.

[Ward 92a] G. Ward. “Measuring and modeling anisotropic reflection”. *SIGGRAPH Comput. Graph.*, Vol. 26, No. 2, pp. 265–272, 1992.

[Ward 92b] G. Ward and P. Heckbert. “Irradiance Gradients”. In: *3rd Eurographics Workshop on Rendering*, pp. 85–98, 1992.

[Warr 02] H. Warren. *Hacker’s Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[Whit 80] T. Whitted. “An improved illumination model for shaded display”. *Communications of the ACM*, Vol. 23, No. 6, pp. 343–349, June 1980.

[Will 05] A. Williams, S. Barrus, R. Morley, and P. Shirley. “An Efficient and Robust Ray-Box Intersection Algorithm”. *Journal of Graphics Tools*, Vol. 10, No. 1, pp. 49–54, 2005.

[Yee 01] H. Yee, S. Pattanaik, and D. Greenberg. “Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments”. In: *In ACM Transactions on Graphics*, pp. 39–65, 2001.

[Yoon 05] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. “Cache-oblivious mesh layouts”. *ACM Transactions on Graphics (Proc. SIGGRAPH 2005)*, pp. 886–893, 2005.

[Yoon 06] S.-E. Yoon and D. Manocha. “Cache-Efficient Layouts of Bounding Volume Hierarchies”. In: *Computer Graphics Forum (Proc. of Eurographics 2006)*, pp. 507–516, 2006.

[Zast 81] A. Zastrow. *Physikalische Analyse der Energieverlustmechanismen im Fluoreszenzkollektor*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 1981.