Real-time Global Illumination by Simulating Photon Mapping

Bent Dalgaard Larsen

Kongens Lyngby 2004
IMM-PHD-2004-130

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Abstract

This thesis introduces a new method for simulating photon mapping in real-time. The method uses a variety of both CPU and GPU based algorithms for speeding up the different elements in global illumination. The idea behind the method is to calculate each illumination element individually in a progressive and efficient manner. This has been done by analyzing the photon mapping method and by selecting efficient methods, either CPU based or GPU based, which replaces the original photon mapping algorithms. We have chosen to focus on the indirect illumination and the caustics.

In our method we first divide the photon map into several photon maps in order to make local updates possible. Then indirect illumination is added using light maps that are selectively updated by using selective photon tracing on the CPU. The final gathering step is calculated by using fragment programs and GPU based mipmapping. Caustics are calculated by using photon tracing on the CPU and the filtering which is performed on the GPU. Direct illumination is calculated by using shading on the GPU.

We achieve real-time frame rates for simple scenes with up to 133.000 polygons. The scenes include standard methods for reflection and refraction and hard shadows. Furthermore, the scenes include our methods for progressively updated caustics and progressively updated indirect illumination. We have compared the image quality of our method to the standard photon mapping method and the results are very similar.

Resumé

Denne afhandling introducerer en ny metode til at simulere photon mapping i real-tid. Metoden benytter både CPU og GPU baserede algoritmer for at øge hastigheden for udregningen af de forskellige elementer der indgår i global illumination. Idéen bag metoden er at udregne hvert enkelt bidrag til den globale illuminations løsning individuelt og på en progressiv og effektiv måde. Dette er opnået ved at analysere photon mapping metoden og for hvert skridt i metoden er der udvalgt en effektiv algoritme, enten baseret på CPU'en eller GPU'en, til at erstatte den originale photon mapping algoritme. Vi har valgt hovedsageligt at fokusere på indirekte belysning og kaustikker.

Vores methode indebærer at photon mappet først bliver inddelt i flere photon maps for at gøre det muligt at lave lokale opdateringer. Indirekte belysning bliver tilføjet vha. light maps som selektivt bliver opdateret vha. selektiv photon tracing på CPU'en. Final gather bliver udregnet vha. fragment programmer og GPU baseret mipmapping. Kaustikker bliver udregnet vha. photon tracing på CPU'en og filtrering på GPU'en. Den direkte belysning bliver udregnet vha. shading på GPU'en.

Vi har opnåset real-tids billedeopdatering for simple 3D scener med op til 133.000 polygoner. Scenerne indkluderer standard metoder for reflektioner, refraktioner og hårde skygger. Yderligerer bliver den indirekte belysning og kaustikkerne opdateret progresivt. Vi har sammenlignet billedekvaliteten som opnåes med vores method med reference billeder som er udregnet vha. standard photon mapping og resultaterne er meget ens.

Contents

Preface

This thesis has been produced at the Image Analysis and Computer Graphics Group at Informatics and Mathematical Modelling (IMM) and submitted to the Technical University of Denmark (DTU), in partial fulfillment of the requirements for the degree of Doctor of Philosophy, Ph.D., in applied mathematics.

The working title of the project is "Collaborative Multi-user Virtual Environments". One primary research topic in this field is to increase the collaborative aspects in multi-user environments. Another primary research topic is to improve the rendering speed and the image quality of 3D scenes. The research performed during the Ph.D. study cover these research topics and a number of projects that focus on specific problems have been carried out. One of the projects is global illumination for real-time application which has become the main topic of this thesis. The projects that have been carried out but did not fit satisfactorily into this thesis are the following:

In [77] we demonstrate a multi-user collaborative 3D application in which it is possible to construct and modify a 3D scene. We use Lego bricks as an example. It is possible to interact with the 3D world from both a standard PC and from a cellular phone. The project is titled: "Using Cellular Phones to Interact with Virtual Environments", and was presented as a technical sketch at the SIGGRAPH Conference in 2002. This is the second version of this application. The first was accessible through a web-browser and was based on VRML and Java.

Another project is real-time terrain rendering. In this project we optimize the rendering of large terrains. Our particular focus is to avoid "popping" when switching between Level of Details (LOD) in a manner that takes advantage of

modern graphics hardware. The project is titled: "Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail" ([79]). It was presented at the WSCG Conference in 2003 and published in the Journal of WSCG 2003.

A third project focuses on improving the image quality of real-time soft shadows. The penumbra region is calculated accurately by using two sets of shadow volumes. The rendering utilizes per pixel operations, which are available on modern graphics hardware, for calculating the penumbra regions. The project is titled: "Boundary Correct Real-Time Soft Shadows" [63]. It was presented at the Computer Graphics International 2004 Conference.

This thesis is mainly based on the following work: "Optimizing Photon Mapping Using Multiple Photon Maps for Irradiance Estimates" ([78]) which was presented at the WSCG Conference in 2003 and "Simulating Photon Mapping for Real-time Applications" ([80]) which was presented the Eurographics Symposium on Rendering 2004. Some of the results in this thesis are currently not published.

In order to read this thesis a prior knowledge of computer graphics is necessary.

Kgs. Lyngby, September 2004

Bent Dalgaard Larsen

Acknowledgement

When looking back at the years I have spend on my Ph.D. study, many people have had a great influence on me. First, I would like to thank my advisor Niels Jørgen Christensen for encouraging me to start my Ph.D. study and for always supporting me during the study. His door has always been open for a good discussion about computer graphics.

I want to thank Andreas Bærentzen for co-advising me during the study both with regard to programming and graphics and thank you Kasper Høy Nielsen for opening my eyes to the area of global illumination and for many interesting discussions.

I want to thank Henrik Wann Jensen for supporting my stay in San Diego at UCSD and for many insightful talks. I also want to thank the people in the Pixel Lab at UCSD for making my stay pleasant.

I want to thank Anthony Steed for supporting my stay in London at UCL. I also want to thank the people in the Computer Graphics and Virtual Environment Group who made my stay pleasant.

I would also like to thank the following people whom I have worked with on projects during my study: Bjarke Jacobsen, Kim Steen Petersen and Michael Grønager. I would also like to thank all the students whom I have been in contact with in the courses "Computer Graphics" and "Virtual Reality" and all the students who I have supervised during the years. To mention a few that have had an influence on my Ph.D. I would like to thank Peter Jensen, Martin Valvik, Jesper Sørensen and Jeppe Eliot Revall Frisvad.

I would like to thank all the people in "The Computer Graphics (and Image Analysis) Lunch Club" for all the insightful and not so insightful discussions and for making IMM a truly pleasant place to be.

I would like to thank Janne Pia Østergaard for correcting a lot of the grammatical errors. The remaining errors are all mine.

Finally, I would like to thank Marianne for supporting me while writing this thesis.

Part I

Background

Chapter 1

Introduction

High visual realism has many important application areas. These areas include applications such as games, virtual walk-throughs and 3D simulations. Visual realism is very important in these applications but an even more important property of these applications is that the images have to be rendered in real-time. In the past it was necessary to choose between either high visual realism or real-time frame rates. Currently a uniting between these two areas is taking place. Many of the same techniques are used both for real-time rendering and when creating high quality images. This area is a very active area of research and in the following we will take a closer look at the some of the achievements.

The illumination in an image does not have to be physically correct, although the more physically correct the images are, the better. In particular this is true for interior illumination. Calculating physically correct images is usually a very challenging task both computationally and mathematically.

In the next sections we will take a closer look at how one calculates the illumination in real-time applications.

1.1 Global and Local Illumination

Local illumination is when a surface is shaded only by using the properties of the surface and the light. The structure of the rest of the scene is not taken into account.

Global illumination is when a surface is shaded using the properties of the surface, the light and all light contributions to this surface from all other surfaces of the scene. Adding global illumination improves visual realism compared to only using the local illumination. Although global illumination is a very important effect, it is mathematically difficult and computationally hard to calculate accurately. The global illumination contributions to a sample point can be divided into a number of individual contributions.

In Figure 1.1 the different elements of calculating global illumination are depicted. Each of the elements will be described in more detail in the following. One important property to note is that each of the contributions are independent. This is a very important when calculating the illumination, as each of the calculations can be performed individually and finally added together.

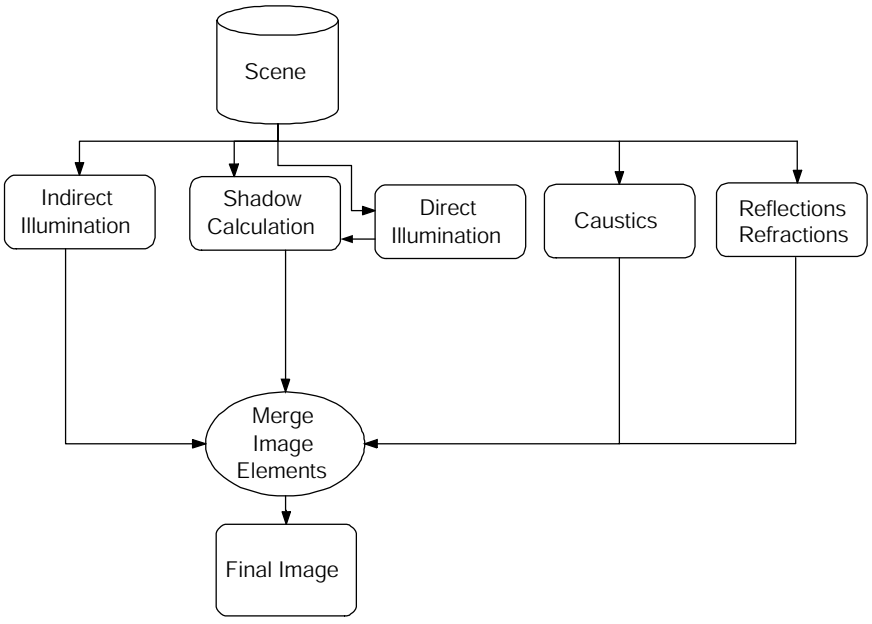

Figure 1.1: Elements in global illumination

The physically correct way to calculate an image would be to simulate the pro-

cess of photons being emitted from a light source using the physical properties of the light and then simulating the interaction between the atoms of the surfaces and the photons.

The energy of a green photon is approximately $3.90e-19J$. The number of photons that would be necessary to distribute from a 60 Watt bulb would therefore be approximately $1.54e20$ per second. Today it is possible to trace approximately $1e6$ photons per second in a scene consisting of simple polygons. This means that computers need to get approximately $1e14$ times faster than they are today to trace this scene. Assuming Moors law will be true for many years to come it will be possible to trace this number of photons in approximately 100 years. Unfortunately it is substantially more complicated to simulate photon interaction with a more realistic scene containing fabrics, organic structures and animals. This means that it will take significantly longer to reach the computational power necessary to handle such a scene. This leaves us with two options. Either we can forget about calculating computer graphics images or we can try to find approximations for calculating these images instead. In this thesis we have chosen the latter approach.

1.2 Rendering Images

Currently two primary methods exist for rendering a 3D model, namely rasterization and ray tracing. Nevertheless, one of the most popular methods used for rendering movies is the Reyes architecture [30]. Reyes is a method for splitting render primitives (e.g. triangles, NURBS and subdivision surfaces) into elements that are smaller than the size of a pixel and then scanline converting these micropolygons. This is the method used in Pixar's official implementation of the RenderMan standard PRMan ([123], [52], [6]). Although this method is very popular for movie production, it seems that the method is currently not relevant for real-time graphics [96]. Furthermore, movie production is primarily based on high level primitives like subdivision surfaces and NURBS, whereas real-time rendering is almost exclusively utilizes triangles (both with regard to rasterization and real-time ray tracing).

Currently there is an ongoing battle between rasterization and ray tracing with regard to which of the methods that is most appropriate for real-time graphics [2]. Traditionally ray tracing has only been considered useful for non-interactive rendering of very realistic images. Rasterization, on the other hand, has been considered most appropriate for fast real-time applications with less photorealistic requirements. But today ray tracing is becoming faster while the image quality of rasterization is constantly being improved. The two methods have so

to speak entered into each other's domains ([132], [102]).

1.2.1 Ray Tracing

Ray tracing was first presented by [7] and later improved by [143]. It turned out that many effects were straight forward to calculate by using ray tracing. In particular, shadows are simple, as they only require an extra ray for each light source per pixel. In [31] distributed ray tracing was introduced and it was shown how to easily integrate over many dimensions simultaneously in order to achieve effects such as soft shadows, motion blur and depth of field.

The most time consuming part of ray tracing is the ray-object intersection calculations (although shading is also becoming a time consuming part ([131])). One method that can be used to optimize the ray tracing process is to cache the results of previous frames and reuse these values. Sending rays to the most important areas in the image and then interpolate the rest is another optimization method. Tricks like these will optimize the ray tracing process but in many circumstances minor errors will occur. As a result, it is therefore more desirable to optimize the general ray tracing process ([125]).

In computer graphics scenes, one of the most frequently used primitives is the triangle, and the algorithm to optimize is therefore the ray-triangle intersection algorithm. This algorithm has been optimized heavily ([90], [5], [41], [111], [125]).

Completely avoiding the ray-triangle intersection calculation for a ray that does not intersect the triangle in any case is of course an even better alternative. This can be accomplished by storing the triangles in a spatial data structure ([55] [85] [75]). In [55] a vast number of spatial structures are examined, and it is argued that the optimal structure in many cases is the kd-tree (sometimes also called a BSP-tree). But generally the optimal spatial structure is dependent on the nature of the scene. Consequently, no single data-structure is the fastest in all circumstances.

The very fast ray tracers depend on a static spatial structure.

Better spatial data-structures usually demands longer pre-processing time. There seems to be a tradeoff between rendering time and preprocessing. Consequently, dynamic scenes are inherently slow to render as the spatial data structure constantly needs to be rebuild. For this reason optimizing ray tracing for dynamic scenes is an active area of research ([105],[132], [127], [82], [81]).

Recently, ray tracing has been used for real-time rendering e.g. in [97]. In [132] it is demonstrated that in static scenes with a very high polygon count and given the special circumstance that all polygons are visible at the same time, ray tracing may even be faster than rasterization. The systems developed by the group lead by Slusallek are all based on clusters of PCs ([127], [126], [131] and [129]). While the system developed by Parker et al. is based on a supercomputer ([97]).

In [133] and [134] the Render Cache approach is described. In Render Cache the ray tracing process is running in a separate thread than the display process. In this way the frame rate is interactive while the image is progressively updated. When the viewpoint is changed, the points in the image are reprojected by using the new camera position. This will create image artifacts and the occlusion may be erroneous while the image is updated again. Nevertheless, the system is at all time interactive.

For some time, the people of the demo scene ([40]) have been creating real-time ray tracers running on standard PC hardware. It seems that many optimized ray tracers with advanced features are currently being created by people of the demo scene. Unfortunately information about their implementations is scarce.

Recently, it has been demonstrated that ray tracing can be implemented on modern graphics hardware. In [99] it is proposed how ray tracing may be performed on future GPUs (Graphics Processing Unit), while in [19] the GPU is used as a fast ray-triangle intersection engine (for a good discussion of these approaches see [125]).

The fastest software ray tracers are those that build a clever optimization structure and only perform few ray-triangle intersections. Therefore, the ray tracers implemented on the graphics-card have not yet been able to exceed the performance achieved by using just software ray tracers. This is because it has not been possible, so far, to implement optimal spatial data-structures on the GPU ([125]). However this might change in the future. Furthermore, graphics hardware accelerated ray tracing implementations are in some ways limited by the speed of graphics hardware. Currently the speed of graphics hardware is increasing much faster than the speed of CPUs ([3]). Therefore, it will be increasingly more advantageous to use graphics hardware implementations. Despite that, it still has to be proven that GPUs are the best option for ray tracing. Ray tracing has also been implemented on the FPGA architecture ([109]). An FPGA is programmable and more flexible in its architecture than the GPU. This FPGA implementation is very fast although it only runs at a low clock frequency (90 MHz). Converting this implementation to another architecture than the FPGA would further increase the speed dramatically.

1.2.2 Rasterization

In the eighties and beginning of the nineties the development was driven by very expensive Silicon Graphics systems, but later on the evolution was driven by graphics cards for the PC.

The improvement in real-time 3D graphics has mainly been concentrated on three areas: performance, features and quality ([3]). The performance is measured in triangles per second and in processed pixel fragments per second. However, at present bandwidth is one of the most important parameters. The features are the visual effects which it is possible to simulate.

1.2.2.1 The Geometry Pipeline

The heart of rasterization is the geometry pipeline. The geometry pipeline consists of a number of steps. As input to the geometry pipeline are the geometry and some parameters that define how this geometry should be processed. In the last step the geometry is rasterized, which means that the individual points are connected to display the desired geometry. Each value is usually called a *fragment*. When a fragment is displayed on the screen, it is termed a *pixel*. If the fragment is used in a texture it is termed a *texel*. An overview of the steps is given in Figure 1.2. A much more thorough explanation of this process is given in [5].

Over time more and more features have been added to the geometry pipeline. Many features have been enabled and disabled by setting on/off flags in the API's and by creating special variables for these features. This has been done in order to render more photo realistic images. But all these extra features make it very complicated to develop the graphics cards because more and more combinations of these flags exist. Many of the individual combinations need to be handled individually and consequently a combinatorial explosion has taken place. Furthermore, developers always want to have new very specific features in order to implement their new algorithm on the graphics card. Many of these options only have limited use and would therefore never be implemented on graphics cards.

In order to solve the problems of the fixed pipeline the programmable geometry pipeline was introduced [83]. Currently two of the stages in the geometry pipeline have been made programmable and it is very likely that more parts will be made programmable in the future. The stages that are currently programmable are the vertex transformation stage and the fragment processing

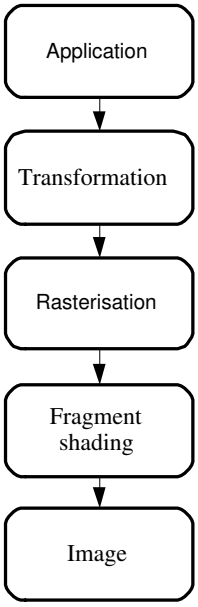

Figure 1.2: Fixed Function Pipeline

stage. Many different names have been given to these two stages. The stage where the vertices are transformed to camera space have been named *Vertex Shaders* and *Vertex Programs*. In the following we will name these programs *Vertex Programs*. The stage where fragments are shaded have been named *Pixel Shaders*, *Pixel Programs*, *Fragment Shaders* and *Fragment Programs*. In the following we will name these programs *Fragment Programs*. It seems that the naming depends on the API that is being used, and the vendor which is writing the documentation.

The parts in the graphics pipeline which have been substituted with programmable elements can be seen in Figure 1.3.

Both vertex and fragment programs are created by using assembly instructions. In general each assembly instruction, whether it is a simple multiplication or a square root, takes one clock cycle.

More and more functionality for 3D calculations has been moved from the CPU to the graphics card. Often the processor on the graphics card is now termed GPU as it has become as powerful and complex as the CPU. Nevertheless, the nature of the CPU is quite different from that of the CPU as the CPU has been created to execute any type of program while the GPU has been created pri-

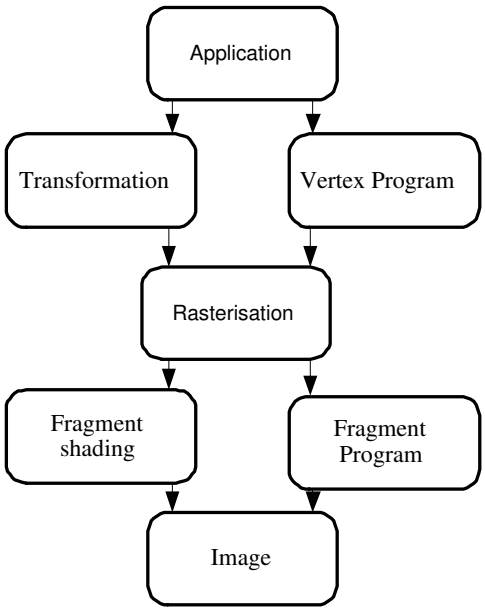

Figure 1.3: Programmable Pipeline

marily for rasterizing 3D geometry. Often the GPU is called a stream processor because it works on streams of data. The GPU was developed to process graphics although it can also perform general computations ([18]). Accordingly the CPU and GPU each excel in their own area and they are not directly comparable with respect to functionality and performance.

Previously, coding had to be done directly in the assembly language but these days one sees a shift to high level languages. One of the more popular high-level languages is Cg ([94] [86]). Cg is a language based on C (C for graphics) but with some modifications in order to make it more appropriate for graphics cards. The creation of Cg was evidently inspired by the Renderman standard. Furthermore, Cg has been constructed in such a way that an optimized compiler creates code that is as fast as handwritten assembly code [102]. Currently, more shading languages are introduced but they all resemble Cg very closely. These are OpenGL Shading Language ([106], [73]) which is a vendor independent OpenGL shading language and HLSL which is an extension to Microsoft's DirectX. High level languages have many advantages compared to assembly languages. E.g. high level languages are faster to write and debug, and they are often compilable on several platforms. In this text we will only use Cg as this language is much more readable than the assembly language. For good overviews of the different shading languages and the evolution of real-time shading languages see [106]

and [95].

1.2.2.2 Reflections and Refractions

Reflections and refractions are straightforward to implement by using ray tracing. Implementing these effects by using rasterization is on the other hand quite difficult. Planar reflections which uses the stencil buffer are described in [34], [74] and [89], while planar reflections which use texture mapping are described in [89]. Currently one of the most advanced examples of what is possible with regard to reflections is presented in [92] where multiple reflections on both curved and flat surfaces is demonstrated. Refractions on the other hand can be approximated by using programmable hardware [102] and although it may be possible to produce visually convincing images, the results are not 100% correct.

1.3 Ray Tracing versus Rasterization Discussion

Rasterization is easy to implement on graphics hardware as all that is needed is a lot of hardware optimized vector arithmetic. Furthermore a pipeline is also very suitable for hardware implementation since dedicated hardware can be made for each step in the pipeline. Ray tracing, on the other hand, does not naturally fit into a pipeline. Since the ray tracing algorithm traverses the entire data-structure for each pixel, it is necessary to have the 3D scene in graphics hardware.

Hence there are a vast number of hardware accelerated graphics card on the market for rasterization but few for ray tracing, although hardware accelerated ray tracing is currently an active area of research ([108], [109]).

Even so ray tracing has proven itself to be better than rasterization in special circumstances ([132]), and only considering either ray tracing or rasterization for real-time graphics will not be viable. Whether one should use rasterization or ray tracing in a real-time application depends on the nature or the application. Despite that, rasterization is at present the preferred real-time rendering method.

As seen in Figure 1.1 calculating the direct illumination is a task separated from the calculation of caustics and indirect illumination. Whether to use ray tracing or rasterization for the direct illumination may therefore be independent from choosing methods used for other effects. However, in practice some of the effects

	Ray tracing	Rasterization
Complexity	$O(\log n)$	$O(n)$
Constant	High	Low
Flat reflections	Yes	Yes
Curved reflections	Yes	No/(Yes)
Arbitrary reflections	Yes	No
Refractions	Yes	No
Suited for parallelization	Yes	Yes (Vertex & Fragment programs)
Suited for pipeline implementation	No	Yes

Table 1.1: Comparison of ray tracing and rasterization

may share larger or smaller implementation parts.

In Table 1.1 we have made a comparison of some of the features of ray tracing and rasterization.

Some claim that ray tracing is the physical correct way to render images. Ray tracing is in some situations more physical correct than rasterization but the only 100% physical correct way to simulate light is to simulate photons with wavelengths as described in a previous section.

It is easy to render a large scene in $O(\log n)$ by using ray tracing. Often a scene is also rendered in $O(\log n)$ by using rasterization. But this is because a lot of auxiliary data structures are used. These are primarily level of detail algorithms and culling algorithms [5]. Consequently, it is easier to achieve $O(\log n)$ render time by using ray tracing than by using rasterization. On the other hand, when one has spend weeks creating a detailed model it only takes a few minutes to insert the portals ([107]).

Currently one of the hot questions in real-time graphics is whether ray tracing will replace rasterization as the preferred method. This has been a very active discussion area and no consensus has been reached so far [2].

1.4 Shadows

Calculating shadows is one of the oldest research topics in computer graphics throughout the years and it has remained a very active research area. Recently it has become even more active because of the new programmable graphics processors.

When one uses ray tracing, it is straight forward to determine whether a sample point is located in the shadow of a point light source. A ray is traced toward

the light source and if an object is intersected before the light source is reached, the sample point is located in shadow ([143]). If the light source is an area light source, a number of points on the area light source are used and again a ray is traced from the sample point to the points on the light source. The percentage of rays intersecting an object between the sample point and the light source point determines the shadow percentage of the current pixel [31]. In order to avoid artifacts when calculating shadows from area light sources, a large number of rays have to be traced.

These two straightforward processes calculate accurately hard shadows and soft shadows, and they are generally considered the most accurate methods. The only problem is that shadow ray tracing is currently too slow to be used in real-time. Even for movie production it has been considered too slow [24]. Hard shadows are slow and soft shadows are many times slower because many more rays have to be used. Often it is necessary to use several hundred shadow rays per light source to achieve smooth soft shadows. In a scene with several hundred light sources more than ten thousand shadow rays will consequently be needed. Several approaches exist for reducing the number of shadow ray, although this is still an active research area ([138], [65], [72], [43]).

In general the research in the real-time shadow generation aims for faster methods that can replace ray tracing. All of these methods have shortcomings and therefore a lot of research has focused on fixing these shortcomings. In general, two goals have been high rendering speed and high image quality.

The two dominant real-time shadow methods have been volume shadowing and the shadow mapping.

The volume shadow method was introduced in [32]. This is a method where a volume is created that surrounds the area that is in shadow. The volume is calculated by finding the contour of the object that casts a shadow as seen from the light source. Then the edges in the contour are extruded in the direction away from the light source. In [58] a hardware implementation is described which uses the stencil buffer. The method can only be used for hard shadows but in [60] a technique is described that renders the shadow volume many times and in this way the shadow can be made soft. Although this method is much slower than the hard shadow version of volume shadows, it is still faster than traditional ray tracing. The hardware implementation of the shadow volume algorithm is problematic when the near viewing plane of the camera intersects with a shadow volume. Consequently, the algorithm is not robust. This problem was solved recently in [42]. In [42] a solution is demonstrated which is equally fast as the previous method and only minimally more complex and it may therefore be surprising that no one had come up with this clever idea before. One shortcoming of shadow volumes is that the entire shadow volume has

to be drawn and the graphics pipeline consequently becomes fill-rate limited. In [88], [84], [70] and [20] a number of techniques are presented which reduces the fill-rate requirement.

Another shortcomings of volume shadows is that they work best for simple objects with few triangles. Furthermore, the object should be a closed 2 manifold which makes it simpler to calculate the silhouette ([104], [88]). When objects are more complex, it becomes harder and more time consuming to calculate the shadow volume. In short, shadow volumes are best suited for relatively simple objects.

The other dominant method is shadows maps ([144]). The scene is rendered from the light, and a texture is created that contains the depth in the resulting image. The depth corresponds to the distance to the nearest object. When the scene is rendered a lookup is made into the depth texture to determine whether the current sample point is further away from the light source than the corresponding sample point in the depth texture. If so, the current sample point is located in shadow. The problems with this algorithm are two types of numerical problems. The first is the resolution with which the scene is rendered from the point of light. The lower the resolution the more blocky the shadow will be. The second problem is determining whether the current sample pixel is in shadow, as a numerical accuracy problem occurs when making a lookup into the depth texture. Many improvements have been developed for this algorithm in order to overcome these two problems. In [103] a method is proposed for avoiding the numerical problems of the limited numerical accuracy in the depth component. In both [117] and [44] methods are suggested for reducing the blocky appearance of low resolution shadow maps. This is done by using information about the camera position. Near the camera position higher resolution is used and further away less resolution is used. In this way the texture is used more efficiently without increasing the resolution. Recently this approach has been further refined ([145], [87], [1], [22]).

Only recently shadow maps have been implemented in commodity graphics hardware. Real-time applications have therefore not been able to utilize this technology. However, a slightly modified version has become quite popular instead, namely the projective texture. This is again an image rendered from the point of the light source but only the shadow casting object is rendered and it is rendered as purely black or grey and with no depth information. When the scene is rendered the image is used as a standard texture, and it is then projected onto the objects that should receive the shadow. The only problem is that the object can not cast a shadow on itself. This has been solved in [62] by dividing the object into several convex objects.

In [20] a method that combines shadow volumes and shadow maps is introduced.

The advantage of this method is increased speed for rendering shadow volumes as the high fill-rate requirement of volume shadows is removed by using shadow maps for parts of the shadow regions.

Another method is to use light maps ([16], [34]). Here a texture is applied to a surface, and the shadow (and light) at that location is pre-calculated. The pre-computation can be made by using e.g. ray tracing or any other technique, as long preprocessing time is acceptable. Geometry with light maps can be displayed very fast as it is only an extra texture that is applied to a surface. Be that as it may, this approach can only be used for static geometry.

A few other methods have been developed which are restricted to only casting shadows on planar surfaces. The simplest is introduced in [15]. By using this method, the 3D object is scaled to be completely flat in the direction of the planar surface and it is then drawn on top of this surface. A method that is also restricted to planar surfaces is the one presented in [51]. Here soft shadows are achieved but the shadow is slightly larger than the correct shadow would be, and the bigger the distance is between the shadow caster and the shadow receiver the more incorrect the shadow will be.

Higher quality soft shadows which can cast shadows onto arbitrary surfaces are currently a very active area of research ([4], [63], [11] [54]). The general real-time soft shadow approach is to use know hard shadow techniques like shadow volumes or shadow maps and then extend these by using the advanced GPU features.

In the near future it seems like shadow maps will be the preferred real-time shadow method because of its simplicity and its scalability ([104]).

Current real-time applications use one or several of these methods. Only very few real-time research applications use ray based shadow methods e.g. [132] and [97]. To our knowledge, no commercial real-time application uses real-time ray based shadows these days.

The methods that are only able to create shadows on planar surfaces were used in many games previously, but recently, game developers have begun to use more advanced geometry, and planar shadows are seldom used today.

Most games today use either shadow volumes or shadow maps (or the simpler projective textures). Most often light maps are used for static geometry.

Comparing shadow algorithms can be done with respect to a number of parameters. Some intuitive parameters would be: The possible types of shadow casting objects, the possible types of shadow receiving objects, and the quality of the

	Self shadows	Cast on curved surface	Possible geometry	Shadow quality	Speed
Ray tracing	Yes	Yes	No limit	Soft+Hard	Slow
Projection Shadow	No	No	No limit	Hard	Medium
Shadow Maps	Yes	Yes	No limit	Hard	Medium
Projective Texture	(No)	Yes	No limit	Hard	Medium
Volume Shadow	Yes	Yes	If contour defined	Hard	Medium
Light Maps	Yes	Yes	Only static	Hard+Soft	Fast

Table 1.2: Comparison of shadow algorithms

shadows.

The first two concern the type of objects that can cast a shadow and receive a shadow using this method. The third parameter applies to the physically correctness or quality of the shadows and whether the shadow method is able to produce hard or soft shadows. In general, the more physical correctness and the more general objects the shadow algorithm should be able to handle, the more computation time is required in order to produce the shadow. This is of course a very general description and not always entirely true, but it can nevertheless be used as a rule of thumb.

Unfortunately, none of the described shadow algorithms are superior to all others with respect to the three parameters which are mentioned above. If that were the case, only one of these algorithms would have to be implemented in any application. When comparing the algorithms on a spectrum of features, each algorithm has advantages and disadvantages. In our opinion, it is not likely that one of these algorithms in the near future will be superior to all other algorithms. In Table 1.2 we have made a simplified comparison of the shadow algorithms described above. It is very hard to create a good comparison as the shadow algorithms are very varied and many features can be used. E.g. it is hard to say which of the algorithms that is the most expensive with regards to rendering time, as it depends on the scene. Nevertheless, we think that the features we have chosen provide a fair comparison.

1.5 Indirect Illumination

Calculating the indirect illumination has turned out to be the most time consuming part of rendering global illumination images. Indirect illumination is the illumination originating from all other surfaces than the light source. The illumination of a sample point is therefore a function of the illumination of scene elements visible in the hemisphere of the sample point. But the sample point itself also contribute to the illumination of all points visible in its hemisphere.

This recursive dependency makes it very hard and most often impossible to create an explicit formula for the illumination at a sample point.

Several approaches exist for calculating the indirect light. The first method proposed was radiosity ([49]). Later a more general method namely path-tracing has been proposed ([69]). Path tracing still stands as the most correct way of calculating the illumination in an image. However the algorithm converges very slowly. Accordingly, path-tracing quickly becomes impractically for larger scenes and more complicated illumination models. For that reason, radiosity was for a period the method of choice, although it is not as general and accurate as path-tracing. The problem with radiosity is that it is very dependent on the complexity of the scene. The time complexity is $O(n^2)$ where n is the number of patches in the scene. This implies that as the scene grows, the method also becomes more and more impractical. Furthermore, when using the radiosity method, it is necessary to refine the mesh where lighting changes as the lighting information is stored directly in the mesh ([53], [57], [61]). This refining makes the complexity of the algorithm grow even more. Another issue is that radiosity is restricted primarily to diffuse interreflections.

Density estimation ([110]) has been invented as a method for calculating global illumination without the need for meshing. (Pseudo) photons are distributed from the light source and stored in a data-structure. The final illumination at a sample point is found by calculating the density of photons at the sample location. Although this method is fairly accurate, a substantial number of photons are required to calculate an image without noise, and it is almost impossible to eliminate the noise completely.

Independently of this approach, photon mapping has been developed and introduced in ([67] and [65]). The main idea in photon mapping is to divide the different effects shown in Figure 1.1 into separate parts. Each of these parts is then calculated separately. By carefully separating these parts, we are guaranteed that all effects are included and that no effect is included twice. The indirect illumination is calculated similarly to the density estimation technique, but a final gather step is added. Final gathering is an integration over the hemisphere, and it is described in greater details in Chapter 5. The advantage of using the final gather step is that far fewer photons have to be distributed, the noise is reduced and the remaining noise is of low frequency. This noise is more pleasing to the eye than high frequency noise. From radiosity method, it is well known that the final gather step removes much of the noise [27].

The final gather step as used when calculating indirect illumination is the most computational expensive step both in photon mapping and radiosity even though both methods can be used without this final gather step. When using radiosity, this step can be optimized by using hardware acceleration ([28]). In

	Monte Carlo Path Tracing	Radiosity	Photon map
Mesh dependent	No	Yes	No
Exploits caching	No	Yes	Yes
Types of illumination	All	Only diffuse	All

Table 1.3: Comparison of methods for calculating indirect illumination

Chapter 11 we demonstrate how it is possible to hardware accelerate the final gather step in photon mapping.

As the final gathering step is very costly, photon mapping is a very slow algorithm in its naive implementation. But a vast number of methods have been developed for speeding up the process ([66], [118], [23]). A key optimization is to use a fast ray tracer as tracing rays is part of the 'inner loop' of the algorithm.

In Table 1.3 some of the methods for calculating indirect illumination are compared.

As illumination changes over time it is necessary to recompute the entire illumination in the scene. In Chapter 8 we introduce a method which makes it possible only to recompute the irradiance at selected locations.

1.6 Using Indirect Illumination in Real-time Applications

In most real-time applications hardware rasterization is the method of choice for creating images. When we want to visualize the indirect illumination only two options currently exists. The first method is to apply a texture to the surface. By using this method the light at the texel centers is calculated and stored in the texture (this is often called *baking*). When the scene is drawn, the texture is modulated with the surface textures or the vertex colors. The second method is to store the illumination in the vertices and then blend the vertex colors with the textures used on the surfaces. These two methods have several positive and negative sides.

When a texture is applied the texture coordinates have to be calculated. This can be done manually by using a 3D modelling tool or it can be done automatically ([45]).

The texture should be applied to the geometry in such a way that the resulting texel-sizes are nearly the same all over the model. What is more, the texture

should be applied in such a way that there is a smooth interpolation across polygon boundaries. The textures on the polygons should fit nicely together. How textures are fitted to a model is a very active area of research nowadays.

Using vertices for storing the illumination has the weakness that enough vertices must be added for the illumination to be accurate enough. Indirect illumination can either be a high frequency or a low frequency function. Only few vertices need to be used in order to capture a low frequency function while many vertices need to be present to capture a high frequency function.

In some of the current gaming platforms (e.g. playstation 2) it is usually not possible to use texture maps for storing the illumination information. Although texture maps are present, the hardware design prevent the use of illumination texture. This makes vertices the only possibility for storing the illumination. The modelers therefore have to place vertices at all locations where the lighting changes. Although this sounds rather complicated it seems to be the standard when developing games ([107]).

1.7 Caustics

Caustics arise when a photon hits a diffuse surface after having been specularly reflected or refracted one or several times directly or indirectly from the light source. The most general way to solve global illumination is to use path tracing as presented in [69]. In this method, all rays originates from the eye point and it is hard to capture caustics accurately. The reason is that caustics is a focusing phenomena. Better methods are bidirectional path tracing methods ([76], [124]) as rays both originate from the light and from the eye. Specialized methods that have solely been designed for capturing caustics, has proven to produce the best results. In [8] a method is presented that traces photons from the light, and when a diffuse surface is hit after one or more specular reflections an energy amount is stored in a texture structure. In [142], three rays are traced from the light source simultaneously and they form a caustic polygon. When these rays hit a diffuse surface, the intensity is determined by the area that the polygon span. This approach does not work well when the diffuse caustic receiving surface is complex. In [21] caustic photons are traced and energy is deposited on the diffuse objects. Energy storing is done in image space, and finally a filtering is performed in image space. In [67] and [65] a slightly different approach is used. The photons are traced and stored in a kd-tree, and during the reconstruction, a search for the nearest photons is performed. This method is currently considered the most accurate method for reconstructing caustics. We will describe the method in more detail in Chapter 4. Real-time caustics

have been approximated in [136] but the solution is limited to single specular interaction and the method assumes that the light sources are positioned far away from the specular reflectors. Furthermore, they do not handle occlusion between the light source and the specular reflector. The quality of the caustics are fairly low for interactive frame rates. Recently, good quality caustics have been implemented by using a fast ray tracer running up to 21 frames per second on a setup with up to 18 dual processor PCs [50].

In Chapter 12 we propose a real-time method based on photon mapping.

1.8 Real-time Global Illumination

A fairly small number of methods for rendering real-time global illumination have been produced. In the following, we will take a brief look at some of these (for an good survey see also [33], [126] and Chapter 12 in [125]).

1.8.1 Progressive Update of Global Illumination

In [122], a method for calculating interactive global illumination is described. The scene is rendered by using graphics hardware and the illumination is stored in a hierarchical patch structure in object space. At first, the top level is used but based on priorities calculated in camera space the patches are then subdivided. The illumination is calculated by using a path tracing algorithm and the illumination of only 10 to 100 patches can be calculated per second on each CPU. As a result of this low number up to 16 CPUs are used. When the camera or objects are moved quickly artifacts will occur. The illumination is stored in vertices which makes the method sensitive to the meshing. The method in [122] has some similarities with the Render Cache ([133], [134]) and the Holodeck ray cache ([137]) as the calculation of the illumination is decoupled from the display process. Although, in Render Cache and the Holodeck ray cache, the calculations are performed in image space while the calculations in [122] are performed in object space. Another similar example of progressive image space global illumination based on progressive ray tracing is the method presented in [12] where discontinuities are tracked by using discontinuity edges.

1.8.2 Instant Radiosity

Instant Radiosity was introduced in [71]. In this method "big" photons are traced from the light source (also called Virtual Point Lights). As each photon hits a surface, the intensity and new direction of the bounced photon are handled according to the BRDF of the surface. The clever part in this algorithm is that a hardware light-source is placed at this intersection point, and then the scene is rendered. On older hardware only eight hardware light sources are available. Therefore, only a few of the photons are traced at each frame. Each frame is added to the accumulation buffer, and by combining the image over a number of frames, the final result is achieved. When a frame reaches a certain age, its contribution will automatically be removed from the accumulation buffer. If the properties of the lights change, the scene will gradually be modified in order to reproduce the new lighting conditions. But if the elements in the scene are moved or the camera changes viewpoint, this will invalidate the content of the accumulation buffer, and the algorithm will need some time to create a new valid image.

On modern hardware, it is possible to implement hardware lighting in both vertex- and fragment-programs, and many more lights per frame is then possible. Fragment-programs calculate the lights more accurately, while vertex programs will calculate the lights faster. With this approach, far fewer frames will be necessary for the image to converge. To our knowledge no one has implemented this yet.

1.8.3 Selective Photon Tracing

In [35] a number of photons are traced from the light sources. As a photon bounces of a surface the direction and energy of the bounced photon are handled according to the BRDF of the surface. Nevertheless, only diffuse surfaces are handled in their application. A fixed number of photons are used and these photons are divided into groups. By using properties of the quasi-random sequence the photons in each group have very similar paths in the scene (see Chapter 6 for more details on photon distribution by using Halton sequences). By continuously retracing the primary photons and by recording which objects these photons intersects with it is possible to register changes in the scene. If a change is discovered, all photons from this group are re-traced using the old scene configuration, and when a photon bounces off a surface the photon energy is subtracted from the nearest vertex. When the photons are traced at the new scene configuration, photon energies are added to the nearest vertex.

As the illumination is stored in the vertices, the quality of the rendering is dependent on the placement of these vertices. More vertices in an area mean more detailed light. On the other hand, if too many vertices are placed in an area, aliasing will occur because relatively few photons are being used. One way to avoid this is to calculate the illumination of a vertex by averaging over the neighboring vertices.

In the implementation in [35] they use their new technique for the indirect illumination. The direct illumination and the shadows are calculated by using traditional hardware point lights and traditional hardware-accelerated hard-shadows. The shadow method that they have used is volume shadows, although any type of hardware-accelerated shadow could have been used.

In Chapter 9 we present a modified version of the QMC photon distribution approach introduced in [71] and [35].

1.8.4 Interactive Global Illumination

In [130] and [128] a method that is related to Instant Radiosity is used. The method is often called *Instant Global Illumination*. "Big" photons are distributed from the light source and small light sources are created when the photons bounce off the surfaces as in Instant Radiosity. In this way, a vast number of light sources are created. The scene is rendered by using ray tracing on a cluster of traditional PCs ([130]). Since a sample point is potentially illuminated by a huge number of light sources, it would be very expensive to trace rays toward all these light-sources. Therefore, rays are cleverly send to those light sources that have the highest probability of illuminating the current sample point ([128]). No frame to frame coherence are utilized, and all the indirect illumination is completely recalculated for each frame. The positive side of this is that large parts of the scene can be modified without any major latency for the illumination calculations to converge. The negative side is that the indirect illumination calculations are quite crude. In order to avoid the typical Monte Carlo noise in their images a screen space filter is used.

1.8.5 Photon Mapping on Programmable Graphics Hardware

In [100] full global illumination is calculated almost entirely on the GPU. The approach used is that of photon mapping. The direct illumination is calculated by using ray tracing which is implemented on the graphics hardware. The

photons are distributed from the light source and stored in texture maps. The density estimates are calculated by lookups in the texture maps. The texture maps are used as ordinary data-structures. In [100] it is demonstrated how to use the photons in the textures for both indirect light and caustics. The number of traced photons is fairly low and the positions at which the photons are stored are also slightly approximated. As this only runs in near real-time, there is no remaining time for the expensive final gathering step traditionally used in photon mapping. Therefore, high frequency noise appears in the indirect illumination. On the other hand the caustics are fairly accurate. The method does not exploit frame-to-frame coherence, and that being the case, the solution has to be recalculated from scratch whenever the scene or viewpoint is modified. Currently CPU based methods are faster than this method. Nevertheless, it is an interesting method with many new ideas which are likely to be improved and further refined as the speed and features of the graphics cards rapidly are improved.

1.9 Real-time Global Illumination Summary

In this introduction we have looked at the different elements that all contribute to a complete global illumination solution. We have described these elements and seen that each element can be computed in different ways. We have especially focused on methods that can be used in real-time. We have also described some elements that will be handled in the forthcoming chapters.

1.10 Analysis and Structure of this Thesis

As described in this introduction the elements that constitute global illumination have been examined extensively in the literature. In that case, what are the challenges left and where is the room for improvement? Many techniques have been presented which make global illumination faster or more accurate, but no definitive solution for real-time global illumination has been introduced.

Based on our literature survey we believe that indirect illumination and caustics are areas for which there have not been developed a sufficient good real-time solution for. These two effects are characterized by several light bounces, which means that the entire scene can affect any point. That is the property that makes them harder to calculate. In this thesis we will focus on creating a real-time solution for these effects.

Many different kinds of strategies have been chosen for calculating caustics and indirect illumination as described in this introduction. For instance solving caustics by using photon mapping is very different algorithmically from using the texture lookup method described in [136]. Also with regard to indirect illumination, methods like e.g. radiosity and photon mapping are very different in their nature, even though they solve the exact same problem. When creating a new method, it can either be based on existing methods or a completely new concept. The area of global illumination is a well researched area. We believe that it is more likely that a good solution will build on known methods than on entirely new concepts. It is therefore important to look at the existing methods and examine their strengths and weaknesses.

When examining indirect illumination, it is clear that it often does not change over a short period of time, or change slowly. This suggests that coherence can be used in order to minimize the calculations per frame. Some of the real-time techniques that have been presented do not use frame to frame coherence, these are [100] and [128]. Instead they continuously recalculate the indirect illumination. We believe that frame to frame coherence as presented in e.g. [35] and [122] is important in order to minimize the recalculation which is necessary per frame.

In Chapter 6 we examine the Quasi Monte Carlo sequences which is one of the building bricks in their selective update of the indirect illumination.

As described in this introduction GPUs have become more powerful and some calculations can be performed on the graphics card with great advantages. But the architecture of the GPU is very different from the architecture of the CPU. Therefore, just porting an existing algorithm is not possible. In [100] many of the basic principles of photon mapping is implemented almost entirely on the GPU. Nevertheless the method is slower and less accurate than photon mapping implemented on the CPU. This shows that all the features and the speed of the GPU do not automatically solve the problems of real-time global illumination. We believe that if the GPU should be used in real-time global illumination, it should only be used where the architecture of the GPU has advantages over the CPU. In Chapter 3 and Chapter 5 we look at algorithms where both graphics hardware and software methods can be used.

Real-time global illumination has been presented by using many parallel processors ([128], [122], [137]). We will not examine this area further but instead we will try to develop a solution that can run on commodity hardware. Nevertheless, it may be that the architecture of the standard PC or game console in the future will contain many processors. In that case the algorithms for using many parallel processors will be increasingly interesting.

Photon mapping is implemented on graphics hardware in [100]. Although their method is not real-time, we believe that it is possible to develop a real-time method based on photon mapping. In Chapter 4 we will describe the photon mapping algorithm in much more detail than in this introduction.

Distributing the photons more evenly e.g. by using Halton sequences does lower the noise but it can not completely remove it. Currently no real-time method uses final gathering. This may be because final gathering is not necessary or because it is two expensive to calculate in real-time. Anyone who have tried to implement photon mapping know that direct visualization of the photon density estimates will produce noisy images. Consequently, something has to be done with the noise in the images. Final gathering is one method for removing the noise in the image, but it may not be the only one.

Images calculated by using radiosity does not contain noise even though final gathering is not used. But radiosity has other disadvantages which will discussed in Chapter 4. Even methods that uses photon distribution like Instant Radiosity [71] and Instant Global Illumination [130] avoids high frequency noise without the need for final gathering. This is achieved by using only few photons with deterministic paths. However, using only few photons will lower the accuracy of the solution.

We believe that final gathering is a very important element when calculating high quality indirect illumination. In Chapter 5 we will take a look at the hemi-sphere integral which is the basis of final gathering.

In this introduction we have described many methods for solving global illumination. Many of the methods have only been described very briefly. Since describing thoroughly all these methods would be more than could fit into several books we will only describe the elements that are important for the Contribution Part. The intention is that the elements in the Theory Part should lead to the new methods that are presented in the Contribution Part.

In the Contribution Part our method for implementing photon mapping for real-time applications will be presented. Finally, a summary and a conclusion will be given in the Conclusion Part.

Part II

Theory

Chapter 2

Illumination Theory

Illuminating a 3D scene is a fundamental problem in computer graphics. Illumination is what happens when a light source such as the sun or a lamp sends out photons and objects are illuminated.

The math behind illumination is well understood, but unfortunately it is usually too time consuming to make these calculations accurately and most of the time it is even impossible. In particular, it becomes problematic when the requirement is that a 3D scene should be illuminated in real-time, but the problem also arises in animations for movies. Therefore, all methods used today for calculating the illumination rely on approximations in one way or the other. In general, the rule is that the faster the images have to be generated the more approximations are necessary. But before we take a closer look at some of these approximations, we will take a closer look at the illumination theory. By doing this, we will know what approximations we must use in order to calculate the images faster.

The rest of this chapter is about the illumination theory.

An overview of the symbols that will be introduced in this chapter can be seen in Table 2.

An overview of the terms used in Physics, Radiometry and Photometry can be

Symbol	Name	Unit
Q_λ	*Spectral radiant energy*	$\frac{J}{nm}$
Q	*Radiant energy*	J
Φ	*Radiant flux*	$W = \frac{J}{s}$
I	*Radiant intensity*	$\frac{W}{sr}$
E	*Irradiance*	$\frac{W}{m^2}$
M	*Radiant exitance*	$\frac{W}{m^2}$
B	*Radiosity*	$\frac{W}{m^2}$
L	*Radiance*	$\frac{W}{m^2 sr}$
L_λ	*Spectral radiance*	$\frac{W}{m^2(sr)(nm)}$

Table 2.1: Symbols used in Radiometry

Physics	**Radiometry**	**Photometry**
Energy	Radiant Energy	Luminous Energy
Flux (Power)	Radiant Power	Luminous Power
Flux Density	Irradiance Radiosity	Illuminance Luminance
Angular Flux Density	Radiance	Luminance
Intensity	Radiant Intensity	Luminous Intensity

Table 2.2: Terms used in Physics, Radiometry and Photometry

seen in Table 2. In general in this thesis we will use the Radiometric terms.

The theory of illumination is divided into two categories, photometry and radiometry. The difference between these two categories is that photometry is the theory about perception of light while radiometry is the physics of light. In the following we will take a look at the physics of light. This thesis will not go into details about the perception of light.

The basic element is a photon, which is a packet of energy (e_λ). This energy is inversely proportional to the wavelength (λ) of its corresponding wave:

$$e_\lambda = \frac{hc}{\lambda} \tag{2.1}$$

The proportionality constant is Planck's constant ($h = 6.6310^{-}34Js$), and the constant c is the speed of light (in vacuum $= 299.792.458\frac{m}{s}$).

The wavelength(λ) of the photon is inversely proportional to its frequency(f).

$$\lambda f = c \tag{2.2}$$

An important property of photons is that two photons do not interact with each other. This is an important property when calculating the illumination as the each photon or energy bundle can be handled individually.

Spectral radiant energy Q_λ is the energy in a number of photons which have the same wavelength.

$$Q_\lambda = ne_\lambda \tag{2.3}$$

Radiant energy (Q) is the energy of a number of photons regardless of their wavelength.

$$Q = \int_0^\infty Q_\lambda d\lambda \tag{2.4}$$

The visible spectra of wavelengths are those having frequencies in the range from 380 to 780 nanometers and usually only this interval is considered in computer graphics.

Radiant energy or radiant flux (Φ) is a measure of light energy flowing per unit time through all points and in any direction. The flux is dependent on the wavelength, and therefore the notation should be Φ_λ, or even more precisely, λ should be in the range $[\lambda, \lambda + \delta\lambda]$. In the following we will write Φ without specifying the wavelength. It is further noted that a given flux does not correspond to a fixed number of photons, as the energy of a photon is dependent on its frequency (as seen in Equation 2.4).

In most practical applications, the light is calculated at three different wavelength, corresponding to red, green and blue. This is an approximation as the equations should of course be solved for infinitely many intervals in the range from 380 to 780 nanometers. Nevertheless, this approximation will be sufficient for most applications. For some applications it may be necessary to use more color bands. The famous Cornell box has in its specification 76 spectral reflection coefficients for its materials ([49], [28], http://www.graphics.cornell.edu/online/box/). This is called multi spectral rendering. However, in the end, all these color bands

have to be re-sampled to red, green and blue as this is what current display devices are capable of displaying. Still, a more accurate rendering is achieved by using multi spectral rendering.

As photons travel at the speed of light, it seems that the illumination happens instantaneously when light is switched on or off. This means that the flow of light finds an equilibrium and does not change unless the environment is modified.

On a surface this equilibrium can be described by using a conservation equation which contains five variables.

- e light *emitted* from the surface.
- i light coming from elsewhere hitting the surface(*incident*).
- s light hitting the surface and flowing through it (*streaming*).
- r light *reflected* on the surface.
- a light that is *absorbed* in the surface.

The flux conservation equation can now be expressed using these five variables:

$$\Phi_e + \Phi_i = \Phi_s + \Phi_r + \Phi_a \tag{2.5}$$

When radiant flux is considered with regard to a surface, it is called *radiant flux area density*, $\frac{d\Phi}{dA}$. This term can be used both for *in-scattered* light and reflected light. A more specific term for reflected radiant flux area density is *radiant exitance* (M) or radiosity (B). The more specific term for in-scattered radiant flux area density is *irradiance* (E).

2.1 Solid Angle

The direction ($\vec{\omega}$) is a vector in 3D. The solid angle $d\vec{\omega}$ corresponds to a patch on a sphere and it is unitless. Although it is unitless it is often termed steradian and the unit used is sr (there is 4π on a complete sphere, and 2π on a hemisphere). Solid angle also exists in 2D where it is an interval on the unit circle (2π on a complete circle). The area on the unit sphere of $d\vec{\omega}$ is $sin\theta d\theta d\phi$. When describing the direction with regard to a surface, $\vec{\omega}$ is usually the vector in the

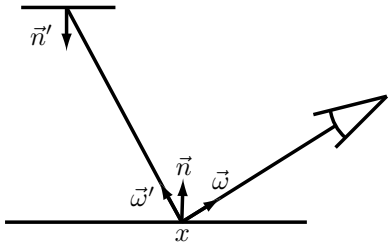

Figure 2.1: Definition of normals and angles

outgoing direction and $\vec{\omega}'$ is the vector in the incoming direction. Both vectors are pointing away from the surface and in the same direction as the normal (see Figure 2.1).

2.2 Radiance

The total amount of energy leaving a surface in all directions is measured as radiant flux area density. *Radiance*(L) is a subpart of this. Radiance is the energy that leaves a surface, per unit projected area of the surface per unit solid angle of direction. Mathematically this can be expressed as:

$$L = \frac{d^2\Phi}{dAcos\theta\, d\omega} \tag{2.6}$$

Radiance is one of the most important properties in computer graphics as it states how much energy is leaving a point on a surface in a particular direction. When knowing this quantity it is possible to illuminate a surface.

Another way of viewing radiance is to think about all the light incident on a surface, and then only consider light leaving the surface in a particular direction. But a point is approximated as an infinitesimal area which is the reason for the dA factor in the equation. Likewise, the direction is also approximated as an infinitesimal cone $d\omega$. The cosine factor is present because the projected area will be smaller as the direction of the cone is altered from the direction of the surface normal. It is noted that the cosine factor $(cos\theta)$ also can be expressed as $(\vec{\omega} \cdot \vec{n})$.

The flux can of course then again be calculated as an integration over the radi-

ance over all directions(Ω) and the area (A) multiplied by the cosine factor.

$$\Phi = \int_A \int_\Omega L(x, \vec{\omega}) cos\theta d\vec{\omega} dx \tag{2.7}$$

Radiance is often expressed as $L(x, \omega)$ meaning that the radiance is a parameter of a given point(x) and a given direction($\vec{\omega}$). A point has three degrees of freedom and a direction has two degrees of freedom. Therefore, radiance is a five dimensional function.

Radiant intensity is the quantity of light in a particular direction going through a surface.

2.3 Reflectance

Reflectance(ρ) is the relative amount of the *incident* light that is *reflected*. The remaining light is either *absorbed* or *streaming* (setting the emission to zero in Equation 2.5). At a surface point (x) this can be expressed as:

$$\rho(x) = \frac{d\Phi_r}{d\Phi_i} \tag{2.8}$$

When setting the emission to 0 it is clear from Equation 2.5 that $\rho(x)$ will be in the range $[0; 1]$ as all variables are positive.

2.4 BRDFs

The relationship between incoming light (E_i) and outgoing light (L_r) at a point (x) can be described using a function of three parameters. These parameters are the point (x), the incoming direction of light ($\vec{\omega}'$) and the outgoing direction of light ($\vec{\omega}$). This function (f) is called the *Bidirectional Reflectance Distribution Function* [91]

$$f(x, \vec{\omega}, \vec{\omega}') = \frac{dL_r(x, \vec{\omega})}{dE_i(x, \vec{\omega}')} \tag{2.9}$$

The function f has to comply with two rules. As f is integrated over the hemisphere the function has to be less than or equal one. This is due to the rule of energy preservation.

$$\int_{\Omega} f(x, \vec{\omega}, \vec{\omega}') d\omega' \leq 1 \tag{2.10}$$

for any point (x) and any direction (ω).

Another property of f is that the incoming and outgoing direction can be swapped. This is called Helmholt reciprocity.

$$f(x, \vec{\omega}, \vec{\omega}') = f(x, \vec{\omega}', \vec{\omega}) \tag{2.11}$$

A diffuse surface (also known as Lambertian surface) reflects incident light uniformly in all directions over the hemisphere. This means that a diffuse surface looks the same from all viewing directions. In that case the BRDF is a constant function:

$$f(x, \vec{\omega}', \vec{\omega}) = f_{r,d} = constant \tag{2.12}$$

If a surface is diffuse, it can simplify many calculations e.g. the calculation of the radiosity. A BRDF can either be an analytical formula or a measured function. A vast number of analytical formulas exist which approximate different types of surfaces e.g. Phong, Blinn, Ward, Cook-Torrance, Ashikmin etc. (a good overview can be found in e.g. [46], [5], [48]). The BRDF of a surface can also be measured by using a physical device ([139]).

When one knows the BRDF of a surface, one can render most surfaces correctly. The assumption of the BRDF is that the energy leaving a point x is a function of the in-scattered energy at that point. This assumption is correct for surfaces such as metal and rocks.

Other types of materials as e.g. milk and skin do not not comply with this assumption. In these materials the light energy enters the surface at a point and it will most likely exit the surface at a nearby point. In order to describe such surfaces, a more advanced function than the BRDF is necessary. Here it is necessary to use a BSSRDF ([91], [68]). Another type of surface is transparent surfaces. These can be characterized by using two BTDFs (Bidirectional Transmittance Distribution Function). A complete description needs 2 BRDFs and 2 BTDFs, which can be gathered in a single BSDF (Bidirectional Scattering Distribution Function).

Most BRDFs can not handle *Fluorescence*, which when the reflected light has a different frequency than the incoming light. Another effects that is usually not accounted for is *phosphorescence*, which is when energy is stored and re-emitted later.

2.5 Calculating the Radiance

In Equation 2.7 we showed how to calculate the flux leaving a surface. The radiance leaving a surface point in a particular direction can be described as:

$$L = L_e + L_r \tag{2.13}$$

Expressing the radiance as a function of the incident radiance and the BRDF of the surface yields the following formula:

$$L(x, \vec{\omega}) = L_e + \int_{\Omega} f(x, \vec{\omega}', \vec{\omega}) L_i(x, \vec{\omega}', \vec{\omega})(\vec{n} \cdot \vec{\omega}') d\omega' \tag{2.14}$$

This equation is also known as the *Rendering Equation* ([69]) although it was originally introduced in a slightly different form.

2.6 Describing the Path of Light

Photons originate from a light source. Before the photon finally enters the eye, the path of the photon can be fairly complex. Heckbert introduces a compact notation for expressing this ([56]).

The notation is defined as follows:

- **L** a light source
- **D** a diffuse surface
- **S** a specular surface
- **E** the eye

In order to describe multiple paths the symbols **+**,*****,**?** and | are introduced. The meaning of these symbols are as follows:

- + one or more
- * zero or more
- ? zero or one
- (a|b) either a or b

For instance the expression **L(S|D)+E** means that the light hits one or more surfaces that are either diffuse or specular before hitting the eye. Caustics can be described using the following expression: **LS+DE**.

2.7 Summary

In this chapter we have introduced many of the fundamental properties of light. This is just a brief introduction. For more information see [114], [66], [37], [38].

Chapter 3

Direct Illumination

As described in the introduction, the direct illumination for real-time applications can be calculated both by using ray tracing and by using rasterization. Both of these methods compute the following equation:

$$L(x, \vec{\omega}) = \int_{\Omega} f(x, \vec{\omega}', \vec{\omega}) L_i(x, \vec{\omega}')(n \cdot \vec{\omega}') d\vec{\omega}' \tag{3.1}$$

If we consider an area light source, the expression $d\vec{\omega}$ can be expressed as:

$$d\vec{\omega}' = \frac{(\vec{n}' \cdot \vec{\omega}') dA'}{(x' - x)^2} \tag{3.2}$$

Where dA' is the area of the light source and $(x' - x)^2$ is the squared distance between the light source and the point being shaded. $\vec{n}'$ is the normal of the light source (See Figure 3.1)

If the distance between the area light source and the objects illuminated by the

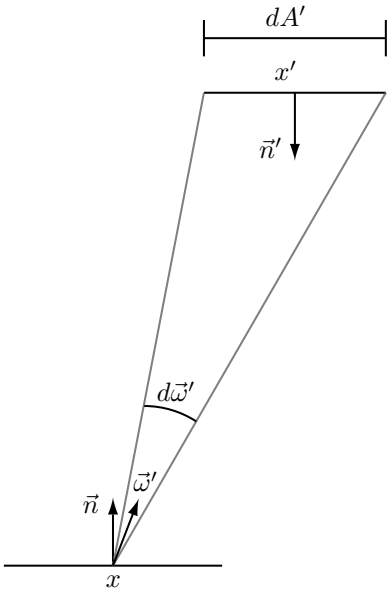

Figure 3.1: Calculation of $d\vec{\omega}'$

light source is large Equation 3.1 can be approximated as:

$$L(x,\vec{\omega}) \approx f(x,\vec{\omega}',\vec{\omega})L_i(x,\vec{\omega}')(\vec{n}\cdot\vec{\omega}')\frac{(\vec{n}'\cdot\vec{\omega}')A'}{(x'-x)^2} \tag{3.3}$$

This expression can be used directly in ray tracing and rasterization for evaluating the direct light.

3.1 Ray tracing

By using ray tracing the shading code can be implemented in C as shown in Figure 3.2. This shader implements a Lambertian surface. The result of this shader can be seen on a sample scene in Figure 3.3.

```
Radiance shade(Vec sample_point, // Point currently being shaded
               Vec sample_normal, // Normal at sample point
               Vec light_point , // Center of light source
               Vec light_normal, // Normal of light source
               Radiance L_i, // Radiance of the light
               float area, // Area of the light source
               Vec diffuse ) // Lambertian coefficients
{
    Vec dir = ( light_point -sample_point).normalize();
    float dist = ( light_point -sample_point).length();
    Vec f = diffuse/M_PI;
    float cos_theta = dot(sample_normal, dir);
    float cos_phi = dot(light_normal, -dir);
    return f * L_i * cos_theta * cos_phi * area / ( dist * dist );
}
```

Figure 3.2: Shading code in C

3.2 Rasterization

The shading, which is demonstrated in Equation 3.3, can also be implemented by using rasterization. As the equation has to be evaluated per pixel it is necessary to implement it in a fragment program. Some of the inputs to a fragment program are most easily expressed in a vertex program, and then the parameters are transferred to the fragment program.

A vertex program accepts two kinds of input variables, *varying* and *uniform*. Varying variables are specific only to a single vertex while uniform variables are values that can be used by all vertices. A vertex program can do any kind of arithmetic on this vertex but in the end it must transform the vertex into the canonical view space.

The fragment program is a program that is executed for each pixel that is drawn to the screen. The fragment program can accept the output variables from the vertex program. But the only output form the vertex program is a color and a depth that are written to the framebuffer or any other buffer which is currently active.

The code for implementing Equation 3.3 can be seen in Figure 3.4. This shader is similar to the one described in 3.2 but the notation is Cg ([94], [86]). The

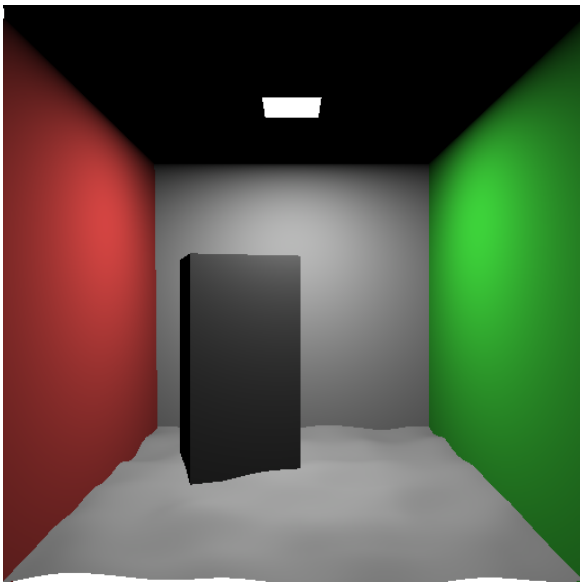

Figure 3.3: A Cornell box with a bumpy floor rendered using ray tracing implemented with the code shown in Figure 3.2

result of this shader can be seen on a sample scene in Figure 3.5.

The scene in Figure 3.3 and 3.5 has a bumpy floor and approximately 8000 polygons. The resolution is 512 times 512 pixels. Rendering the simple scene using rasterization took 0.01 seconds while rendering the scene by using ray-tracing took 2.5 seconds on a Pentium 4, 2.4 GHz. It is probably possible to optimize both the ray tracing implementation and the rasterization implementation. Nevertheless, it is likely that the timing differences will still be approximately the same.

It is noted that the shading of both ray tracing and rasterization is similar.

When rendering area light sources it is common to use many samples on the light source and average these. This can be implemented by using ray tracing and selecting random points on the light source. It can also be implemented by using rasterization. In that case a number of image could be rendered and the final result would be an average of these images, although other options exists for implementing this.

The surface that we chose to render was a simple Lambertian surface but using

```
void vertex_program(float4 in_sample_point : POSITION,
                    float3 in_sample_normal:NORMAL,

                    uniform float4x4 model_view_proj,

                    out float4 h_point: POSITION,
                    out float3 sample_point,
                    out float3 sample_normal)
{
    h_point = mul(model_view_proj, in_sample_point);
    sample_point = in_sample_point.xyz;
    sample_normal = in_sample_normal;
}

float3 fragment_program(float3 sample_point,
                        float3 sample_normal,
                        float3 light_point ,
                        float3 light_normal,
                        float3 L_i,
                        float area,
                        float3 diffuse ) : COLOR
{
    float3 dir = ( light_point -sample_point).normalize();
    float dist = ( light_point -sample_point).length();
    float3 f = diffuse/M_PI;
    float cos_theta = dot(sample_normal, dir);
    float cos_phi   = dot(light_normal, -dir);
    return f * L_i * cos_theta * cos_phi * area / ( dist * dist );
}
```

Figure 3.4: Cg code for shading using a Lambertian surface and an area light source.

textures or procedural textures will not change the result ([39]). Currently there are some limitations to how complex the shaders can be in vertex and fragment programs, but these limitations are constantly being reduced.

Although much functionality is achieved by using the programmable pipeline, a number of solutions for advanced rendering exist. They use the fixed function pipeline. Rendering arbitrary BRDFs using the fixed function pipeline is introduced in [59]. Here a few number of cubemaps are used to factorize the general BRDFs.

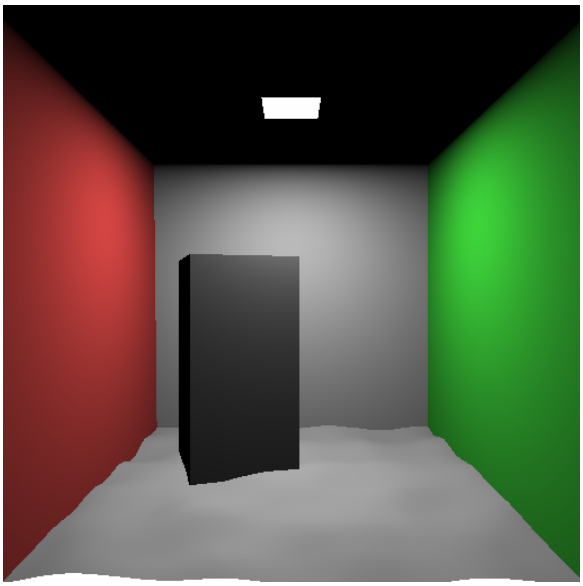

Figure 3.5: A Cornell box rendered using the code in shown in Figure 3.4

In [29] the shade tree is introduced. The shade tree is the basis for the RenderMan shading language which is an advanced widely used shading language ([52]). In [98] a method is introduced which uses the fixed function pipeline for approximating some of the RenderMan shading language features. This is done by using multipass rendering where intermediate values are written to the framebuffer and modified several times. The modification is performed by rendering the same geometry over and over but using different textures for doing mathematical operations directly in the framebuffer. The disadvantage of this approach is that the same geometry has to be rendered many times and this can be quite costly. Although this multi pass technique is quite powerful, it is also somewhat limited as only some mathematical operations are possible.

3.3 Summary

In this chapter it has been demonstrated that ray tracing and rasterization can be made to produce the same results. This is the case in many scenes. In many situations it will therefore be equally valid to use either ray tracing or rasterization.

Chapter 4

Photon Mapping

Photon mapping is a technique for calculating global illumination in a scene. It is a hybrid method that uses different techniques for calculating the individual contributions described in Chapter 1. In the following, each of the components will be described.

4.1 Dividing the Incoming Radiance

The basic principle of photon mapping is to divide the incoming radiance into a number of components which can be handled individually. The incoming radiance at a sample point can be divided into three components (See also Figure 1.1).

$$L_i(x, \vec{\omega}') = L_{i,l}(x, \vec{\omega}') + L_{i,c}(x, \vec{\omega}') + L_{i,d}(x, \vec{\omega}') \tag{4.1}$$

where

- $L_{i,l}$ is the direct light, i.e. $L(D|S)E$

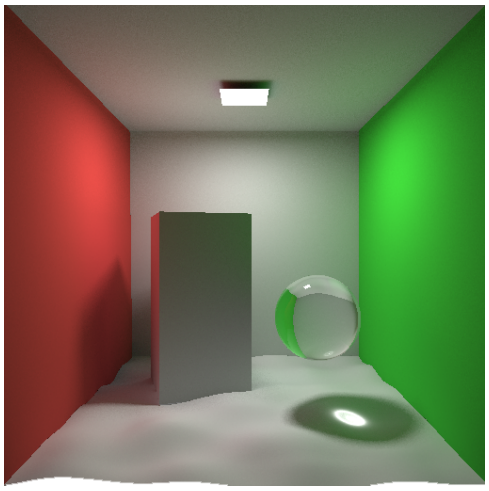

Figure 4.1: Cornell box with both direct illumination, caustics and indirect illumination

- $L_{i,c}$ is the caustics, i.e. $L(S+)DE$
- $L_{i,d}$ is the indirect light, i.e. $L(D|S) * D(S|D) + E$

The direct illumination is calculated using traditional Whitted Style ray tracing ([143]).

The caustics and indirect illumination parts are each calculated separately in a two step process. The first step is to distribute a number of pseudo photons from the light source. The second step is the *reconstruction* phase where the photons are used to calculate the caustics and the indirect illumination. This step is called the reconstruction because it only uses already distributed energies.

In Figure 4.1 a Cornell box which contains both direct illumination, caustics and indirect illumination is rendered using photon mapping. In Figure 4.2 only the direct illumination is displayed, in Figure 4.3 only the caustics are displayed and in Figure 4.4 only the indirect illumination is displayed.

The indirect and caustic illumination are calculated from the photon maps by using a density estimation technique. In the following, we will give a more detailed description of this process.

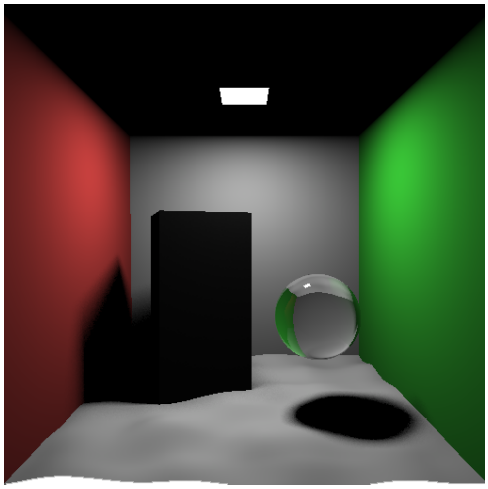

Figure 4.2: Cornell box with direct illumination and soft shadows

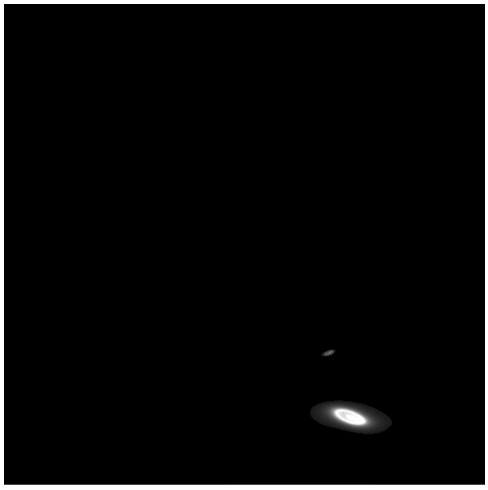

Figure 4.3: Cornell box with caustics

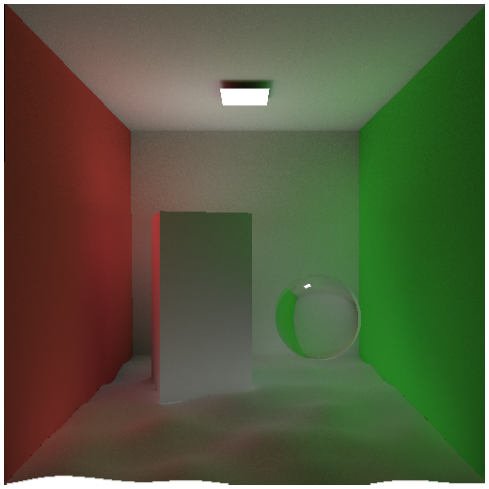

Figure 4.4: Cornell box with indirect illumination

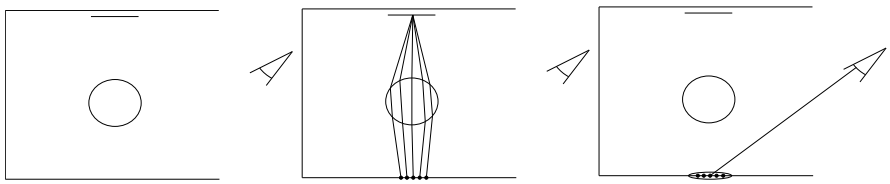

Figure 4.5: The process of calculating caustics using photon mapping.

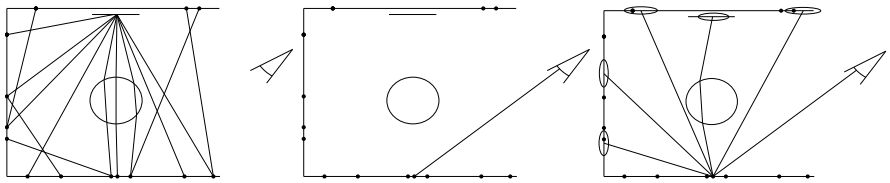

Figure 4.6: The process of calculating indirect illumination using photon mapping.

4.2 Distributing Photons

A light source has a color and an amount of energy which it distributes to the environment. In photon mapping, this energy amount is divided into a number of pseudo photons. Each of these photons are traced through the environment. The initial direction of each photon is dependent on the type of light source.

Photons are distributed twice from the light source. Once for the indirect illumination photon map, and once for the caustic photon map. The photon map method is designed in such a way that no light contribution is counted twice even though the light is calculated in many different ways.

4.2.1 Storing Photons in the Indirect Illumination Photon Map

As a photon is traced through the environment and intersects a surface, it is necessary to handle the intersection appropriately. If the surface is a specular surface, the photon is not altered in any way but reflected or refracted according to the BRDF of the surface. If the surface is diffuse, the photon is stored in the photon map used for the indirect illumination. Whether the photon is terminated or reflected is determined by using a technique called Russian Roulette ([116], [10] [66]). If it is determined that the photon is reflected, the direction is chosen based on the BRDF of the surface. E.g. a diffuse surface will reflect a photon in any direction on the hemisphere with equal likelihood. As the Russian Roulette method terminates some photons, it is necessary to re-scale the power of other photons in order not to remove energy from the system. The energy of the photon is also modified by the BRDF of the surface. The Russian Roulette method is used in order to avoid very long photon paths where the energy of the photon only contributes minimally to the final result.

4.2.2 Storing Photons in the Caustic Photon Map

Caustics only occur when the photons first hit a specular surface (e.g. a mirror or glass) and then hit a diffuse surface. Accordingly the paths that we are interested in are $L(S+)DE$ (see Section 2.6).

When the photons are distributed from the light source, all photons that do not hit a specular surface as the first hit are ignored. If a photon hits a specular surface, it is reflected or refracted according to the properties of the surface.

When the photon then hits a non-specular surface, it is stored in the caustic photon map.

An optimization to this scheme is not to trace photons in directions where no specular surfaces are present, as caustics only occur when photons intersect with specular reflectors [66].

4.3 Density Estimation

Storing the photons can be done in any data-structure, but during the second step of the photon mapping algorithm, the density needs to be found. The density of the photons is then used to estimate the irradiance.

Density estimation is the process of finding the density at a specific point. As the photons are distributed in 3D the probability of finding any photons at a random 3D point is zero. Therefore techniques have to be used to give a measure of the density at a specific position. Many different techniques exist for this purpose ([113]). Currently, the most popular method is the N-nearest neighbors method. A fixed number of nearest neighbors are chosen and these nearest photons are found. The energy of all these photons is summed up and divided by the area that the photons span.

By using this method the outgoing radiance can be described in the following way:

$$L_r(x,\vec{\omega}) = \int_\Omega f(x,\vec{\omega}',\vec{\omega})\frac{d^2\Phi_i(x,\vec{\omega}')}{dA_i} \approx \sum_{p=1}^{n} f(x,\vec{\omega}',\vec{\omega})\frac{\Phi_p(x,\vec{\omega_p})}{\Delta A} \qquad (4.2)$$

The area which these photons span can be calculated in different ways. One method is to use the convex hull of the photons. A faster but less accurate method is to make a sphere around the photons. Since speed is very important, as the density has to be calculated many times, the sphere method is often the method of choice.

Finding the N-nearest photons can be quite time consuming. Therefore, it is desirable to store the photons in a data-structure that makes queries for the N-nearest neighbors easy. A kd-tree is fast to query for the N-nearest neighbors. Therefore the kd-tree is traditionally the preferred data-structure. The fastest kd-tree is the left-balanced kd-tree as it exploits cache-coherence on the CPU

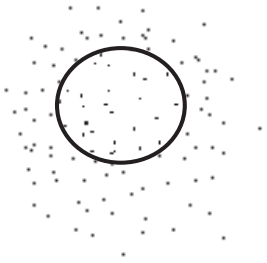

Figure 4.7: Density estimate for the caustic

best ([14], [66], [17]).

4.4 Reconstructing the Caustics

The reconstruction of the caustics is done by using density estimates in the caustics photon map at the current sample point (see Figure 4.7). Caustics are a high frequency effect and many photons are therefore needed to reconstruct the caustics accurately. In order to avoid blurred caustics it can be desirable to use filtering together with the density estimates ([66]). The filtering process is a weighting of the photons where photons near the sample point is weighted higher than photons further away from the sample point.

4.5 Reconstructing the Indirect Illumination

The indirect illumination is calculated using a Monte Carlo based final gathering method (see Chapter 5 for more information on final gathering methods). At the sample point (y), at which the indirect illumination is calculated, a vast number of rays are traced (usually 500-2000) (see Figure 4.8). At each hit location x, the outgoing radiance is found. The outgoing radiance can be calculated by using the irradiance and the BRDF of the surface. The irradiance is found as described above by using density estimation. The outgoing radiance from all these surfaces in the direction of the sample point is then used to find the incoming irradiance at the sample point. This incoming radiance is then used together with the BRDF of the surface of the sample point to find the radiance at the sample point (y). Thus the sample point is only illuminated by the light of all other surfaces and not by the light of the light sources. Therefore it is

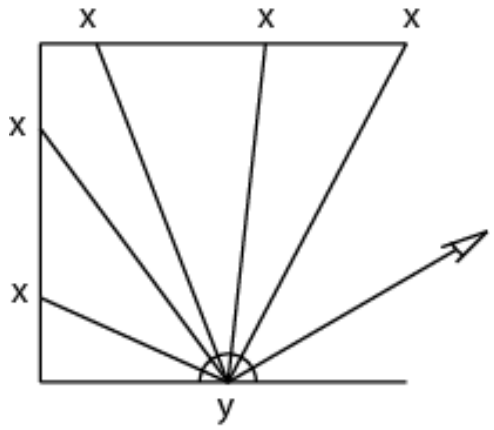

Figure 4.8: Final gather for indirect illumination. A high number of rays are distributed from the point being shaded

exactly the indirect light that is accounted for when one uses this method.

If the indirect illumination is calculated directly by using the photons at the sample location as we do with the caustics, the image will be noisy. The reason for this difference in the reconstruction is that caustics are a high frequency effects while indirect illumination is a low frequency effect.

4.6 Making Photon Mapping Useful

The basic photon mapping method as described above is very powerful but also quite slow. In particular the final gathering step of the indirect illumination is slow. Therefore many methods have been developed to speed up photon mapping. In the following we will describe the most important ones.

4.6.1 Pre-computed Density Estimates

In [23] a method for pre-computing density estimates is presented. When using the photon mapping method, the N-nearest neighbors has to be calculated numerous times during the final gathering pass. For each indirect illumination calculation between 200 and 5000 rays need to be traced, and for each hit point, a density estimate which uses the N-nearest neighbors in the kd-tree has

to be calculated. In total, this gives a very high number of calculated density estimates. Hence it is proposed in [23] that the density estimates should be pre-calculated and stored at the photon positions. When one queries for the density estimate at a given sample point, it is therefore only needed to find the nearest photon and read the pre-calculated density estimate. Furthermore, according to [23], it is sufficient only to store pre-calculated density estimates in every fourth photon. This does not reduce the image quality.

4.6.2 Importance Driven Path Tracing using the Photon Map

In order to lower the number of necessary rays to shoot during the final gathering pass, a technique for only sending rays in important directions is presented in [64]. At the given sample point, a number of nearby photons are found. Then it is examined from which directions these photons originate. This is possible because an incoming direction is stored along with the photons. The hemisphere above the sample point is then divided into a number of cells. The more photons that have arrived through a specific cell the more important is the direction of this cell for the illumination of the sample point. In this way the final gathering sample rays are distributed according to the incoming directions and the work is concentrated where it matters the most. The method described above is called an importance sampling technique (see Chapter 5 for more details on importance sampling techniques and final gathering).

4.6.3 Controlling the Number of Photons

A very interesting question is: How many photons should be distributed from the light source? This is not known beforehand and dependens on the geometry of the scene. A method for controlling the number of photons to distribute is proposed in [119]. Here a number of *importons* are distributed from the eye-point. These importons are stored in a structure similar to the photon map. Many importons in an area indicates that this area is important to the final image. When the photons are distributed, they are directed to the areas where many importons are located, and only few photons are stored in areas where the density of importons are low ([118], [119]).

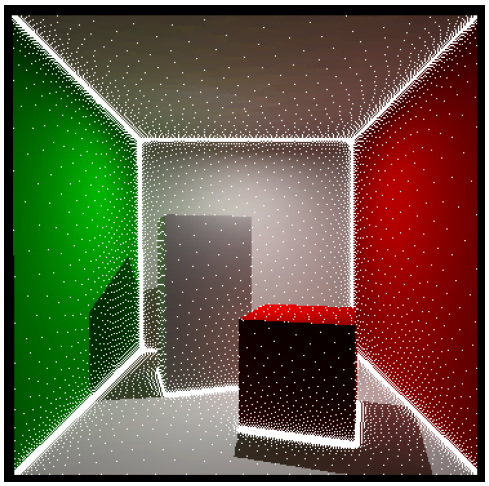

Figure 4.9: Points where the irradiance is calculated when using irradiance caching

4.6.4 Irradiance Caching

The final gathering step when calculating the indirect illumination is the most expensive. One method for speeding up this calculation is to use irradiance caching. The method was initially presented in [141]. The basic idea is that the irradiance only changes slowly and it can therefore be sufficient to only calculate the irradiance at selected locations and interpolate at positions in between these points. Each point which is calculated can be used by nearby points. Whether the calculated value can be used depends on distances to other surfaces. In corners e.g. the distances to other surfaces are small and the area where this calculation can be used is therefore small. If the distance to other surfaces is large, the calculated point can be used further away (see Figure 4.9). The further the distance to other surfaces the larger the radius in which this irradiance is valid. Also the validity or the calculated irradiance is dependent on the normal of the point at which the radius was calculated and on the currently examined point.

The algorithm saves the irradiance values in an octree defined in world space. In Figure 4.9 an image is displayed where the dots mark the points in which the irradiance is calculated. The remaining irradiance values are interpolated.

4.6.5 Irradiance Gradients

In [140] the method is improved by adding the use of gradients. The values needed to calculate the irradiance gradient are already available in the parameters used for calculating the irradiance cache. Thus it is not significantly more costly to calculate the irradiance gradients, and it gives a significant quality improvement.

The quality of the image is very dependent on the distance at which the irradiance values are calculated. This distance is determined partly by the algorithm and partly by a user defined parameter.

The method is further improved in [115]. In this paper it is shown how one can find better positions for calculating the irradiance estimates.

As the irradiance estimates are interpolated, it is only possible to use the irradiance cache for diffuse surfaces.

4.6.6 Photon Mapping and Movie Production

Global illumination is rarely used in movie production. This is mainly due to two reasons. First, global illumination has previously been fairly slow to calculate and secondly global illumination takes away the artistic freedom, since everything has to be physically correct. In [24] it is demonstrated that even though photon mapping is a physically correct calculation method, it can also create images that are tweaked for artistic purposes. Movie scenes are often huge and these scenes does not fit into memory. It is only possible to render these scenes using advanced caching schemes ([26]). The indirect illumination will therefore also be very complex. A method for handling very complex and detailed indirect illumination that does not fit in memory is presented in [25]. The first movie that uses global illumination in large scale is Shrek II ([121]). In this movie photon mapping is not used. Instead a simple one bounce indirect illumination scheme is used. Research in global illumination for movies will probably be a very active area in the near future.

4.7 Discussion

Compared to other methods that solves the rendering equation, photon mapping has several advantages.

Radiosity depends on the calculation of light transfer between all patches (A patch is often identical to a polygon) which means the complexity can be approximated as $O(n^2)$ (Where n is the number of patches). Often the number of patches is increased while calculating the solution in order to increase the accuracy of the illumination ([53]). This makes radiosity a poor choice for very large scenes, e.g. a fractal scene would be very hard or impossible to solve by using radiosity. On the other hand photon mapping is meshing independent as the photons are stored in a separate data structure. In complex scenes, good global illumination can be achieved by using fewer photons than triangles. Furthermore, radiosity is most appropriate for calculating the light transfer between perfectly diffuse surfaces. Radiosity can be combined with ray tracing in order to account for specular surfaces. Be that as it may, radiosity is not suited for calculating light transfer between non-diffuse surfaces. Caustics are furthermore not possible when one uses radiosity. Finally, the complex transport of light interaction with glass surfaces can be impossible to simulate using radiosity.

Monte Carlo path tracing is usually considered the most correct method for calculating global illumination. It is unbiased, and given any scene, it will produce the correct result ([9]). Even though path-tracing will produce the correct result, the time for the solution to converge will often be too long for practical purposes. Therefore, path tracing will often be of a more theoretical interest than of practical use. Caustics are particularly hard for path-tracing to capture accurately. On the other hand, path tracing is a very good tool for verifying whether an image which is calculated by using another method is correct, because even though it will take longer to calculate the image, it is guaranteed that the result will be correct.

Bidirectional path tracing ([76]) is similar to path tracing except that rays are traced both from the light source and from the eye. The advantage of it, compared to traditional path tracing, is that some effects, as for instance caustics, are easier to calculate when tracing from the light source. The problem with bidirectional path tracing is the same as with path tracing, namely that the images are noisy unless enough samples are used.

Chapter 5

The Hemisphere Integral

One of the most important expressions to evaluate in global illumination is the expression that calculates the total incoming light (Irradiance) at a specific point and uses this incoming light to calculate the outgoing light (Radiance). The integral is the following:

$$L(x,\vec{\omega}) = \int_{\Omega} f(x,\vec{\omega}',\vec{\omega})dE(x,\vec{\omega}') = \int_{\Omega} f(x,\vec{\omega}',\vec{\omega})L_i(x,\vec{\omega}')(n\cdot\vec{\omega}')d\omega' \qquad (5.1)$$

This integral integrates over the hemisphere (see Figure 5.1). Only in very rare circumstances is it possible to solve this integral analytically and accordingly this usually not considered a viable option. In the following, we will describe two different strategies of solving this equation. The first is a Monte Carlo method based on ray tracing, while the second is a hardware optimized method based on rasterization.

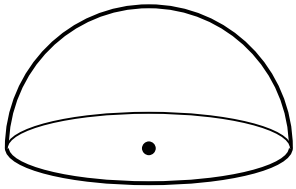

Figure 5.1: The hemisphere.

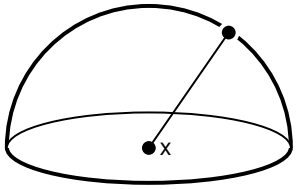

Figure 5.2: A sample on the hemisphere

5.1 Monte Carlo Integration

Monte Carlo integration is usually the default method for solving this integral. When using Monte Carlo integration, a number of samples are evaluated, and the integral is solved by using these samples as an approximation (see [37] for more information on Monte Carlo methods). A sample in our case is a ray which is traced from the sample point x in any direction on the hemisphere (see Figure 5.2).

The most obvious method may be to uniformly distribute the rays on the hemisphere. This is called *blind* Monte Carlo. But often this will not be an optimal strategy. Tracing a ray is expensive and it can therefore be desirable to minimize the number of sample rays needed in order to reach a satisfactory image quality.

By looking at the expression we are integrating, a number of strategies can be used for minimizing the number of rays needed ([37]). All the methods are called importance sampling techniques as the directions of most importance to the final appearance of the point are sampled more intensively. These methods are also called *informed* Monte Carlo. In the following each of the expressions in Equation 5.1 will briefly be described in order to find a more optimal sampling strategy.

$f(x, \vec{\omega}', \vec{\omega})$ This part is the BRDF (as described in Section 2.4). If the BRDF

represents a highly specular surface, it is only a very limited number of directions that contribute to the final appearance of the surface. It is therefore optimal to sample these directions more intensively than other directions. For a diffuse surface, the BRDF is a constant, and in this case all sampling directions will be equally good. But in all other circumstances, some directions are more important to the final appearance than others.

$L_i(x, \vec{\omega}')$ This is the incoming radiance. The incoming radiance usually varies over the hemisphere. It is therefore desirable to sample the brighter directions more intensively than the darker directions. The problem is to determine what directions to sample from before actually sampling. By using photon mapping this is possible as the photons near the sample point x are examined and accordingly their incoming directions indicate what directions the highest light contribution originates from. See [64] for more details.

$n \cdot \vec{\omega}'$ The cosine expression states that light at normal incidence to the surface contributes more to the final appearance of the surface than light almost perpendicular to the surface. It will therefore be desirable to sample orthogonal light more intensively than perpendicular light.

These different strategies can each be used independently or they can be combined as desired.

In order to use importance sampling, a PDF (Probability Density Function) is needed. For more information on using and creating a PDF see [37].

5.2 The Hemi-cube

The hemi-cube is a hardware acceleration technique based on rasterization that can be used to calculate the integral in Equation 5.1 ([28]). As it is not possible to rasterize onto a hemi-sphere, a hemi-cube is used instead (see Figure 5.3). This is described in the following.

The hemi-cube is placed with the sample point x as the center. Then the scene is projected onto each of the 5 surfaces of the cube. This can be done by interpreting each of the 5 surface as image-planes and then setting up a viewing where the eye-point is the center of the hemi-cube and the rest of the viewing parameters are set up to project onto the surfaces of the hemi-cube. The scene is in this way rendered 5 times. The incoming radiance in a specific direction is defined by using the color in the fragment buffer on a given hemi-cube surface.

The area on the hemisphere that a fragment on the top surface of the hemi-cube spans is defined as:

$$A_h = \frac{1}{\pi(x^2 + y^2 + 1)^2} \Delta A_f \tag{5.2}$$

Where A_h is the area of the hemisphere and A_f is the area of the fragment. The (x, y) components are the distances from the center of the hemi-cube.

The area on the hemisphere that a fragment on one of the side surfaces of the hemi-cube spans is defined as:

$$A_h = \frac{z}{\pi(y^2 + z^2 + 1)^2} \Delta A_f \tag{5.3}$$

Again A_h is the area of the hemisphere and A_f is the area of the fragment. The y component is the distance of the plane from the center while z is the height of the fragment from the ground surface.

By using these values and the BRDF it is now possible to calculate the integral in Equation 5.1.

One of the expensive steps in the hemi-cube method is to read the fragments in the image planes back from the frame buffer and to multiply each of these fragment values with the BRDF. The fractional area that each of the fragments span as defined in Equation 11.2 and Equation 5.3. An optimization to this problem will be described in Chapter 11.

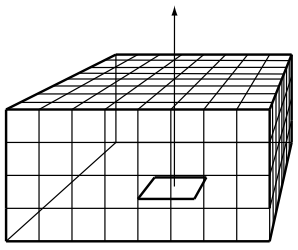

Figure 5.3: The hemi-cube.

5.2.1 Optimizing the Hemi-cube

In [112] a method for calculating the integral over the hemisphere is presented which is related to the hemi-cube method. This method is based on the hemi-cube method but it only utilizes one plane instead of five planes. As this approach does not create a solution which covers the entire hemisphere, the result will not be accurate. But by changing the size of the plane, it is possible to reduce this inaccuracy, even though it is not possible to remove it (see Figure 5.4).

Figure 5.4: Final gathering using a single plane

In [47] an optimized method for calculating an environment map is presented. Instead of using a cubic environment map, a tetrahedron is proposed instead. In this way, the number of image planes for rendering is reduced from six sides to four sides. This tetrahedron map could also be used to integrate over the hemisphere as with a hemi-cube. In this case the number of sides would be three which is two fewer than with the hemi-cube. Compared to the single plane method described above, this method is more accurate as covers the entire hemisphere.

5.3 Discussion

When calculating the hemisphere integral it is not obvious whether to use a Monte Carlo based method or a hemi-cube method. What to choose depends on the nature of the scene and the surface. The hemi-cube samples the hemisphere fairly uniform which in some cases may be desirable, especially if the surfaces are diffuse. Therefore the hemi-cube method is desirable when the surfaces are perfectly diffuse. For a perfectly specular surface the hemi-cube method is of little or no use as it may be that no samples are taken in the exact specular direction. This is because importance sampling does not work well together with the hemi-cube method. Monte Carlo based methods on the other hand are very general approaches that can handle all surfaces effective although in many cases it is not as effective as the hemi-cube method. It can therefore be concluded that neither of these methods are superior in all cases.

Chapter 6

Halton Sequences

Sometimes the purpose of using random numbers is that each number should be equally random, but for other purposes it is more important that the numbers covers an interval uniformly. Halton sequences can be used as random numbers, and although they are deterministic, they cover the interval uniformly. This type of sequence is also called a quasi-Monte-Carlo sequence (QMC).

6.1 Definition of Halton Sequences

In technical terms a Halton sequence is called a reverse radix-based sequence. The radical inverse function is used to obtain a number in the interval [0;1[from an integer number. The radical inverse $\Phi_b(i)$ is the number obtained by expressing i in base b, then reversing the order of the resulting digit sequence, and then placing the floating point at the sequence beginning:

$$\Phi_b(i) = \sum_{j=0}^{\infty} a_j(i) b^{-j-1} \Leftrightarrow i = \sum_{j=0}^{\infty} a_j(i) b^j \tag{6.1}$$

where $a_j(i)$ are subsequent digits of i.

The best way to describe this function is by some simple examples. In base 10 the radical inverse of the number 1234 is 0.4321. This is expressed as $\Phi_{10}(1234) = 0.4321$. A few other examples are $\Phi_{10}(524) = 0.425$, $\Phi_{10}(2) = 0.2$ and $\Phi_{10}(23) = 0.32$.

The base b is the number generating the sequence. The sequence continuously subdivide the open ended unit interval [0;1[. The first b numbers are $\frac{i}{b}$. These b numbers divide the interval into equally sized subintervals. This is called the first level. The second level is the next $b^2 - b^1$ numbers, which divide each subinterval into b new intervals. The next $b^3 - b^2$ numbers are the third interval. These numbers again divide the intervals into new equally sized intervals. There is no limit to the number of possible levels.

The sequence for base 2 is:

$$\{0, \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}, \frac{1}{16}, \frac{3}{16}, \frac{5}{16}...\} \tag{6.2}$$

It is easily observed that a standard Halton sequence is far from random. But the sequence is highly uniform as long as entire levels are used. In order to achieve more randomness in the Halton sequence, techniques such as leaping, scrambling, and shuffling can be used ([135]).

6.2 Multiple Dimensions

When creating numbers in higher dimensions, it is important that the different dimensions are uncorrelated. Only if they are uncorrelated will the numbers will be distributed uniformly in the s-dimensional space $[0; 1[^s$

By choosing prime Halton bases, e.g. base 2 and base 3, uncorrelated 2D points can be generated (see Figure 6.1). It is clear that these numbers are distributed more uniformly than numbers generated by using a traditional random number generator (see Figure 6.2). The numbers in 6.2 are generated by using the *drand48()* function.

An interesting property of Halton sequences is that it is simple to divide the numbers into non overlapping intervals. Using the base 2 and the first level as dividers, the numbers in the levels which are greater than 1 can be divided into the intervals $[0; \frac{1}{2}[$ and $[\frac{1}{2}; 1[$. For base 3, level 1 will divide the remaining numbers into 3 intervals. Using base 2 and base 3 in 2D, 6 non overlapping areas can be created (see Figure 6.3).

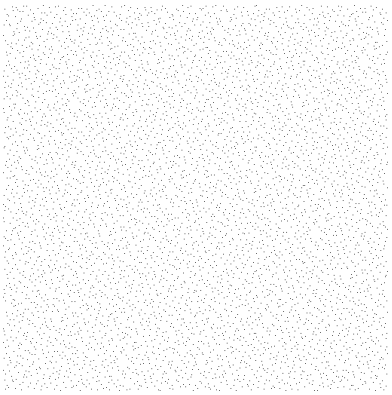

Figure 6.1: Points generated using Halton sequences with base 2 and 3.

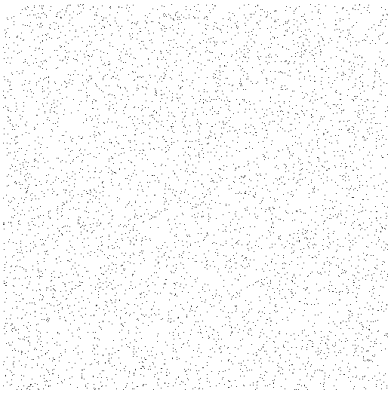

Figure 6.2: Points generated using random numbers.

6.3 Distributing Photons Using Halton Sequences

Distributing photons from a light source is usually done by generating two random numbers and converting these to a direction. From a point light source the conversion will be as follows:

$$\phi = \sqrt{r_2}$$
$$\theta = 2\pi r_1$$

Where ϕ and θ defines the direction while r_1 and r_2 are random numbers in the interval $[0, 1]$.

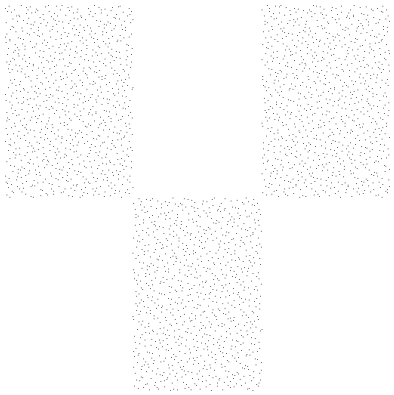

Figure 6.3: A 2D area divided into 6 groups. Half of the groups are omitted.

From an area light source the conversion will be the following:

$$\phi = acos(\sqrt{r_2})$$
$$\theta = 2\pi r_1$$

Halton sequences can be used instead of random numbers. This will cause the photons to be distributed more evenly, and in many cases it will create illumination with less noise.

Another feature when using Halton sequences is that it is simple to divide the photons into a number of groups with the same amount of photons (as demonstrated in Figure 6.3). This feature is exploited in [35]. Here the photons are divided into a number of groups, and when the scene is modified, only photons from selected groups are redistributed. We will introduce a new method for exploiting Halton sequences for redistributing photons in Chapter 9.

Part III

Contributions

Chapter 7

Problem Analysis

In the previous part we have described selected elements that can be used for calculating global illumination. In this part, we will extend and combine these methods in order to develop a method for calculating global illumination for real-time applications. We will base our techniques primarily on photon mapping.

The first obvious option for increasing the speed of photon mapping would be to use more CPUs. This approach works well with ray-tracing as it is easy to parallelize ([132]). The problem with photon mapping is that it is a two pass algorithm and that it queries and updates global data structures. If the photon distribution phase is to be parallelized each of the CPUs should run a program that distributes a number of photons. When this distribution is over the global photon map should be built. But then all the traced photons on each CPU should be distributed to all other CPUs, and then the photon maps should be build on all CPUs as it is not possible to build only selected parts of the photon map. Because of the problems in the photon distribution phase is not directly suited for parallelization. However, a solution for calculating caustics by using photon mapping in a distributed setup has been demonstrated [50]. No such solution has yet been demonstrated for indirect illumination. The second phase of photon mapping is better suited for parallelization. But because of the nature of the first phase of photon mapping, it is currently not possible to approach real-time speed for the indirect illumination just by parallelizing.

Mapping the entire photon mapping method to GPUs, as done in [100], is another option. But as described in the introduction, this approach is currently slower than photon mapping implemented in software, and also produces a poor image quality. We believe the lack of speed and image quality is caused by the difference in nature of the CPU and GPU.

Distributing the photons can either be done instantly or progressively. In [130] and [71] all the photons are distributed each frame. This is possible because only few photons are used. In [35] the photons are distributed progressively. They use a very high number of photons which is necessary because they use the photons for direct visualization of the indirect illumination. If all photons should be distributed each frame it would be very time consuming. Furthermore it would seem wasteful as almost all the photons would follow the exact same path as they did in the last frame. Consequently, we will aim for a solution that exploits frame to frame coherence. On disadvantage of the solution presented in [35] is that it is necessary to store old scene configurations. We will discuss a solution for this in Chapter 9.

Updating indirect illumination can either be done instantaneously or progressively by using frame to frame coherence. In [130] the updates are calculated instantaneously but in their solution they are using a cluster of PCs and the quality of the images are decreased. In [100] the indirect illumination is updated instantaneously but image quality is also low and it is not real-time. Usually the indirect illumination changes slowly in a scene it is therefore an attractive property to exploit. When using this approach the illumination is updated progressively. This is done in e.g. [35] and [122].

The progressive update of the illumination can either based on image space updates or object space updates. Examples of progressive image space updates are [137], [133] and [134]. Examples of progressive object space algorithms are [35] and [122].

It is clear that instant update of indirect illumination will give the highest quality results, but this is computationally very costly.

The Render Cache ([133] and [134]) uses an image space progressive update. As described in the introduction this is a method where pixels in the image plane is progressively updated by using ray tracing. The render cache implementation only includes direct illumination and specular effects. Nevertheless, indirect illumination and caustics could just as well be added. When one moves the image contains a lot of artifacts. This may be a good solution for certain kinds of application e.g. an architectural visualization. In an architectural visualization one wants to move interactively and when the right view is found one may accept to wait for a while to see the detailed image. But for other applications such as

games this will probably never be acceptable.

Comparing progressive updates in image space and in object space we find progressive object space updates to be the most visually pleasing methods. We will therefore strive to create a method that uses object space updates.

In object space the indirect illumination can be stored in either texture maps or in vertices as described in Chapter 1 (In special situation spherical harmonics can also be used ([101] [93]). In [35] the indirect illumination is stored in the vertices. In our solution we want to include final gathering. If we should calculate the final gathering for each vertex or each face it would be computationally demanding. If instead we chose to use textures we can apply a fairly coarse texture to even a detailed mesh and in this way reduce the number of final gatherings. This will of course produce less detailed indirect illumination, but in most cases this will not be significant as indirect illumination usually is a very low frequency function.

In [141] the indirect illumination is also stored in object space in a hierarchical octree. Unfortunately, this type of data structure currently does not map well to GPUs.

Our main idea is to be able to selectively update each of the data structures used in photon mapping. As a first step, we will need to divide the photon map into several photon maps. We describe this process in Chapter 8. This step is necessary step since we want to be able to update only parts of the total illumination stored on the surfaces in the scene. Our focus in that section is not real-time, and we only present it as an optimization that can be used on some scenes. In the following sections, we use this method as one of our main elements in our real-time photon mapping solution.

In [35] a method for selectively distributing photons over a period of time is described. This method uses Quasi Monte Carlo sequences as described in Chapter 6. In Chapter 9 we extend the method from [35] and combine it with the idea of dividing the photon map into several photon maps as introduced in Chapter 8. By combining these methods we achieve the possibility of updating many of the calculations which are used for distributing the light at a fine grained level. The purpose is to exploit the high frame-to-frame coherence and minimize repetitive calculations that produces similar results. Furthermore, we want to retrieve information about where the most significant changes have occurred, because these areas should be updated first.

Together Chapter 8 and 9 deal with the selective distribution of the light while Chapter 10 and 11 describe how to use the distributed energies to calculate the color used for the indirect illumination. We call this the *reconstruction* of the

illumination as it is a step where we only use the already distributed energies. In Chapter 10 we use the energies to calculate an approximation of the illumination by using the distributed energies directly. This is done as described in Chapter 4 but the reconstruction is displayed by using textures. In Chapter 5 we described different methods for calculating the hemi-sphere integral which is a key part of the final gathering step in photon mapping. We described how this integral can both be solved by using ray-tracing and by using rasterization. In Chapter 11 we introduce a method based on rasterization and GPU functionalities for calculating a hemi-sphere integral entirely on the GPU.

Chapter 8, 9, 10 and 11 describe our method for calculating indirect illumination in real-time and the following Chapter 12 describes our method for calculating real-time caustics. Our caustic calculation is based primarily on the photon mapping approach as described in Chapter 4. Our distribution of photons is similar to the original photon mapping method. Traditional photon mapping uses a filter on the photons that is applied for each pixel. We believe this is a good strategy, and therefore we have also implemented a filter on the photons. We are using rasterization for displaying the photons and we use a fragment program for the filtering.

Chapter 8

Using Several Photon Maps for Optimizing Irradiance Calculations

Usually two photon maps are used in the photon mapping algorithm. One for caustics and one for indirect illumination (we ignore participating media). In this chapter we will describe an optimization for calculating the indirect illumination.

The basic idea behind the following optimization suggestion is to divide a large problem into smaller problems and then solve each of these smaller problems individually.

8.1 The Method

In order to calculate a good approximation of the irradiance, a large number of photons are needed. Furthermore, it is important that the nearest photons are used. One exception to that is when the photons are located on a surface that does not point in the same direction as the surface where the irradiance

is calculated. One solution to this problem is to use a disc to find the nearest photons on surfaces pointing in the same direction (see Figure 8.1 and 8.2).

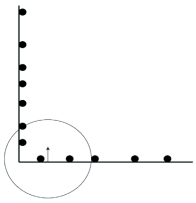

Figure 8.1: Photons from one surface leaking to another surface

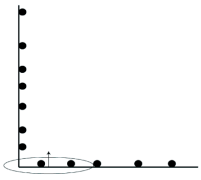

Figure 8.2: Leaking avoided by using a disc instead of a sphere

Another solution is to compare the normal at the photon to the normal of the surface that is examined. If these two normals point in approximately the same direction the photon should be included in the density estimate, otherwise it should be ignored.

This suggests that one could avoid leaking by using different photon maps on adjacent surfaces that have a large angle between them. The important question to answer can therefore be summarized to this: When should two adjacent polygons use the same photon map and when should each of them use a different photon map? The rule we have chosen is the following:

- If the angle between two adjacent polygons is below a predefined threshold (α) they should use the same photon map for storing photons and performing lookups. The shared edge between such two polygons is classified as connected.
- If the angle between two adjacent polygons is above the same predefined threshold (α) they should use different photon maps for storing photons and performing lookups. The shared edge between such two polygons is classified as unconnected.

It is noted that the angle between two polygons is the angle between their normals.

This method is very similar to the way hard and soft edges are found in e.g. VRML. Here a variable called crease-angle is used to specify whether or not the normals of an edge should be interpolated between the two polygons which this edge connects ([13]). In the 3D modelling tool 3D Studio Max, the polygons are classified as belonging to different smoothing groups if the angle between them is above a predefined threshold.

The method for assigning a photon map to a polygon can be described using the following pseudo-code:

1. Mark all edges as either connected or unconnected
2. Assign a unique ID to all polygons
3. If two polygons share a connected edge make their ID identical
4. Create a photon map for each of the remaining ID's

Each polygon is now connected to a photon map and several polygons can share the same photon map. An example of the resulting photon maps from the algorithm can be seen in Figure 8.3 and 8.4.

8.2 Results

The key issue is to answer the question: How big is the advantage of dividing N photons into M photon maps compared to having one photon map with N photons? In an attempt to answer this question, we have made two different

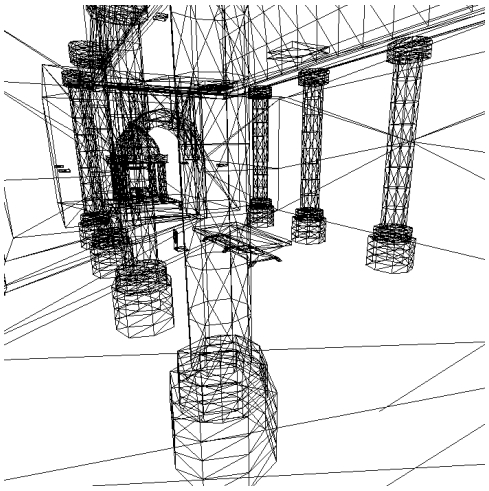

Figure 8.3: A wireframe polygon model of a scene

tests. In the first test we measure the balance time and the time to calculate an irradiance estimate. We perform this test multiple times while using different sizes of the photon maps. For this purpose, we use photons that are distributed randomly in space, although this is unlikely to happen in a real scene. Nevertheless, it will indicate the performance optimization that can be achieved. In the second test we use a simple scene and render it by both using one photon map as usual and several photon maps as we propose. All tests are performed on a P4 1.7 MHz Dell Portable with 512 Kb level 2 Cache. The code used to calculate the irradiance is the code made available in [66]. The results of the first test can be seen in Figure 8.5 and Table 8.1. In Figure 8.6 the balance time per photon is measured against the number of photons in the photon map. Balancing a binary tree is done in $O(nlog(n))$ time. It is therefore expected that the growth in the diagram is constant when time is drawn logarithmically. In our implementation, each photon uses 40 bytes and with a 512 Kb level 2 cache there is room for approximately 13.000 photons in this cache. We believe this is the reason for the different appearances of the graph before and after 13.000 photons.

In Figure 8.5 the cost of calculating an irradiance estimate is shown as a function of the number of photons in the photon map. The savings are not as significant as when we balance the photons, but, nevertheless, a few hundred percent can still be saved by creating smaller photon maps.

In Table 8.2 we took N photons and divided these photons into M photon maps.

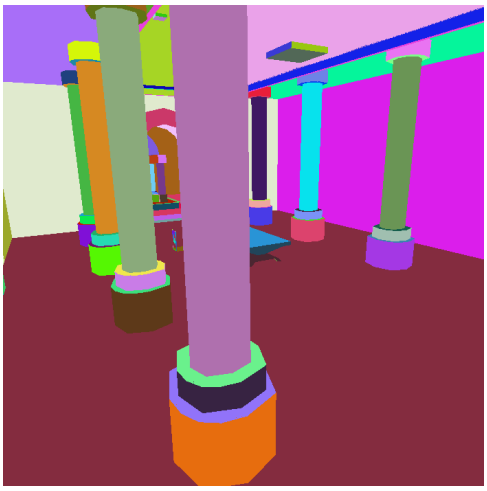

Figure 8.4: All polygons marked by a color from the photon map they refer to

Photons (N)	Maps (M) each with N/M photons	Balancing the M photon maps	Lookup for 500 nearest photons
1000000	1	5.05 s	332 μs
1000000	10	3.08 s	291 μs
1000000	100	0.70 s	213 μs
1000000	1000	0.55 s	116 μs

Table 8.1: Comparison of photon map lookups using different amounts of photon maps for storing the same amount photons. The photon positions were generated randomly.

All these results were found by creating 100 completely random photon maps and then averaging the timings. In the second test we rendered the same scene by using one photon map (as usual with the photon mapping method) (see Figure 8.7) and by using several photon maps as we propose (see Figure 8.8). As expected there is no visible difference between the two images. In the scenes displayed in figure 8.3 and 8.4 the timing difference for one and several photon maps is substantial (see Table 8.2). 50.000 photons were used in this experiment.

It is clear from our results that it is an advantage to distribute the photons into several photon maps.

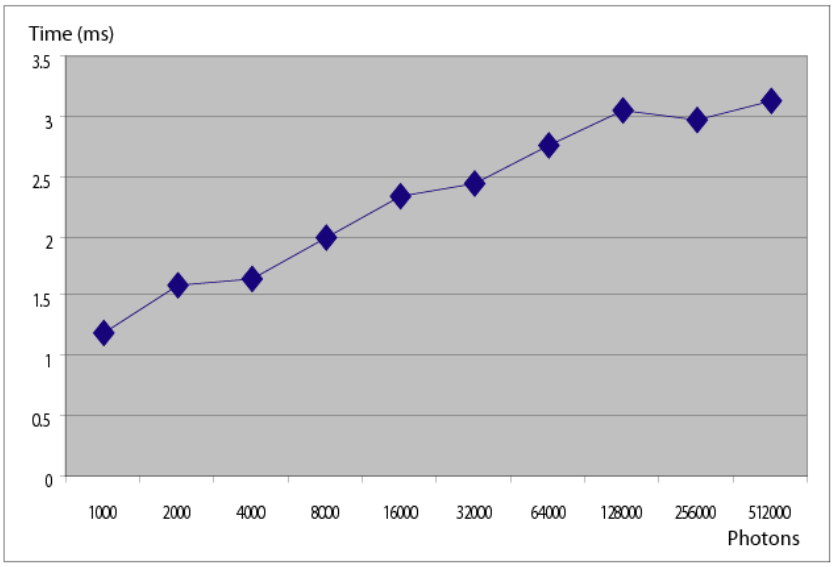

Figure 8.5: Cost of calculating an irradiance estimate using 500 photons (in ms) using photon maps of different sizes

	Balancing the photon map(s)	Precalculating the irradiance estimates
One photon map	0.56 s	13.80 s
Several photon maps	0.08 s	5.88 s

Table 8.2: Comparison of timings for precalculating irradiance estimates for the images in figure 8.3 and 8.4

8.3 Discussion

Although our proposed solution results in a speedup, the method does not apply well to all scenes. Using several photon maps instead of one should be done with care. A scene with many small triangles and sharp angles between these will not be a good candidate for this optimization as the angles between the triangles will no longer be a good measure for when to split the photon map. This will typically be the case in scenes generated by using fractal algorithms. In general, it is important to have a significant amount of photons in each photon map. If dividing the photon map into multiple photon maps violates this property, it is not a good idea to split up the photon maps. The photon map is created by using a left balanced photon map. If photons are added or removed from the

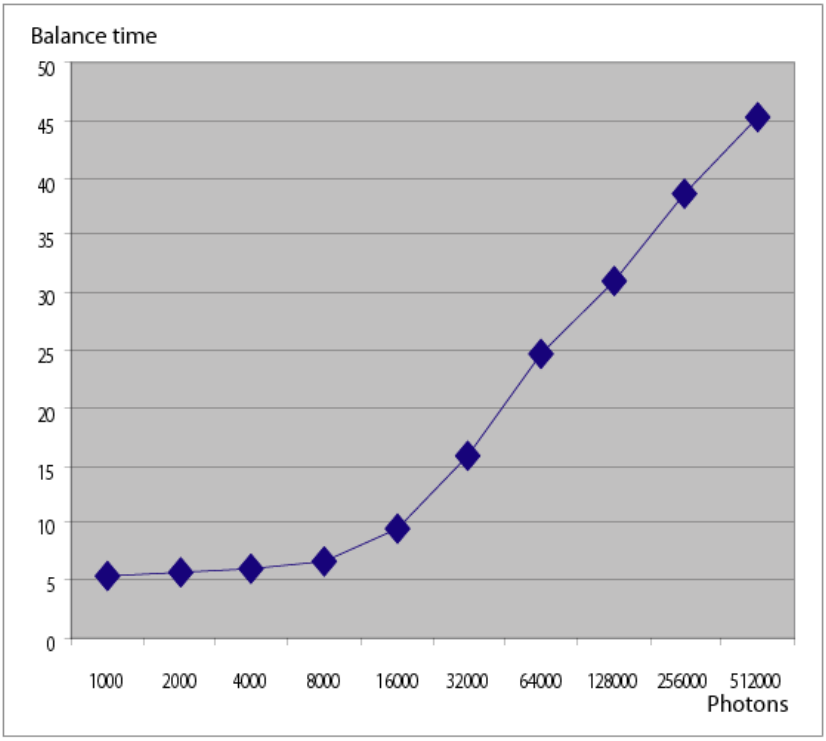

Figure 8.6: Balance time per photon (in μs) as a function of number of photons in the photon map

data structure, it is necessary to rebalance the entire tree. But if the photon map is split up into several photon maps, it is only necessary to update the photon maps that have been modified. This property can be very useful in animations in which only some parts of the scene are modified from frame to frame. Furthermore, if the irradiance has been precalculated as described in [23], this precalculation can also be reused. In addition, by dividing the photon maps into several photon maps, as we suggest, it is no longer necessary to store the additional normal introduced by [23], although removing this normal can only be done if the surfaces are perfectly diffuse. But this is also an assumption which is often seen, e.g. irradiance caching ([141]) only works with perfectly diffuse surfaces.

Another problem is how much memory to reserve when initializing the photon maps. In general, it is not known how many photons to store as it depends on the

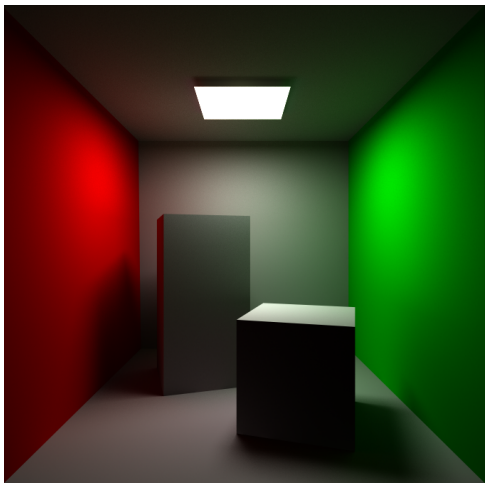

Figure 8.7: Calculated by using one photon map

path that the photons take in the environment. For each time a photon bounces using the random russian roulette algorithm, one more photon is stored in the photon map. A photon distributed from the light source can therefore bounce many times if the surfaces have a high reflectance, or it can just disappear if it does not intersect any surface or is terminated (determined by the Russian Roulette algorithm).

This problem can be solved in two ways. Either the memory is allocated dynamically as more memory is needed, or a fixed amount of memory is set aside. The approach of using dynamically allocated memory is usually the easiest when implementing, while allocating a fixed memory pool is the fastest computationally as memory defragmentation is avoided.

In our implementation we use the fixed size photon maps. We set aside a *sufficiently* large memory area and then we calculate how much memory each photon map can be granted. This calculation is based on the summed area of the polygons in the photon map divided by the summed area of all the polygons in the scene. This approach may be inefficient if the scene is large and all the photons hit some few bright areas while the rest of the scene is dark.

To summarize: The advantages of having several photon maps are

- Faster irradiance calculations
- Faster balancing of the photon maps

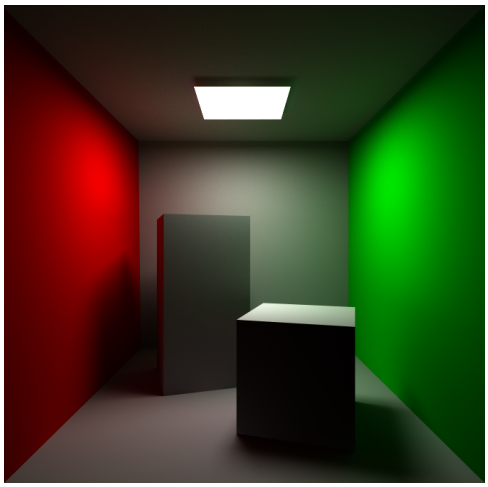

Figure 8.8: Calculated by using several photon maps

- No leaking problems in corners
- It may be possible to update a limited number of photon maps when creating animations

Disadvantage of having several photon maps

- Connectivity has to be calculated
- A scheme for memory allocation for the photon maps has to be chosen
- It does not apply to all scenes

To use our method on the caustics and volume photon maps from the photon mapping algorithm is not as easy as using indirect illumination. This is because it is difficult to figure out when to split the photon maps. But if the problem of figuring out when to split the photon maps is solved, then our optimization applies to them as well.

Chapter 9

Selective Photon Emission

In traditional photon mapping as described in Section 4 all photons are traced in a first pass. But tracing all photons each frame in a real-time application is too computationally expensive (at least when using only one CPU). The selective photon tracing introduced in [35] solves this problem in some ways as only intelligently selected photons are re-emitted each frame. They give each photon a fixed initial direction using QMC Halton sequences, and the photons from the light source are divided into groups where the initial direction of each photon is similar. Each group contains an equal amount of photons with equal energies. The photons are traced on the CPU. We find this to be an attractive strategy, but we distribute the photons a bit differently. We do this to avoid the weaknesses of the method which are described in the introduction (1.8.3).

9.1 The Method

We enumerate the photons in each group, and for each frame only one photon from the group is traced. In the next frame, a new photon from the same group is selected. This is done in a Round-Robin fashion. The path of each photon is stored, and if the new path diverges from the previous path, all photons from the group will be marked for redistribution. In this way, more effort is spent in areas where the scene is modified. It is also guaranteed that minor changes will

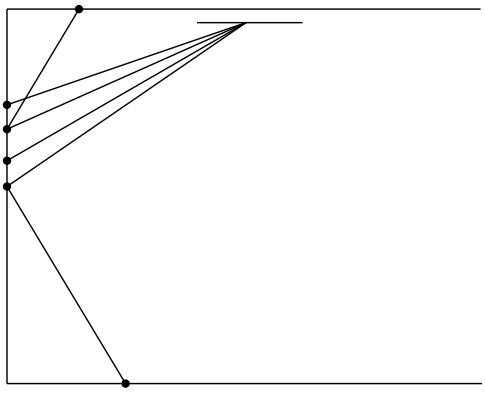

Figure 9.1: Photons distributed from a single group. The initial directions are similar.

eventually be seen by a photon. It may nevertheless take more time to discover minor changes than major changes. This is the case as only few photons from the group may be invalidated and the group may act for a longer time as if no changes have occured in its domain. Major changes will be registered faster as many or all photons from a particular group will be invalidated. Photon bounces are handled by using Russian Roulette ([66]).

The photons are stored as *photon-hits* on the surfaces along the path of the traced photon. The complete path of each photon is also stored. In this way, it is easy to remove the photon-hits from the scene in constant time if the path of the photon is invalidated. It is also faster to determine whether the photon path has been invalidated. Each surface also has pointers to the photon-hits that have hit that particular surface, making it faster to determine the total amount of photon energies on a surface. The extra storage needed per photon is an energy amount for each hit, and a pointer from the photon-hit to the surface, and a pointer from the surface to the photon-hit. The average length of a path is fairly short when using Russian Roulette. The memory overhead for storing the photon path is therefore not substantial.

As a result of our chosen strategy of storing the photon paths, a moving light source will cause all photon paths from this particular light to be invalidated each frame.

In Figure 9.1 an example of how photons are distributed from a single group is shown. In Figure 9.2 a screen shot of this situation is shown. In Figure 9.4 all primary photons which are distributed in one frame are shown. It is noted that for each frame, new primary photons are traced. All photons which are

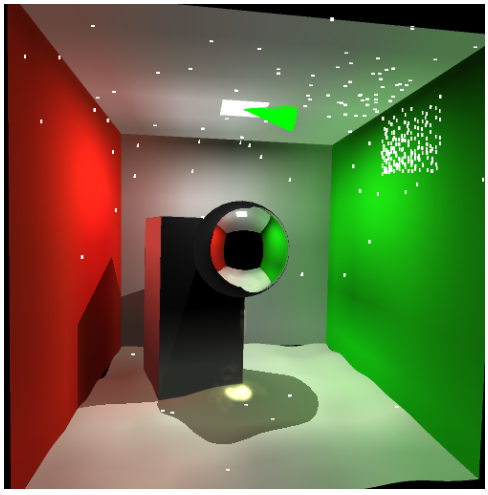

Figure 9.2: Photons from one group

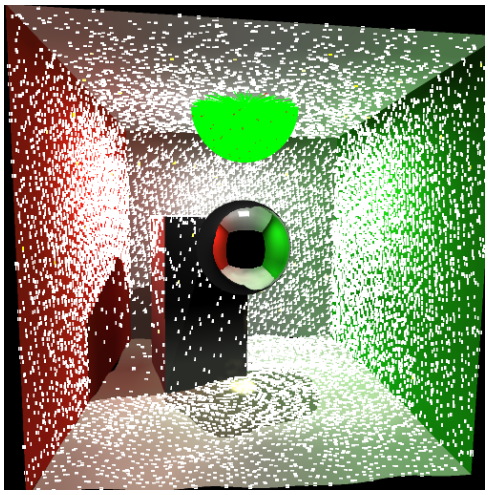

Figure 9.3: All photons distributed in the scene

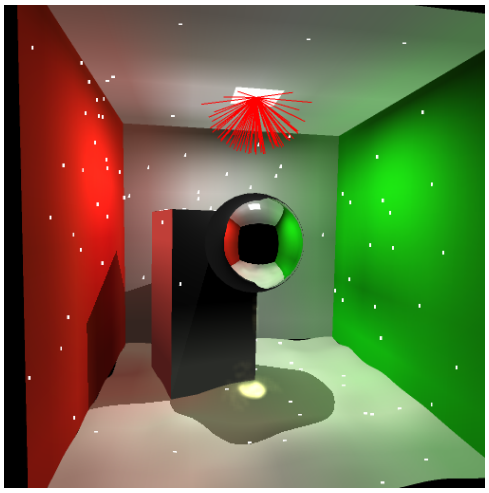

Figure 9.4: All primary photons

distributed are shown in Figure 9.3.

9.2 Discussion

As each light source has a fixed number of photon groups, it may be expensive to trace a single photon from each group in every frame if many light sources are present in the scene. It may therefore be desirable to trace fewer photons from each light source per frame. Whether to trace one photon from each photon group for each frame or to reduce this number can e.g. be determined by looking at the distance from the light source to the viewer.

Chapter 10

Approximated Reconstruction of the Full Illumination

The photons distributed as described in the previous section carry information about the full illumination in the scene. These photons are used for reconstructing an approximation of the full illumination. This process is described in the following.

If enough photons are distributed, the photons will represent the correct illumination in the scene, and they can be calculated directly by using density estimation ([110]). The problem is that "enough" photons are many millions and it is almost impossible to completely remove the noise. Therefore, the photons are used to reconstruct an approximation of the full illumination, and then this approximation is used in the final gathering step to calculate a smooth and accurate approximation of the indirect illumination. The number of photons needed when performing the final gathering is many times smaller than the photons needed for density estimation ([66]).

10.1 The Method

In order to approximate the full illumination we compute the irradiance using an N-nearest neighbor query of the photons, and the total energy of these photons is divided by the area that they span. The radiance of the surface is stored in reconstruction texture maps applied to the surfaces. We call these texture maps approximated illumination maps (AIMs).

Kd-trees are fast for N-nearest neighbor queries. In order to build the kd-tree, only photons that are located on surfaces which could possibly contribute to the irradiance of the current surface are considered. We use a technique similar to the one presented in [78]. In [78] the division into surfaces is done automatically, however in our implementation we have performed the division by hand by using a 3D modelling tool. When a photon hits a surface, it is not stored in a global data structure but in a structure local to the surface that was hit (see the discussion for an analysis of this choice). A surface can contain an arbitrary number of polygons. Figure 10.1 shows a Cornell box with a bumpy floor and unique colors for each surface.

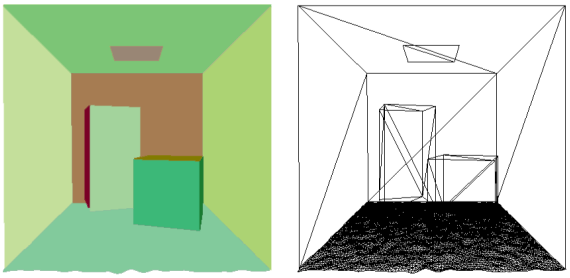

Figure 10.1: Left: A unique color is displayed for each surface, Right: Same scene shown using wireframe (13.000 polygons and 16 surfaces

Updating all the AIMs for each frame is computationally expensive and undesirable. Consequently, we use a delta value for each surface to control when its AIM should be updated. The value (Δ_f) is a delta value for the full approximated illumination. The Δ_f value is affected by the energy of any photon that is removed or stored on a surface. Only when Δ_f is larger than a small threshold value, the AIM should be updated.

In practice it can be necessary to limit the amount of work done per frame. In our implementation we first handle surfaces with high $\frac{\Delta_f}{A_s}$, where A_s is the area of the surface. We only update a limited amount of surfaces per frame.

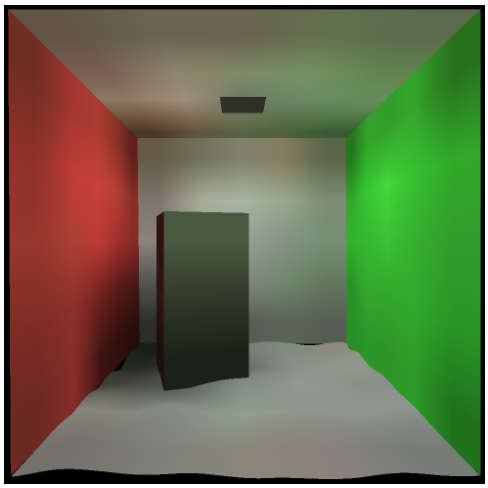

Figure 10.2: The full approximated illumination

As only one photon map from a single surface is utilized at any point in time, it is only necessary to create a single photon map in the memory. This photon map only need to store the photons of a single surface. This makes the memory requirement smaller than if one global photon map containing all the photons had been used.

In Figure 10.2 the full illumination using the approximated textures (AIMs) is shown.

Chapter 11

Indirect Illumination using Hardware Optimized Final Gathering

As described in Chapter 5 a hardware optimized final gathering method was introduced in [28], namely the hemi-cube. This method needs to render the scene 5 times for each resulting *final gathering value*. The 5 renderings are one for each side of the hemi-cube. The front side of the hemi-cube contributes with about 56% of the incoming directions, while each of the side planes each contribute with about 11% of the incoming directions. In [112] a method is introduced which only uses the front plane of the hemi-cube and then enlarges this front plane. In this way, a more accurate solution can be achieved, as we only use the front plane. If e.g. the front side is enlarged to double side length, thus making the total area 4 times larger, this accounts for about 86% of the total incoming directions (see Figure 5.4).

11.1 The Method

We use the hemi-plane method for the final gathering step. The scene is rendered to the hemi-plane by using the full approximated illumination from the photons. This is achieved by rendering the scene with the AIMs. Each pixel of this rendering must be multiplied with the fraction of the hemisphere that it spans and a cosine factor in order to calculate the irradiance. The irradiance is defined as:

$$E = \int_{\Omega} L(\omega')(n \cdot \omega')d\omega \tag{11.1}$$

where E is the irradiance, L is the incoming radiance, n is the normal, ω is the direction of the incoming light, and $d\omega$ is the solid angle.

This expression can be approximated in the following way.

The cosine weighted area (This corresponds to $(n \cdot \omega')d\omega$ in Equation 11.1) on the hemisphere which a fragment on the rendered surface of the rendered plane spans is defined as:

$$F_h(x, y) = \frac{1}{\pi(x^2 + y^2 + 1)^2}\Delta A_f \tag{11.2}$$

where F_h is the cosine weighted area of the hemisphere and ΔA_f is the area of a fragment. The (x, y) components are the distances from the center of the plane([28]). The irradiance can then be approximated as:

$$E \approx \sum_{x,y} p(x, y)F_h(x, y) \tag{11.3}$$

where $p(x, y)$ is the pixel value of the rendered plane. The irradiance value should be divided by the percentage of the hemisphere which the rendered plane spans in order to compensate for the missing areas.

The calculation above can be implemented on the GPU. The way we have implemented this is first to render the scene to a pixel buffer (pbuffer). Then this pbuffer is used as input to a fragment program which multiplies each pixel with a value calculated by using Equation 11.2. Then the resulting summation is calculated by using hardware MIP map functionality (we use the OpenGL extension SGIS_generate_mipmap). The MIP map function calculates the average of a texture in the topmost level of the MIP map. Therefore we multiply each pixel by the total number of rendered pixels. This is performed in the fragment

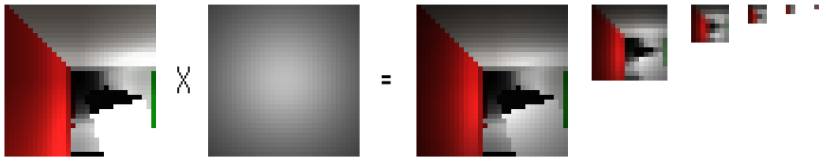

Figure 11.1: First, the hemi-plane rendering is multiplied with the area weighted cosine term. The final irradiance value is calculated by using MIP mapping.

program which is run before the MIP map is executed (see Figure 11.1). We copy this final pixel to a texture that is applied to the surface in the scene. We call this texture the indirect illumination map (IIM) (In our implementation they have the same resolution as the AIMs). All the steps are executed on the GPU, and no expensive read-back to the CPU is necessary.

Calculating the radiance for the indirect illumination can be done in several ways. If a texture is applied to the surface, the irradiance should be stored in the IIM and multiplied with the texture during rendering. But if the surface has a uniform color, the radiance could just as well be stored directly in the IIM. This can be done by pre-multiplying the irradiance values with the reflectance of the surface in the fragment program.

Displaying the illumination is often done in real-time applications by using texture maps which are also called light maps. Light maps usually contain both direct and indirect illumination and they are often coarse. Since the indirect illumination usually changes slowly over a surface it is possible to use an even coarser texture.

It is noted that when we use this approach, only diffuse reconstruction of the indirect illumination can be handled.

Many techniques can be used for applying a texture with uniform size to a model, and several full automatic methods have been proposed ([45]). In our implementation, we have applied the textures manually by using a 3D modelling tool.

As with the AIMs, it is computationally expensive to update all the IIMs for each frame. Therefore, we introduce Δ_i which is the delta value for the indirect illumination. This value is similar to Δ_f except that it is only affected by photons that have bounced at least once. As with the AIMs the surface with the highest Δ_i will have its IIM updated first.

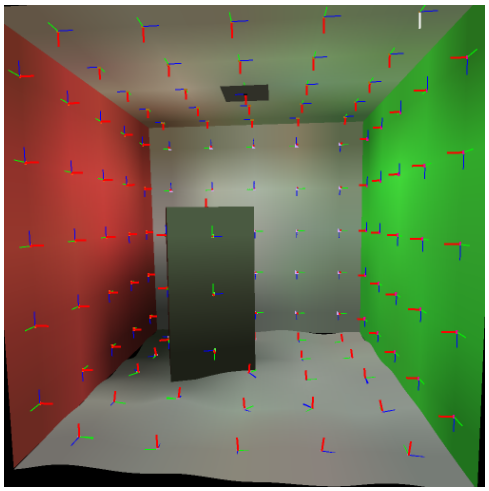

Figure 11.2: The gather coordinates

In our implementation, we restrict the number of texels which can be updated per frame in order to keep a high frame rate.

We use two textures for the IIMs. One that is visualized and one that is being updated (double buffering). When the update of a texture is done, the two textures are switched using a blend between the textures. This is done in order to avoid popping. But for the indirect illumination to be updated this is a trade-off between popping and lag. We have therefore set the blend function to be fairly fast. Whether a quick or a slow blend should be used depends on the application.

In Figure 11.2 the centers of the GPU based gathering is shown. These are the centers of the textures.

11.2 Discussion

The final gathering method renders the entire visible scene using the center of the texel that should be updated as the camera point. Since all pixels of this rendering are averaged using MIP mapping before they are used, it is not necessary to render an extremely accurate image, and the lowest level of detail for all objects in the scene might as well be chosen. Culling algorithms should of course also be enabled using the final gathering renderings.

We have implemented the summation of all pixels in the final gathering step using hardware MIP mapping. Currently this can only be done using 8 bit precision. In the next generation of GPUs it is likely that this can be done using floating point precision. Another option would be to calculate the MIP mapping using fragment programs which is done in e.g. [93].

Aliasing is usually a problem when using hemi-cube based methods. To illustrate that aliasing is not an issue one can imagine an AIM with infinitely high resolution. The rendering to the hemi-plane will in this case correspond exactly to an N-nearest query lookup in the photon map. We use the hemi-cube for gathering radiance values from textures with large texels. The large texels is a filtering of the radiances which will reduce an eventual aliasing problem. Further it is cheap to increase the size of the hemi-cube (see Table 14.2).

Chapter 12

Hardware Optimized Real-time Caustics

Caustics arise when a photon hits a diffuse surface after having been specularly reflected one or several times directly from the light source. When using photon mapping, caustic photons are traced in the same way as with the photons used for the indirect illumination. But photons are only traced in directions where known caustics generators (specular reflectors) are located ([66]) (It should be noted that this implies that caustics caused by indirect illumination can not be captured). When the photon hits a diffuse surface, its location is stored. Our method is based on this strategy, which means that we do not have the limitations of the real-time caustic method described in [136].

12.1 The Method

We distribute the photons evenly by using QMC Halton sequences in order to lower the noise and avoid flickering. The photons are traced by using a standard CPU ray tracer. We store the photons in a simple list. We do not divide the photons into groups as with the indirect illumination because caustics are very localized.

In order to reconstruct the caustics, we do the following. First we draw the scene by using the color black to a pbuffer with the depth buffer enabled. This is done in order to get the right occlusion of the photons. Then all the photons are drawn additively as points by using blending to this pbuffer. Afterwards the pbuffer contains a count of the photons that have hit a particular pixel. This pbuffer is used as a texture, which makes it possible for a fragment program to read the color value of the current pixel from the previous rendering. A screen size polygon with this texture is therefore drawn by using a fragment program.

Furthermore it is also possible to read the photon count. Based on the photon count of the nearby pixels, a filter is applied to the pixels:

$$c(x,y) = s \sum_{i=-k}^{k} \sum_{j=-k}^{k} t(x+i, y+j)\sqrt{1+2k^2-(i^2+j^2)} \qquad (12.1)$$

where $c(x,y)$ is the resulting color at position (x,y) and $t(x,y)$ is the texture value at position (x,y). s is a scaling value that adjusts power of the photon energies. We use a filter of size $7x7$ (i.e. $k=3$).

By using this method, it is possible to count 255 photons (8 bits) at each pixel, and this will be sufficient in most cases. When counting the photons in the framebuffer it is assumed that all lights and all caustics generators have the same color, otherwise a floating point pbuffer is needed.

This is a screen space filtering while photon mapping traditionally uses filtering in world space. As a result the caustics are by no means physically correct. If one zooms in on the caustic fewer photons will be filtered and consequently the caustic will look less intense. If one zooms out or just changes the angle at which the surface with the caustic is viewed, more photons will be filtered and consequently the caustic will look brighter.

In order to solve this problem we want to scale the intensity of the caustic to make it appear equally bright when the angle is changed and when one zooms in and out. We have chosen to calculate the world space area size of a screen space pixel and scale the intensity of the caustic by using this value. The area of a pixel in world space can be described as follows:

$$A = (n_s \cdot n_e)(4d^2 \tan(\frac{f_x}{2p_x}) \tan(\frac{f_y}{2p_y})) \qquad (12.2)$$

Where A is the world size area of the pixel. f_x and f_y are the field of view in the x and y direction. p_x and p_y are the number of pixels in the x and y direction.

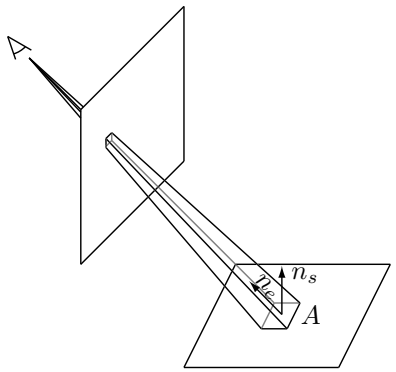

Figure 12.1: World size area of a pixel.

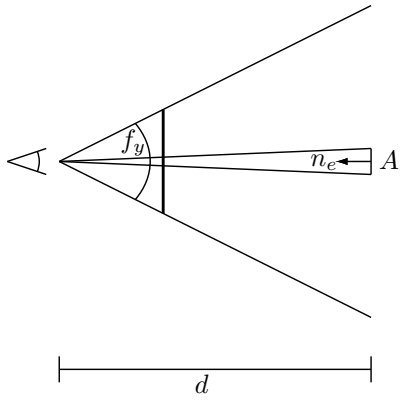

Figure 12.2: World size area of a pixel seen from the side.

d is the distance from the eye point to the pixel (or the geometry that the pixel represents). n_e is the direction from the pixel toward to eye. n_s is the normal of the surface at the pixel (See Figure 12.1 and Figure 12.2).

The process is as follows: First the depth of the scene is rendered to one component in a floating point pbuffer (e.g. the red component). Simultaneously, the dot product between the n_e and n_s is rendered to another component in the floating point pbuffer (e.g. the green component). The fragment program that filters the caustic photons is then given this floating point pbuffer as input. By using this approach it is possible for the fragment program to have access to the current depth value and the dot product and as a result it is possible to evaluate Equation 12.2 in the fragment program. In the following we will discuss some of the implementation details.

Calculating the screen space position (in the canonical view frustum) of a vertex is done as follows:

$$p_s = M_p p_w \tag{12.3}$$

Where p_s is the screen space position in homogeneous coordinates, M_p is the model view projection matrix and p_w is the world space position of the vertex. This calculation is performed in a vertex program. In a fragment program the z value that is written to the floating point pbuffer is calculated as follows:

$$z = \frac{\frac{p_s.z}{p_s.w} + 1}{2} \tag{12.4}$$

Where z is the interval $[0; 1]$.

In the fragment program which filters the caustic the z value of the current pixel is read from the pbuffer. The d value can then be calculated from the z value by using the following formula:

$$d = -\frac{n * f}{z * (f - n) - f} \tag{12.5}$$

Where n is the near plane and f is the far plane in the current projection matrix.

The surface normal n_s can be found by multiplying the model matrix with the normal of the object, while n_e is found by creating a normalized vector from p_s to the eye point.

By using the approach described above all variables in Equation 12.2 are now available in the fragment program. Nevertheless, this approach is still more inaccurate than the the traditional nearest neighbor approach. By using our approach a fixed search radius is used. When one zooms in very close the radius will be too small and too few photons will be included. As a result the caustic will become inaccurate. When one zooms out the radius will be too large and the photons will not span the radius and the intensity will become too low. However, weighting the intensity of the caustic with the area of a pixel makes the caustic more physically correct and does improve the image quality (see Figure 14.6).

Tracing all photons in every frame may not be necessary. If the application runs at e.g. 30 fps, it may not be notable if each photon is only retraced every second to every tenth frame. If the specular object is moved fast, it will be possible to see a trail after the specular object. Whether this is visually acceptable depends

on the application. In our experience, the delayed photon distribution does not disturb the visual appearance when the frame rate is high, and if the object moves slowly it is hard to notice.

The implementation of equation 12.1 is a time consuming fragment program. When using a filter of size $7x7$, the summations are unrolled to a program that has about 400 lines of assembly code and 49 lookups in the texture. Therefore it is desirable to limit the use of the fragment program to areas where caustic photons are actually present. This is done by using the following method.

Before the photons are drawn to the pbuffer, stenciling is enabled. The stencil is set to increment on zpass. When the photons are drawn, they are also marked in the stencil buffer. Then the screen is divided into a number of grid cells. For each grid cell, an occlusion query is started as a quad is drawn. The stencil function is set only to let pixel be written to the pbuffer if the stencil value in the pbuffer is greater than zero. When the quad is drawn, the occlusion query will return the number of pixels that were modified. If no pixels were modified, no photons need to be filtered in this area. In this way, the inexpensive occlusion query can identify the areas that have caustic photons. Often the caustic only fills a few percent of the screen. The process is illustrated in Figure 12.3. The process is similar to the process described in [36].

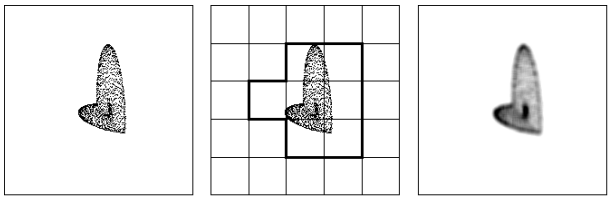

Figure 12.3: Left: photons are drawn on the screen. Middle: Areas on the screen are tested for photons. Right: Areas are filtered in screen space.

Chapter 13

Combining the Contributions

For the direct illumination we can chose either to use ray tracing or rasterization. In Chapter 3 we made a simple comparison between ray tracing and rasterization. We concluded that in many situations the result will be visually similar. Nevertheless, rasterization will in many situations be faster. In our solution we will therefore use rasterization for the calculating the direct illumination.

We use stencil buffer shadow volumes for calculating the shadows. Our implementation uses hard shadows but real time hardware rendered soft shadows could just as well have been used ([63], [11], [54]). Any soft shadow algorithm is equally well suited as the shadow is rendered to a separate pbuffer. When combining the light contributions the shadow is applied by using the shadow pbuffer as a screen size texture.

We use a dynamic environment map for specular reflections. This is done by using a cube-map. For each frame the scene is rendered six times in order to update the sides of the cube. Multiple interreflections could have been used as well ([92]).

Combining the various contributions is an additive process. We create a separate pbuffer for the shadows and another for the caustics. When the scene is rendered, the direct illumination is calculated for each pixel and multiplied by the content in the shadow pbuffer. The texture value for the indirect illumina-

tion is sampled in the IIMs which are applied to each surface. These values are added to the final color along with the caustic's value (see Figure 13.1). In this way, we combine the contributions in a final pass.

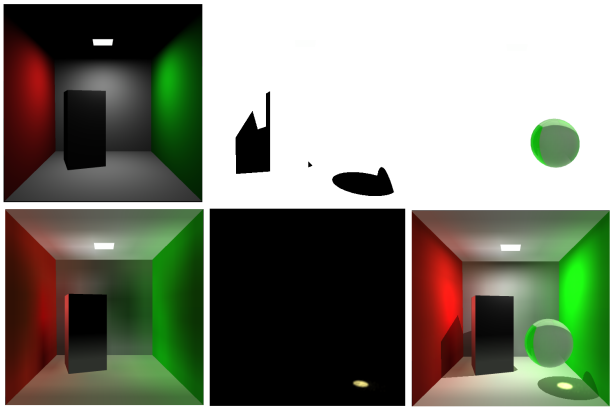

Figure 13.1: Top left: The direct illumination without shadows. Top middle: Shadows. Top right: Reflections. Bottom left: Indirect illumination. Bottom middle: Caustics. Bottom right: Complete illumination

The formula is as follows:

$$L = L_{indirect} + L_{caustics} + L_{specular} + L_{direct} * shadow \tag{13.1}$$

When several lights are present in the scene the formula is as follows:

$$L = L_{indirect} + L_{caustics} + L_{specular} + \sum_{i=0}^{lights} L_{direct}(i) * shadow(i) \tag{13.2}$$

Chapter 14

Results

We have implemented our application on a Pentium 4, 2.4 Ghz with a GeForceFx 5950 graphics card running Windows. All code has been written in C++ and compiled by using Visual Studio 6. Cg was used for all vertex and fragment programs ([86]). Our photon-tracer utilizes a standard axis-aligned BSP-tree build by using a cost function based on surface areas ([55], [120]). It is reasonably fast. Even so, it is not as fast as the very optimized ray tracer used in [125].

Each dynamic object in the scenes uses a separate BSP-tree. Any photon traced in the scene is therefore tested for intersection with all BSP-trees.

The scene in Figure 13.1 with indirect illumination and caustics runs a 35+ fps. The cube and the sphere are dynamic objects. 10000 photons are used for the caustics, and they are completely updated over 8 frames. For the indirect illumination, 77 photon groups are used each with 40 photons. A maximum of 20 texels per frame are updated by using final gathering. The big surfaces on the walls have textures of 5 by 5 texels for both the AIM and IIM. In total, the scene has 140 texels for the AIMs and similarly 140 texels for the IIMs. The scene is rendered at a resolution of 512 by 512 pixels and all pbuffers are also 512 by 512 pixels. The environment map is rendered as a cube-map and the scene is rendered 6 times per frame. Each side in the cube-map has a resolution of 128 by 128 pixels. The memory used for storing the photons and their paths is in this case the number of photons multiplied by 3 floats for the energies and

the size of two pointers multiplied by the average path length, which in our case is approximately 2. This gives a total memory requirement of approximately 120 Kb.

If we consider two unconnected rooms, and modifications only occur in one of the rooms, then no updates will be necessary in the other room. Only minimal computational power will be spent in the other room as no photons will be invalidated. This case is shown in the top-most scene in Figure 14.1. When the rooms are connected, updates made in one of the rooms will now affect the illumination in both rooms (see middle image in Figure 14.1). The bottom-most image in Figure 14.1 shows a scene, where the right room is illuminated primarily by indirect illumination.

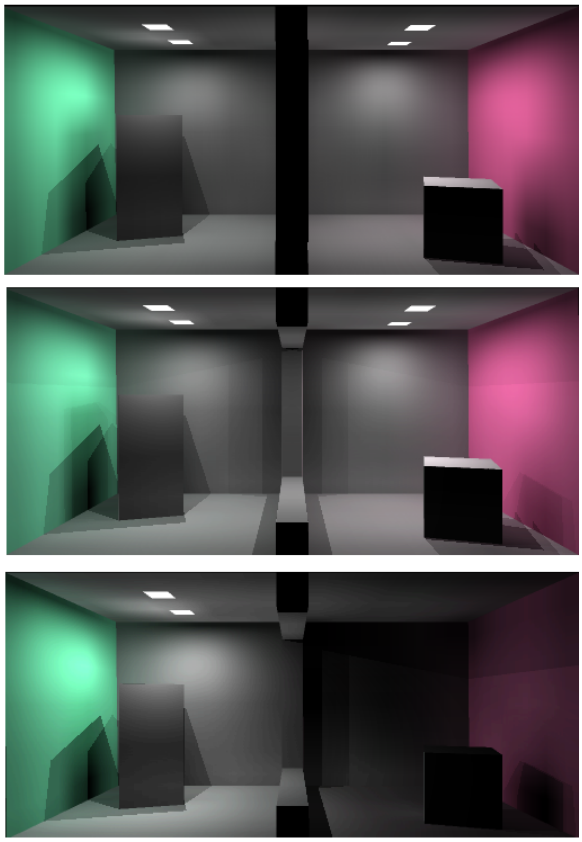

Figure 14.1: Top: A scene with two divided rooms each with two light sources. Middle: The rooms have been connected. Bottom: Two of the light sources have been turned off.

Photons	Irradiance lookup (100 photons)	Balancing time
300	0.023 ms	0.10 ms
500	0.027 ms	0.18 ms
1000	0.029 ms	0.39 ms
2000	0.034 ms	0.84 ms

Table 14.1: Timings for balancing a kd-tree with photons

Render size	Fragments	Polygons in scene	Timings
8	8x8	34	0.75 ms
16	16x16	34	0.76 ms
32	32x32	34	0.78 ms
8	8x8	13,000	0.90 ms
16	16x16	13,000	0.91 ms
32	32x32	13,000	0.97 ms
8	8x8	133,000	1.55 ms
16	16x16	133,000	1.57 ms
32	32x32	133,000	1.59 ms

Table 14.2: GPU final gathering timings

Balancing a kd-tree for fast searching is computationally cheap when the number of photons in the kd-tree is low. In Table 14.1, timings for balancing a kd-tree are shown. The time for finding the nearest 100 photons is also shown.

When the indirect illumination is updated it is important that the final gathering step is fast. We have measured how much time a single final gathering takes. A single final gathering includes a rendering of the scene using textures of the approximated illumination, a fragment processing of each pixel, summation of the pixels using hardware MIP map generation, and a copy of the final pixel to a texture (see section 4). We have timed these steps using scenes with different polygon count. The scenes with 13,000 and 133,000 polygons were made by subdividing the surfaces and making the surfaces more bumpy. The results can be observed in Table 14.2. It is noted that the timings are not very sensitive to the number of polygons in the scene. Furthermore, it is observed that improving the quality by increasing the render size does not significantly decrease the performance.

Another option would be to use traditional ray-tracing for the final gathering step and send the calculated value to the graphics hardware. Our timings show that copying a single texture value from the CPU to the GPU takes 0.65 ms. Tracing 1024 (32x32) rays in a scene with one dynamic object (i.e. two BSP-trees) and 8000 polygons takes 16.7 ms using our implementation of the axis-aligned BSP tree. Furthermore, the radiance should be calculated at the surface

that each ray hits and a final cosine weighted summation should be performed. Using our measurements, it can therefore be concluded that our hardware optimized final gathering method is many times faster than a ray-tracer based approach.

The filtering of caustics in screen space is optimized by using the occlusion query which is described in a previous section. Our timing of the occlusion query shows that it takes 0.33 ms to run 100 occlusion queries over an area of 512 times 512 pixels. Using our GPU caustics filter on an area of 512 times 512 takes 37.6 ms while only filtering 1% of this area takes 0.38 ms. In a typical scene, the caustics fills less than 5% of the screen (see Figure 14.2). The time spent on the occlusion query is therefore well worth the extra effort.

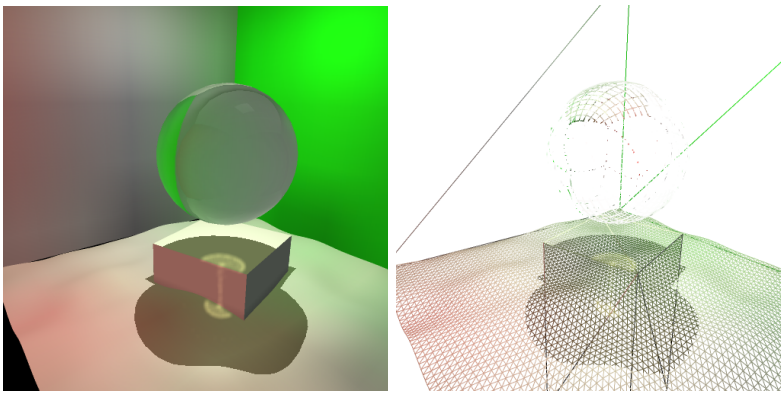

Figure 14.2: Left: Caustics being cast from a dynamic object onto another dynamic object and a bumpy floor. Right: The same scene shown using wireframe

In Figure 14.3 we have calculated the same scene both using traditional photon mapping and by using our method. It is observed that the results are very similar.

In Figure 14.4 the scene is shown with low and high texture resolution for the indirect illumination. The image with the low texture resolution uses textures with approximately 5 by 5 texels on the large surfaces. The image with the high texture resolution uses textures with approximately 50 by 50 texels on the large surfaces. In the image with the low resolution textures minor mach banding artifacts can be seen on the ceiling. Nevertheless, it is noted that in the given situation a low texture resolution is sufficient.

In Figure 14.5 we have compared the indirect illumination calculated by using our hardware accelerated MIP-mapping approach and by using traditional ray tracing based final gathering. The remaining calculations (direct illumination,

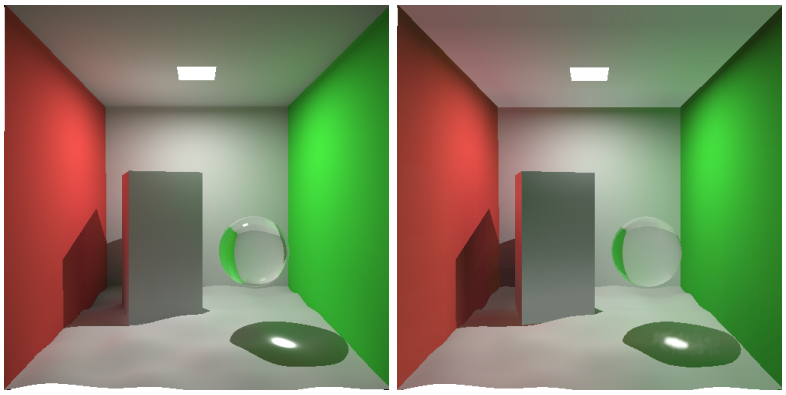

Figure 14.3: Left: Full illumination in a scene calculated by using ray tracing (hard shadows have been used for comparison). Right: The same scene shown where the illumination is calculated by using our method

shadows, specular surfaces, caustics and photon distribution) are calculated by using our method. The difference can therefore only be caused by the limited 8 bit precision of the calculations and the missing areas on the hemisphere caused by our use of the hemi-plane. It is noted that the results are very similar, which justifies the use of our hardware accelerated MIP-mapping method.

In Figure 14.6 we have compared the appearance of the caustic when one zooms out. It is noticed that the brightness of the caustics are approximately the same. This is due to our area weighting as introduced in Equation 12.2.

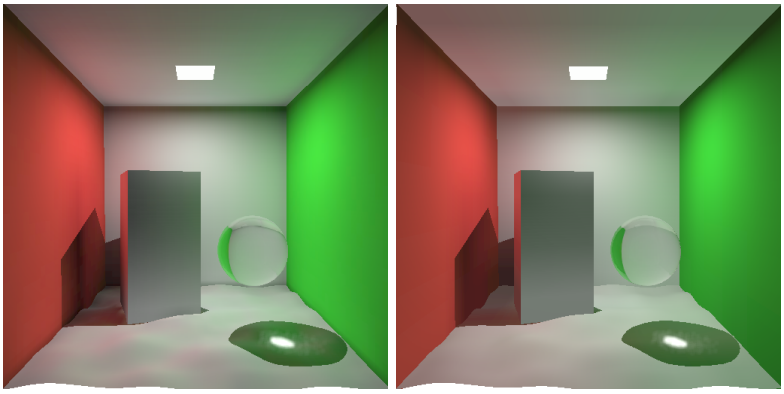

Figure 14.4: Left: Illumination calculated using our method with high texture resolution. Right: Illumination calculated using our method with low texture resolution.

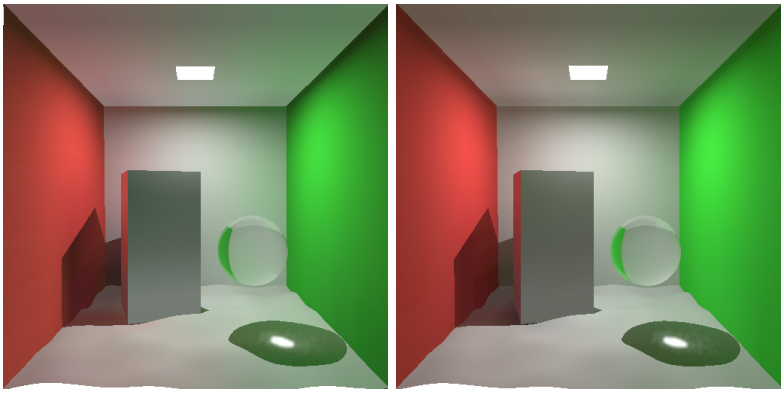

Figure 14.5: Left: Indirect illumination calculated using our method. Right: Indirect illumination calculated using ray tracing based final gather.

Figure 14.6: Left: Our area based method is used for scaling the intensity of the caustic. Right: The same scene calculated by using traditional ray tracing and photon mapping.

Part IV

Discussion & Conclusion

Chapter 15

Discussion

Many methods can be used for optimizing a 3D application. Popular methods are Occlusion Culling, Level of Detail (LOD), tri-stripping, Portals and front-to-back render order rendering, but many others can be used ([5]). These all work well with our new methods.

Tracing photons is done by using an axis-aligned BSP-tree. If the scene is divided into a number of cells e.g. by using portals the BSP-tree can just as well be divided into several trees. Since photon tracing using a BSP-tree only takes $O(\log n)$ it may not be desirable to split the BSP-tree.

Both our method for final gathering and photon tracing scales well with regard to the number of polygons in the scene. The limiting factor is the movement of objects in the scene that causes photons to be invalidated and the indirect illumination to be updated. Particularly the accuracy of the indirect illumination, i.e. the number of texels in the IIMs, is a limiting factor. Consequently, making the scene larger e.g. with more rooms and floors, will not affect the lag and frame rate if modifications occur locally. But the computation time will be affected heavily if the objects get more detailed and more and smaller texels have to be used for representing the illumination. Our method scales well with the size of the scene but not with a lot of fine details in the geometry. Nevertheless, a fractal floor, as demonstrated in our example, can be handled appropriately. Avoiding many small texels in the IIMs is an area of future research.

As each light source has a fixed number of photon groups it may be expensive to trace a single photon from each group every frame if many light sources are present in the scene. It may therefore be desirable to trace fewer photons from each light source per frame. Whether to trace a photon from each photon group or trace fewer photons from a light source can e.g. be determined by the distance from the light source to the viewer.

One of the biggest advantages of using photon mapping compared to e.g. radiosity is that it is mesh independent. When using our approach, we group the geometry into surfaces and use local photon maps, and this suggests that our method is less geometry independent than traditional photon mapping. But when calculating the irradiance by using traditional photon mapping with an n-nearest neighbors query, only photons with normals similar to the center of the query are usually used. This can be viewed as an implicit division of the geometry similar to our grouping of the surfaces.

One disadvantage of our method is our use of textures for storing the indirect illumination. When using traditional photon mapping only point sampling is used, and there is no need for applying texture maps to the surfaces. Applying texture maps to surfaces is a complicated task. Currently it is a very active research area ([45]). Nevertheless, creating animations by using photon mapping utilizing Monte Carlo integration and point sampling often produces popping and flickering ([33]). By storing the indirect illumination in textures popping and flickering is avoided.

By using our method, shadows and direct illumination is updated in every frame while the indirect illumination is updated progressively. We find this to be a good strategy since our observation is that correct direct illumination and shadows are more important than indirect illumination for the visual impression of a scene.

Our strategy for updating the indirect illumination is in many ways similar to [122] as the indirect illumination is updated selectively in object space. This is in contrast to a number of other methods like [133], [134] and [137] in which the updates are performed in image space.

We based our priorities on invalidated photons in object space. However, in [122] the priorities are calculate in camera space (as in [133], [134] and [137]). When using our method it is therefore possible to move the camera quickly without severe artifacts. This is something that is often done in e.g. games. This is possible because the indirect illumination of the entire scene is cached and because the indirect illumination is assumed to be diffusely reflected. To the best of our knowledge, there are no other approaches which performs progressive updates based on object space information. The drawback of our approach is

that expensive calculations are performed in parts of the scene that can not be seen. On the other hand, the updates may often be caused by the user and as a result the updates will often occur in the visible areas of the scene.

The Instant Global Illumination method ([128], [125]) utilizes no frame-to-frame caching. Likewise, we do not use any frame-to-frame caching for direct illumination, shadows and specular surfaces. However, we base our method on rasterization while the Instant Global Illumination method is based on ray tracing.

Our approach is based on the combination of one CPU and the GPU. This is also the case with [71], [35] and [100], whereas [128] and [122] utilizes many CPUs.

The texture resolution for the indirect illumination (IIM) is fixed in our implementation. Further research should be made to address the problem of dynamically choosing the texture resolution in order to reconstruct the indirect illumination more accurately. One direction for this research could be to apply a filter to the texture in order to find high second order derivatives, as this is probably a good location for increasing the texture resolution. Another direction would be to use methods that depend on distances to other surfaces similar to what is used in irradiance caching ([141]). A hierarchical method similar to [122] could also be used for subdividing the surfaces although it requires a fine meshing or a constant re-meshing of the scene.

It should be easy to add our methods at specific locations. E.g. in one room, indirect illumination could be enabled and at an outdoor location, caustics could be enabled for a single object. In this way, the designer of e.g. a game can make sure that the application always runs at a sufficient frame rate while adding additional features only where it will not compromise the frame rate.

Chapter 16

Conclusion

This chapter concludes this thesis. The main topics treated in this thesis are summarized in Section 16.1. An overview of the contributions are described in Section 16.2. In Section 16.3 we give our view on directions for future research. The Chapter is ended in Section 16.4 where we give our final comments.

16.1 Summary

In this thesis we have examined the components of global illumination. In the introduction we gave an overview of the parts that constitute global illumination. We also described a number of different methods for solving each of the parts in global illumination. In particular we described ray tracing and rasterization and the advantages and disadvantages of each of these methods.

The theory part had details on illumination but only the subjects that were needed in order to describe the subject in the contributions' part were treated. First it was described how direct illumination from an area light source can both be calculated using rasterization and ray tracing. It was demonstrated that in some cases the result from these two methods are similar although rasterization is substantially faster. Then the basics of photon mapping were described. Furthermore many of the optimizations that were needed for photon mapping to

run faster was described. In the discussion we concluded that photon mapping in many ways is superior to path tracing and radiosity. This is due to the speed and generality of photon mapping. In Chapter 5 we took a closer look at one of the most time consuming parts when calculating global illumination, which is the final gathering step. We described two main methods for solving the final gathering integral. First we described a ray tracer based method and different optimizations. Then we described several rasterization based methods all derived from the hemi-cube method. The conclusion was that rasterization methods can be made to run faster, while ray tracer based methods are more general. But in many cases both methods are equally good, especially when we are dealing with diffuse surfaces.

The contribution part begins with a method for dividing the photon map into several photon maps. This is necessary since the latter algorithms make local updates to each surface. Then a modified method for distributing the photons selectively is introduced. The advantage of this method is that only areas, where changes are made to the scene, will be updated. Then the full illumination is calculated based on the photon energies on the surfaces. The full illumination is stored in texture maps on the surfaces. The full illumination is used to calculate the indirect illumination using final gathering. We then present a new method for calculating the final gathering by using rasterization, fragment programs and mip-mapping. The indirect illumination is also stored in textures.

Caustics are calculated using traditional photon tracing but the photon hits are stored in a simple list. They are drawn to the screen by using points and they are then filtered by using a fragment program. An optimization is presented that limits the areas in which filtering is performed.

Finally all the individual components are combined by using a fragment program. In the result section, real-time performance is demonstrated.

16.2 Contributions

The contributions in this thesis are all components that can be used to simulate photon mapping in real-time.

The first contribution is to divide the photon map into several photon maps. We demonstrated that it gives a speedup. But we have also concluded that this solution is not the best approach in all circumstances. However, in order to selectively update the indirect illumination as we introduce a method for in Chapter 10 and Chapter 12, dividing the photon map in to several photon maps

is a necessary step.

The second contribution is to selectively distribute the photons. By storing the photon paths we are able to detect changes in the scene and in this way concentrate the photon tracing to these areas. This has produced a very scalable solution where the cost of updating the scene depends exclusively on the changes in this scene.

A third contribution is selectively to update the approximated full illumination and indirect illumination stored in textures. The updates are based on the photon energies that are added or removed from the surfaces. Continuously the surfaces on which the energies have changed the most are updated.

A fourth contribution is to calculate the final gathering by using rasterization, fragment programs and MIP-mapping. By using this technique, it is now possible to calculate the final gathering directly on the GPU with no expensive readbacks to the CPU.

The fifth contribution was to calculate caustics by using traditional photon tracing but using fragment programs for filtering in image space to reconstruct the caustics.

The sixth contribution was a technique for optimizing the caustic filtering by using occlusion queries for testing in which areas one should perform filtering. This produced a substantial optimization for calculating the caustics.

16.3 Directions for Future Research

Many methods exist for solving the global illumination problem. We believe that the best method will always be a hybrid method, and it is unlikely that a single new algorithm will outperform all others with respect to both quality and speed. Photon mapping is a hybrid algorithm, and we believe that for many years this algorithm and derivatives of it will be the best methods.

We use rasterization in our real-time photon mapping solution instead of ray tracing for both direct illumination and final gathering. Nevertheless, ray tracing is more flexible and general, and it is therefore desirable that real-time global illumination can be achieved by using ray tracing at some point.

What clearly lacks in our work is to demonstrate that our methods work for other scenes than fairly simple ones.

16.4 Final Remarks

Currently, the quest for faster global illumination is a very active area. The goal of this thesis was to create a method which could produce global illumination for real-time applications. This has to some degree been accomplished. However, this is not the end of the road as our new methods does not apply to all scenes. Some of the generality and the high image quality of the original photon mapping algorithm has suffered in order to make the algorithm run fast. We believe that our new methods can be used in some applications. Further, we believe that there is still room for improving our method with regard to generality, image quality and speed.

Bibliography

[1] T. Aila and S. Laine. Alias-free shadow maps. *Eurographics Symposium on Rendering*, 2004.

[2] Kurt Akeley, Brad Grantham, David Kirk, Tim Purcell, Larry Seiler, and Philipp Slusallek. When will ray-tracing replace rasterization? In *ACM SIGGRAPH Panel Proceedings*, 2002.

[3] Kurt Akeley and Pat Hanrahan. Cs448a: Real-time graphics architectures, 2001.

[4] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbrawedges. 2002.

[5] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering.* A. K. Peters, Ltd., 2002.

[6] Anthony A. Apodaca and M. W. Mantle. Renderman: Pursuing the future of graphics. *IEEE Computer Graphics and Applications*, pages 44–49, 1990.

[7] Arthur Appel. Some techniques for shading machine renderings of solids. *AFIPS, Spring Joint Comptr. Conf.*, pages 27–45, 1968.

[8] James Arvo. Backward ray tracing. *Developments in Ray Tracing, SIGGRAPH '86 Course Notes, Volume 12*, 1986.

[9] James Arvo, Phil Dutre, Alexander Keller, Henrik Wann Jensen, Art Owen, Matt Pharr, and Peter Shirley. Monte carlo ray tracing, siggraph course note, 2003.

[10] James Arvo and David Kirk. Particle transport and image synthesis. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 63–66, 1990.

[11] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Trans. Graph.*, 22(3):511–520, 2003.

[12] Kavita Bala, Bruce Walter, and Donald P. Greenberg. Combining edges and points for interactive high-quality rendering. *ACM Trans. Graph.*, 22(3):631–640, 2003.

[13] Gavin Bell, Anthony Parisi, and Mark Pesce. The virtual reality modeling language, 1995.

[14] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[15] J. Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, 1988.

[16] Blythe, D., Grantham, B., Kilgard, M. J., McReynolds, T., Nelson, S. R., Fowler, C., Hui, S., and Womack. Advanced graphics programming techniques using opengl: Course notes. *Proceedings of Siggraph*, 1999.

[17] J. Andreas Bærentzen. On left-balancing binary trees. Technical report, Technical University of Denmark, 2003.

[18] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *Siggraph*, 2004.

[19] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the conference on graphics hardware*, pages 37–46, 2002.

[20] E. Chan and F. Durand. An efficient hybrid shadow rendering algorithm. *Eurographics Symposium on Rendering*, 2004.

[21] Shenchang Eric Chen, Holly E. Rushmeier, Gavin Miller, and Douglass Turner. A progressive multi-pass method for global illumination. In *Proceedings of Siggraph*, pages 165–174, 1991.

[22] H. Chong and S. Gortler. A lixel for every pixel. *Eurographics Symposium on Rendering*, 2004.

[23] Per H. Christensen. Faster photon map global illumination. *Journal of Graphics Tools*, 4(3):1–10, 1999.

[24] Per H. Christensen, Byron Bashforth, Dana Batali, Chris Bernardi, Thomas Jordan, David Laur, Erin Tomson, Guido Quaroni, Wayne Wooten, and Christophe Hery. Renderman, theory and practice. *Siggraph Course Note*, 2003.

[25] Per H. Christensen and Dana Batali. An irradiance atlas for global illumination in complex production scenes. *Eurographics Symposium on Rendering*, pages 133–141, 2004.

[26] Per H. Christensen, David M. Laur, Julian Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Eurographics*, pages 543–552, 2003.

[27] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. In *Proceedings of Siggraph*, pages 75–84, 1988.

[28] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: a radiosity solution for complex environments. In *Proceedings of Siggraph*, pages 31–40, 1985.

[29] Robert L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, 1984.

[30] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 95–102. ACM Press, 1987.

[31] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of Siggraph*, pages 137–145, 1984.

[32] Franklin C. Crow. Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248, 1977.

[33] Cyrille Damez, Kirill Dmitriev, and Karol Myszkowski. State of the art in global illumination for interactive applications and high-quality animations. *EUROGRAPHICS, STAR State of The Art Report*, 2002.

[34] Paul J. Diefenbach and Norman I. Badler. Multi-pass pipeline rendering: realism for dynamic environments. In *Proceedings of Siggraph*, 1997.

[35] Kirill Dmitriev, Stefan Brabec, Karol Myszkowski, and Hans-Peter Seidel. Interactive global illumination using selective photon tracing. *Eurographics Workshop on Rendering*, pages 25–36, 2002.

[36] Craig Donner and Henrik Wann Jensen. Faster gpu computations using adaptive refinement. *Siggraph, Technical Sketch*, 2004.

[37] Dutré, Philip, Bekaert, Philippe, Bala, and Kavita. *Advanced Global Illumination*. A K Peters, 2003.

[38] Philip Dutré. Global illumination compendium, 2003.

[39] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, Kenton F. Musgrave, Bill Mark, and John Hart. *Texturing and Modeling: A procedural Approach, Third Edition*. Morgan Kaufmann, 2002.

[40] Theo Engell-Nielsen, Eric Haines, Saku Lehtinen, Vincent Scheib, and Phil Taylor. The demo scene. In *ACM SIGGRAPH 2002 Panel Proceedings*, 2002.

[41] Jeff Erickson. Pluecker coordinates. *Ray Tracing News, Volume 10, Number 3*, 1997.

[42] Everitt, Cass, and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. 2002.

[43] Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Local illumination environments for direct lighting acceleration. *Eurographics Workshop on Rendering*, 2002.

[44] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive shadow maps. In *Proceedings of Siggraph*, pages 387–390, 2001.

[45] M. S. Floater and K. Hormann. Surface parameterization: a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Multiresolution in Geometric Modelling*. Springer, 2004.

[46] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*. 1995.

[47] Tommy Fortes. Tetrahedron environment maps. Master's thesis, Chalmers University of Technology, 2000.

[48] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, 1995.

[49] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of Siggraph*, pages 213–222, 1984.

[50] Johannes Guenther, Ingo Wald, and Philipp Slusallek. Realtime caustics using distributed photon mapping. In *Proceedings of the Eurographics Symposium on Rendering*, 2004.

[51] Eric Haines. Soft planar shadows using plateaus. *Journal of Graphics Tools, 6(1):19-27*, 2001.

[52] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proceedings of Siggraph*, pages 289–298, 1990.

[53] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *Proceedings of Siggraph*, 1991.

[54] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. *Eurographics*, 2003. State-of-the-Art Report.

[55] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[56] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In *Proceedings of Siggraph*, pages 145–154, 1990.

[57] Paul S. Heckbert. Discontinuity meshing for radiosity. *Eurographics Workshop on Rendering*, pages 203–216, 1992.

[58] Tim Heidmann. Real shadows, real time. *Iris Universe*, (18):28–31, 1991.

[59] Wolfgang Heidrich. *High-quality Shading and Lighting for Hardware-accelerated Rendering*. PhD thesis, Der Technischen Fakultät der Universität Erlangen-Nürnberg, 1999.

[60] Herf and Michael. Efficient generation of soft shadow textures. Technical report, Carnegie Mellon University, 1997.

[61] N. Holzschuch and L. Alonso. Combining higher-order wavelets and discontinuity meshing: A compact representation for radiosity. *Eurographics Symposium on Rendering*, 2004.

[62] J. C. Hourcade and A. Nicolas. Algorithms for antialiased cast shadows. *Computers and Graphics, vol. 9, no. 3*, 1985.

[63] Bjarke Jacobsen, Niels Jørgen Christensen, Bent Dalgaard Larsen, and Kim Steen Petersen. Boundary correct real-time soft shadows. In *Computer Graphics International*, 2004.

[64] Henrik Wann Jensen. Importance driven path tracing using the photon map. *Eurographics Workshop on Rendering*, 1995.

[65] Henrik Wann Jensen. Global illumination using photon maps. In *Eurographics Workshop on Rendering Techniques*, pages 21–30, 1996.

[66] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Natick, MA, 2001.

[67] Henrik Wann Jensen and Niels Jørgen Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers and Graphics vol. 19 (2)*, pages 215–224, 1995.

[68] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. *Proceedings of Siggraph*, 2001.

[69] James T. Kajiya. The rendering equation. In *Proceedings of Siggraph*, pages 143–150, 1986.

[70] Jan Kautz, Marc Stamminger, Tomas Akenine-Möller, Eric Chan, Wolfgang Heidrich, and Mark Kilgard. Real-time shadowing techniques. *Course 26, Siggraph*, 2004.

[71] Alexander Keller. Instant radiosity. In *Proceedings of Siggraph*, pages 49–56, 1997.

[72] Alexander Keller and Ingo Wald. Efficient importance sampling techniques for the photon map. In *Vision Modelling and Visualization*, pages 271–279, November 2000.

[73] John Kessenich, Dave Baldwin, and Randi Rost. The opengl shading language. Technical report, 2004.

[74] Kilgard and M. J. Improving shadows and reflections via the stencil buffer. Technical report, 1999.

[75] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, pages 42–51, 1997.

[76] E. P. Lafortune and Y. D. Willems. Bidirectional path tracing. *Proceedings of CompuGraphics*, pages 95–104, 1993.

[77] Bent Dalgaard Larsen, Jacob Andreas Bærentzen, and Niels Jørgen Christensen. Using cellular phones to access virtual environments. *Siggraph*, 2002.

[78] Bent Dalgaard Larsen and Niels Jørgen Christensen. Optimizing photon mapping using multiple photon maps for irradiance estimates. *WSCG Posters*, pages 77–80, 2003.

[79] Bent Dalgaard Larsen and Niels Jørgen Christensen. Real-time terrain rendering using smooth hardware optimized level of detail. *Journal of WSCG*, 2003.

[80] Bent Dalgaard Larsen and Niels Jørgen Christensen. Simulating photon mapping for real-time applications. *Submitted to Eurographics Symposium on Rendering 2004*, 2004.

[81] Jonas Lext and Tomas Akenine-Möller. Towards rapid reconstruction for animated ray tracing. *Eurographics*, pages 311–318, 2001.

[82] Jonas Lext, Ulf Assarsson, and Tomas Möller. Bart: A benchmark for animated ray tracing. Technical report, Chalmers University of Technology, 2000.

[83] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of Siggraph*, pages 149–158, 2001.

[84] B. Lloyd, J. Wendt, N. Govindaraju, and D. Manocha. Cc shadow volumes. *Eurographics Symposium on Rendering*, 2004.

[85] MacDonald, J. David, Booth, and Kellogg S. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 1990.

[86] William R. Mark, Steve Glanville, and Kurt Akeley. Cg: A system for programming graphics hardware in a C-like language. In *Proceeding of Siggraph*, pages 896–907, 2003.

[87] T. Martin and T.-S. Tan. Anti-aliasing and continuity with trapezoidal shadow maps. *Eurographics Symposium on Rendering*, 2004.

[88] Morgan McGuire, John F. Hughes, Kevin T. Egan, Mark J. Kilgard, and Cass Everitt. Fast, practical and robust shadows. Technical report, Brown University and NVIDIA Corporation, 2003.

[89] McReynolds, T., Blythe, D., Grantham, B., Nelson, and S. Advanced graphics programming techniques using opengl. *Siggraph Course 32*, 2000.

[90] Tomas Möller and Ben Trumbore. Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools, 2(1)*, pages 21–28, 1997.

[91] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometrical considerations and nomenclature for reflectance. Technical report, National Bureau of Standards, 1977.

[92] Kasper Høy Nielsen and Niels Jørgen Christensen. Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. *Journal of WSCG*, (3):91–98, 2002.

[93] Mangesh Nijasure, Sumanta Pattanaik, and Vineet Goel. Real-time global illumination on GPU. *Submitted for publication*, 2003.

[94] NVidia. Cg toolkit - user's manual, 2003.

[95] Marc Olano, John C. Hart, Wolfgang Heidrich, and Michael McCool. *Real-Time Shading*. AK Peters, 2002.

[96] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing reyes and opengl on a stream architecture. *Proceedings of Eurographics*, 2002.

[97] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 119–126, 1999.

[98] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proceedings of Siggraph*, pages 425–432, 2000.

[99] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of Siggraph*, pages 703–712, 2002.

[100] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. *Proceedings of the conference on graphics hardware*, pages 41–50, 2003.

[101] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of Siggraph*, pages 497–500, 2001.

[102] Fernando Randima, Kilgard, and Mark J. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[103] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of Siggraph*, pages 283–291, 1987.

[104] Ashu Rege. Shadow considerations, 2004.

[105] Erik Reinhard, Brian Smits, and Chuck Hansen. Dynamic acceleration structures for interactive ray tracing. *Eurographics Workshop on Rendering*, 2000.

[106] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley, 2004.

[107] Steven Rotenberg. Cse 191: Video game programming seminar, University of California San Diego, 2003.

[108] Jörg Schmittler, Alexander Leidinger, and Philipp Slusallek. A virtual memory architexture for real-time ray tracing hardware. *Eurographics Workshop on Graphics Hardware, pages 27-36*, 2002.

[109] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime ray tracing of dynamic scenes on an fpga chip. *Graphics Hardware*, 2004.

[110] Peter Shirley, Bretton Wade, Philip M. Hubbard, David Zareski, Bruce Walter, and Donald P. Greenberg. Global illumination via density estimation. In *Eurographics Workshop on Rendering*, pages 219–230, 1995.

[111] Ken Shoemake. Plücker coordinate tutorial. *Ray Tracing News, Volume 11, Number 1*, 1998.

[112] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. In *Proceedings of Siggraph*, pages 335–344, 1989.

[113] B.W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1986.

[114] Mel Slater, Anthony Steed, and Yiorgos Chrysanthou. *Computer Graphics and Virtual Environments: From Realism to Real-Time*. Addison-Wesley, 2001.

[115] Miloslaw Smyk and Karol Myszkowski. Quality improvement for indirect illumination interpolation. *ICCVG*, 2002.

[116] Jerome Spanier and Ely Gelbard. *Monte Carlo Principles and Neutron Transport Problems*. Addison-Wesley, 1969.

[117] Marc Stamminger and George Drettakis. Perspective shadow maps. In *Proceedings of Siggraph*, pages 557–562, 2002.

[118] Frank Suykens. *On Robust Monte Carlo Algorithms for Multi-Pass Global Illumination*. PhD thesis, Katholieke Universiteit Leuven, 2002.

[119] Frank Suykens and Yves D. Willems. Density control for photon maps. In *Eurographics Workshop on Rendering*, pages 23–34, 2000.

[120] László Szirmay-Kalos, Vlastimil Havran, Benedek Balázs, and László Szécsi. On the efficiency of ray-shooting acceleration schemes. In *Proceedings of the 18th spring conference on Computer graphics*, pages 97–106, 2002.

[121] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer-generated films. *Proceedings of Siggraph*, 2004.

[122] Parag Tole, Fabio Pellacini, Bruce Walter, and Donald P. Greenberg. Interactive global illumination in dynamic scenes. In *Proceedings of Siggraph*, pages 537–546, 2002.

[123] Steve Upstill. *The Renderman Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.

[124] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of Siggraph*, pages 65–76, 1997.

[125] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[126] Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek. Interactive distributed ray tracing on commodity pc clusters – state of the art and practical applications. *Lecture Notes on Computer Science*, 2790:499–508, 2003. (Proceedings of EuroPar 2003).

[127] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[128] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive global illumination in complex and highly occluded environments. *Eurographics Symposium on Rendering*, 2003.

[129] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In *Eurographics Symposium on Rendering*, 2004.

[130] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray tracing. In *Eurographics Workshop on Rendering*, pages 15–24, 2002.

[131] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime ray tracing and its use for interactive global illumination. *Eurographics, STAR State of the Art Report*, 2003.

[132] Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing. In *State of the Art Reports, EUROGRAPHICS*, pages 21–42. 2001.

[133] B. Walter, G. Drettakis, and S. Parker. Interactive rendering using the render cache. *Eurographics Workshop on Rendering*, pages 19–30, 1999.

[134] Bruce Walter, George Drettakis, and Donald P. Greenberg. Enhancing and optimizing the render cache. In *Eurographics workshop on Rendering*, pages 37–42, 2002.

[135] Jørgen Bundgaard Wancher. Drawing a random number. Technical report, Technical University of Denmark, 2004.

[136] Michael Wand and Wolfgang Strasser. Real-time caustics. In *Computer Graphics Forum, Proceedings of Eurographics*, volume 22(3), 2003.

[137] Gregory Ward and Maryann Simmons. The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments. *ACM Trans. Graph.*, 18(4):361–368, 1999.

[138] Gregory J. Ward. Adaptive shadow testing for ray tracing. *Eurographics Rendering Workshop*, 1991.

[139] Gregory J. Ward. Measuring and modeling anisotropic reflection. In *Proceedings of Siggraph*, pages 265 – 272, 1992.

[140] Gregory J. Ward and Paul Heckbert. Irradiance gradients. In *Eurographics Workshop on Rendering*, pages 85–98, 1992.

[141] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. In *Proceedings of Siggraph*, pages 85–92, 1988.

[142] Mark Watt. Light-water interaction using backward beam tracing. In *Proceedings of Siggraph*, pages 377–385, 1990.

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

[144] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of Siggraph*, pages 270–274, 1978.

[145] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. *Eurographics Symposium on Rendering*, 2004.