Master thesis at IMM, DTU

Real-time Caustics

Sune Larsen, s991411

IMM-M.Sc.-2006-35

Abstract

A caustic is a lighting effect that occurs in nature and due to the beauty of the effect, the computer graphics field is interested in simulating it. The effect has successfully been simulated in off-line rendering and attempts have been made to adapt an algorithm for the effect in realtime rendering. However none of the fast algorithms has established itself as the standard method in the computer graphics industries. This thesis attempts to sum up what algorithms has been attempted to simulate caustics in real-time and suggest an improved algorithm.

Resumé

In this report the problem of rendering fast caustics that are coherent in close ups is the topic. A full method for generating fast caustics using a ray tracer is presented. Photons will be rendered to a texture and filtered using shaders, rather than a classic cpu approach. The use of the ray tracer will enable us to handle arbitrary surfaces, but speed is a concern. In this thesis a simple ray tracer will be implemented and this ray tracer is too slow for real-time usage. Instead the results an existing ray tracer combined with measurements in this thesis will be used to estimate running times. Three methods are explored to deal with coherency issues of the screen filtering approach. The problem is not completely solved, but the steps towards a solutions are taken. A fast solution using the auto mip-map generation capabilities of modern computers produces fast, but flawed caustics. A pre-filtering method using ray differentials is also explored and can produce nice looking results. This method is however costly. Different filtering methods for the radiance estimate are examined and a speed optimization is changed to support caustic filtering at in closeups.

Preface

This was produced at the Institute for Image Analysis and Computer Graphics at Danish Technical University, Kgs. Lyngby, Denmark.

Reading the report requires a basic understanding of Computer Graphics.

The report is structured so an overview of some of the algorithms already available is given first followed by the general background theory that will form the basis of the work.

In chapter 4 the details of the total algorithm is given, this will include some specific problems and strengths of the different parts of the algorithm. In chapter 5 the implementation itself will be discussed, including UML diagrams of the classes and details on the environment used. In chapter 6 the testing method and results will be given followed by a discussion of those results. Finally some ideas that was not explored or implemented will be given in chapter 8 and a conclusion of project. In chapter 3 an analysis of the problem and the thoughts behind choices made are given.

Acknowledgments

I would like to thank friends and family for support during this project.

I would also like to thank some of my co-students that have made the learning experience more enjoyable.

Finally I would like to thank Bent D. Larsen for guidance and useful suggestions during this project.

Contents

Chapter 1

Introduction

Here the topic of caustics will be introduced followed by a survey of some algorithms that already exist for the real-time (or near real-time) simulation of caustics. As will become clear, there are many different approaches, but none have reached mainstream usage as of yet. The algorithms are suitable different applications, but there is still room for improvement. However it seems likely that one might be able to combine or improve an existing algorithm to create a more generally applicable algorithm, that is able to generate real-time caustics.

1.1 Caustics

Caustics is a captivating lighting effect, that will captivate most people at the age of children. I think most people remember marvelling at the light phenomena caused by the sunlight hitting a magnifying glass, creating a bright spot on the targeted surface. The bright spot is a caustic and it is this lighting effect that would be fascinating to be able to incorporate into graphics applications.

Another common and beautiful caustic is that caused by light hitting water and being refracted onto a surface.

To create a caustic light needs to be focused by a reflective surface (such as a brass ring) or a refractive surface (such as water, a glass lens or a transparent ball) onto a diffuse surface. On figure 1.1 a caustic from a cognac glass is shown.

1.2 Related work

The area of caustics is well researched and many different algorithms have been suggested, all of which have there pros and cons. So far an algorithm for general usage has not emerged. Here we shall take a look at some of the work that has been done up until now, in this field of research.

The first algorithm was suggested by Arvo [9]. Arvo uses a two step algorithm. The first step rays of light are traced from the light source into the scene using a standard Monte Carlo ray tracer[1]. When the rays intersect a diffuse sur-

[1]The act of tracing a ray from the light source into the scene is also called forward ray

Figure 1.1: A caustic from a cognac glass.

face, they are stored in an illumination map, which is a texture for each surface containing illumination. The second step is rendering, where the map is used to look up the information needed to calculate global illumination including caustics. The greatest weakness of this method is the memory consumption of the storage method. Arvo's algorithm was also designed mainly for offline usage.

Henrik Wann Jensen expanded Arvo's algorithm in [5]. It would now be able to handle more complex lighting phenomena such as participating media. The biggest change was the addition of a new method for storage and rendering of caustics (and general global illumination). The photons are traced in the same manner as with Arvo, but instead they are stored in single data structure, namely a Kd-tree. During rendering the volume that a chosen amount of photons take is found and this information is used for solving the rendering equation. This method is more memory efficient, but is still intended for real-time usage.

The two discussed algorithms have influenced much of the work done in the field of real-time caustics. We will move on to discuss some of the fast algorithms that have been suggested for generating caustics.

The perhaps most direct descendant of Jensens classic photon-map was presented by Timothy J. Purcell et al. in [23]. It is an adaptation of the classic algorithm for calculation on the gpu and can handle full global illumination. Instead of a Kd-tree, which cannot immediately be implemented for use with shaders, a grid structure is used that is easily compatible with textures. Two methods of storage and search are suggest. One method is through by using a Bitonic Merge Sort, which requires many rendering passes. The other method

tracing or path tracing

uses the stencil buffer, which can be accomplished by limiting the number of photons per grid cell. With the second method a photon can be stored correctly in a single pass. This method delivers full global illumination, but has a rendering time of 10 secs.

Another algorithm was suggest by Wand et al. in [6]. Their algorithm divides a specular surface (ie. the surface of a possible caustics generator) into patches. It then renders the light sources into an environment map for each specular object. A set of sample point are now chosen on the object, which are used as pinhole cameras. The diffuse receivers are then rendered and the direction from the current raster position to a sample point on the specular surface is then reflected (using the specular surface normal). The reflected direction is used as a lookup in the environment map. The sample points are distributed uniformly over the surface. The caustics produced by this algorithm suffer from aliasing artifacts, visibility is not calculated fully and distributing sample points increases the cost hurts scalability. This algorithm also only supports single reflective bounce caustics, with the possibility of expanding to include single refractive bounce caustics.

Musawir Shah et al. presents an algorithm in [1] which uses a technique similar to shadow maps. The algorithm uses 3 steps. The first step is rendering the receiver surfaces (diffuse surfaces) from the lights point of view and storing the world space positions in a texture. For the second step, the specular surfaces are rendered. A vertex shader is used to estimate the point resulting from the reflection or refraction at the vertex. Several passes may be used to get a better estimation. The points are splatted onto the receiver geometry and used to estimate the caustic. The caustic is estimated using the ratio between the triangles that surround the specular vertex and the receiving triangles. This method handles dynamic scenes naturally. It supports single surface and double surface refractions and reflections, which may be sufficient. It however has issues with the precision of its reflection. The detail of the caustic is also very dependant on the tessellation of the geometry. The biggest issue is that if the caustic is formed outside the light source view frustum it will not be generated, which can be an issue with point or complex light sources.

The last algorithm we will discuss was presented by Bent Dalgaard Larsen in [3]. It uses a fast Monte Carlo ray tracer to distribute photons. The photons are stored in a simple structure and rendered to a texture. The texture is filtered and blended with a rendering of the scene. By using the ray tracer this method is able to handle arbitrary scenes, possibly with any type geometry and advanced lighting situations. This methods ability to handle dynamic scenes depends on the ray tracer. The filtering method is fast and produces nice looking caustics, but does not handle close ups well. It is this method we will expand upon.

In this thesis the interest is an algorithm that handles arbitrary scenes, without the use of cluster computing. However it's worth mentioning that other algorithms have been suggested using CPU clusters. Also single bounce refraction caustics that occur due to the presence of water have been given special attention.

Chapter 2

Background theory

In this part of the thesis a summary of the theory, which is the foundation of the resulting algorithm is explained. First a brief introduction to the physical description of light is given. This is followed by various other theories that have affected the algorithm.

2.1 Solid Angle

Solid angle is a value often used in radiometry. It exist in both 3d and 2d. In 3d the solid angle represents the area of the ray on the unit sphere and in 2d the solid angle is the interval on the unit circle. The solid angle has the unit steradian (where the angle for the entire unit sphere area is 4π steradians). An illustration is shown in Figure 2.1. The differental solid angle can be described

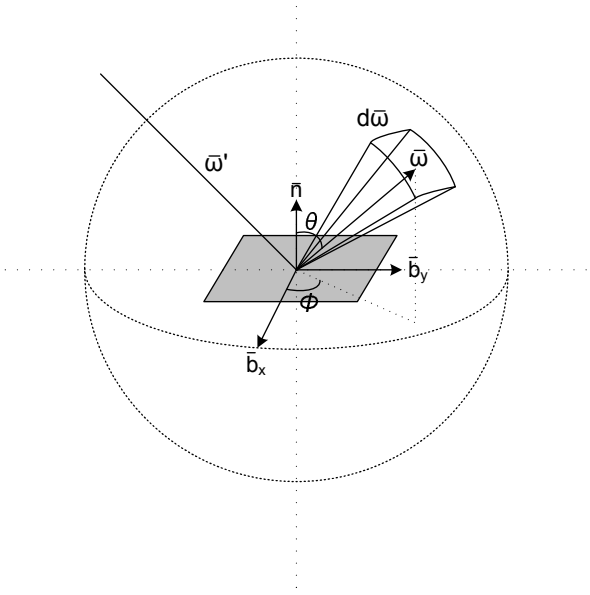

Figure 2.1: Illustration of solid angle. Total area of sphere is 4π steradian.

by spherical coordinates :

$$d\vec{\omega} = sin\theta d\theta d\phi \tag{2.1}$$

where θ is the angle between the light direction and the surface normal n. θ is the angle between the light direction projected onto the surface plane and $\vec{b_x}$ (where $\vec{b_x}$ and $\vec{b_y}$ are two vectors orthogonal to n in the surface tangent plane). The direction of the solid angle is given by :

$$\vec{\omega} = sin\theta cos\phi\vec{b_x} + sin\theta sin\phi\vec{b_y} + cos\theta\vec{n} \tag{2.2}$$

2.2 Radiometry

Radiometry is the description of light. It is the basis used in all equations that follows in this thesis. The most notable being radiant flux, irradiance and even more so radiance. The basic unit in lighting is the photon. A photon is a part of an electromagnetic wave, in this context light. A photon could be perceived as a wave with a wavelength, λ, which energy is given by :

$$e_\lambda = \frac{hc}{\lambda} \tag{2.3}$$

where $h \approx 6.63 \cdot 10^{-34} J \cdot s$ (Planck's constant) and $c = c_0 = 299,792,458 m/s$ is the speed of light in vacuum. In some respects a photon acts as a particle and this is the way a photon is often consider in computer graphics. Light can be considered as a large amount of photons. The spectral radiant energy for a collection of n_λ photons with the same wavelength λ is given by :

$$Q_\lambda = n_\lambda e_\lambda = n_\lambda \frac{hc}{\lambda} \tag{2.4}$$

For n_λ photons with varying λ the radiant energy Q is the integral over all possible wavelengths :

$$Q = \int_0^\infty Q_\lambda d\lambda \tag{2.5}$$

The radiant flux Φ is the flow of radiant energy over time through a point space. This is also called radiant power since it is the power of the light travelling through that point.

$$\Phi = \frac{dQ}{dt} \tag{2.6}$$

The radiant flux area density is defined by :

$$\frac{d\Phi}{dA} \tag{2.7}$$

which is often divided into two parts. The flux leaving the surface, radiant exitance M and the flux leaving the surface called irradiance, E :

$$E(x) = \frac{d\Phi}{dA} \tag{2.8}$$

The radiant intensity I gives the power of a light beam per solid angle unit.

$$I(\vec{\omega}) = \frac{d\Phi}{d\vec{\omega}} \tag{2.9}$$

The Radiance gives the amount of light that passes through or is emitted from a surface. I considers light of a single wavelength, from all angles (incoming or outgoing) over the area considered. The radiance L is the radiant flux per solid angle unit :

$$L(x,\vec{\omega}) = \frac{d^2\Phi}{cos\theta dA d\vec{\omega}} = \int_0^\infty \frac{d^4 n_\lambda}{cos\theta d\vec{\omega} dA dt d\lambda} \frac{hc}{\lambda} d\lambda \tag{2.10}$$

2.3 Light-surface interaction

When light hits a surface it is either absorbed or scattered. How this happens determines the visual appearance of the viewed surface and is therefore central to computer graphics.

2.3.1 BRDF

One of the central topics of computer graphics is the interaction between light and a surface. When a beam of light hits an object in nature it penetrates the surface and may scatter inside that object before leaving through a possibly different point. This interaction is described by an BSSRDF [15] (*Bidirectional Scattering Surface Reflectance Distribution Function*). If one makes the assumption that a light beam will be reflected at the intersection point, rather than enter the object, one can describe the light/surface interaction by the simpler BRDF [15] (*Bidirectional Reflectance Distribution Function*).
The BRDF, f_r, describes the relation between reflected radiance $dL_r(x,\vec{\omega})$ and irradiance $dE_i(x,\vec{\omega}')$, given by :

$$f_r(x,\vec{\omega}',\vec{\omega}) = \frac{dL_r(x,\vec{\omega})}{dE_i(x,\vec{\omega}')} \tag{2.11}$$

for a given point x, incoming direction $\vec{\omega}'$ and outgoing direction $\vec{\omega}$.
If one knows the BRDF and incoming radiance for a surface one can find the reflected radiance for that surface by integrating over the hemisphere of incoming directions Ω:

$$L_r(x,\vec{\omega}) = \int_\Omega f_r(x,\vec{\omega}',\vec{\omega}) dE(x,\vec{\omega}') = \int_\Omega f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}')(\vec{\omega}'\cdot\vec{n}) d\vec{\omega}' \tag{2.12}$$

where n is the surface normal at x with $\vec{\omega}'\cdot\vec{n} = cos\theta'$.

2.3.2 Reflectance

When light hits a surface some will be absorbed or transmitted and some will be reflected. The amount of light that the surface reflects is given by the reflectance ρ of the surface :

$$\rho(x) = \frac{d\Phi_r(x)}{d\Phi_i(x)} \tag{2.13}$$

where $d\Phi_r(x)$ outgoing flux and $d\Phi_i(x)$ is the incoming flux.

2.3.3 Reflection

Reflection can be handled by different BRDF's here we will describe two special cases of reflection namely perfectly diffuse and perfectly specular.

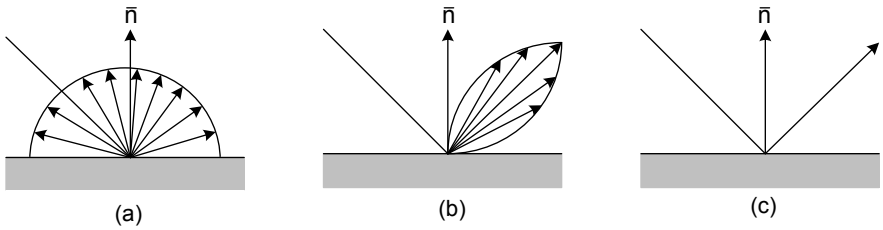

Figure 2.2: Illustration of three types of light scattering. (a) shows diffuse scattering, (b) shows glossy specular reflection and (c) shows perfect specular reflection. Glossy reflection occurs at surfaces that are both diffuse and specular.

Diffuse reflection

Diffuse reflection can occur when light hits a surface and is scattered in different directions. This happens with rough surfaces. Perfectly diffuse reflection (or Lambertian reflection) is when light is scattered in perfectly random distribution of all directions (see Figure 2.3). This gives the visual appearance of equal lighting from every angle. This results in the BRDF $f_{r,d}$ being constant over the hemisphere :

$$L_r(x,\vec{\omega}) = f_{r,d}(x)\int_{\Omega} dE_i(x,\vec{\omega}') = f_{r,d}(x)E_i(x) \tag{2.14}$$

where BRDF itself is $f_{r,d} = k_d$, with $k_d \epsilon [0;1]$ being a diffuse constant. The reflectance ρ_d for a Lambertian surface is :

$$\rho_d(x) = \pi f_{r,d}(x) \tag{2.15}$$

and the outgoing direction of the reflection is chosen at random (due to the random nature of the diffuse scattering) by two variables $\xi_1 \epsilon [0,1]$ and $\xi_2 \epsilon [0,1]$ using :

$$\vec{\omega}_d = (\theta,\phi) = (cos^{-1}(\sqrt{\xi_1}), 2\pi\xi_2) \tag{2.16}$$

where, in spherical coordinates, θ is the angle with the angle with the surface normal and phi is the rotation.

Specular reflection

Specular reflection is the light reflection off a smooth surface (metal, water etc.), which leads to the visible appearance of highlights. Unlike diffuse reflection there is only a degree of scattering of the light ray, which is caused by the roughness (glossiness) of the surface. A glossy BRDF describes a non-perfect specular reflection, but the most simple and common BRDF is that of the perfect specular reflection. For a perfectly specular surface the light is reflected completely in the mirror direction as shown on Figure 2.3. The reflected radiance, L_r, is determined by :

$$L_r(x,\vec{\omega}_s) = \rho_s(x)L_i(x,\vec{\omega}') \tag{2.17}$$

where ρ_s will be determined by the Fresnel equations. For perfect reflection the mirror direction is given by.

$$\vec{\omega}_s = 2(\vec{\omega}' \cdot \vec{n})\vec{n} - \vec{\omega}' \tag{2.18}$$

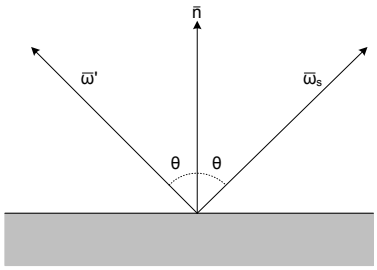

Figure 2.3: Illustration of specular reflection with angles and vectors.

Fresnel

As a light hit a surface some of the light might be reflected and some refracted. The amounts are given by the Fresnel reflection coefficient, F_r

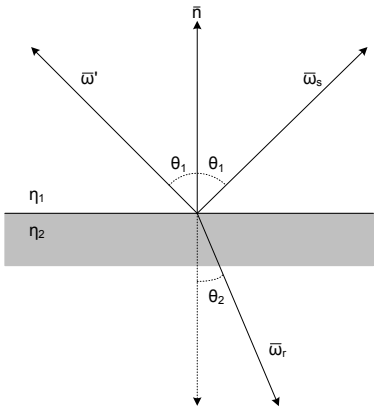

Figure 2.4: Illustration of reflection and refraction with angles and vectors.

$$F_r(\Theta) = \frac{1}{2}(\rho_{\parallel}^2 + \rho_{\perp}^2) = \frac{d\Phi_r}{d\Phi_i} \tag{2.19}$$

the values $\rho_{\parallel}^2$ and $\rho_{\perp}^2$ are given by the Fresnel equations.

$$\rho_{\parallel} = \frac{\eta_2 cos\Theta_1 - \eta_1 cos\Theta_2}{n_2 cos\Theta_1 + n_1 cos\Theta_2} = \left(\frac{cos\Theta_1 - cos\Theta_2}{cos\Theta_1 + cos\Theta_2}\right)^2 \tag{2.20}$$

$$\rho_{\perp} = \frac{\eta_1 cos\Theta_1 - \eta_2 cos\Theta_2}{n_2 cos\Theta_1 + n_1 cos\Theta_2} = \left(\frac{tan\Theta_1 - tan\Theta_2}{tan\Theta_1 + tan\Theta_2}\right)^2 \tag{2.21}$$

some common approximate values for η are

η	$\approx$
air	1.0
$water$	1.33
$glass$	$1.5 - 1.7$

Christophe Schlick [22] approximated the Fresnel coefficient by the simpler :

$$F_r(\Theta) \approx a + (1-a)(1-cos\Theta)^c \tag{2.22}$$

where the values can be selected, suggest values are $a = F_0$ (where F_0 is the Fresnel reflection coefficient at normal incidence) and $c = 5$ is a constant that can be chosen, the value 1 is suggested by Schlick. F_0 is the value of the normal Fresnel coefficient at the incident. F_r gives the amount of reflected light and $1 - F_r$ gives the amount of refracted light. An even faster, but cruder empirical approximation to Schlick's model is given in [24] as

$$max(0, min(1, F_r(\Theta) \approx a + b(1 + \vec{\omega} \cdot N)^c)) \tag{2.23}$$

this is based on the appearance of light rather than the physics of light.

2.3.4 Refraction

Refraction for a smooth surface is shown in Figure 2.4. The angles Θ_1 and Θ_2, and refractive indices η_1 and η_2 are related by Snell's law

$$\frac{\eta_1}{\eta_2} = \frac{sin\Theta_2}{sin\Theta_1} \tag{2.24}$$

The outgoing direction from a refraction, $\vec{\omega}_r$, is given by

$$\vec{\omega}_r = -\frac{\eta_1}{\eta_2}(\vec{\omega} - (\vec{\omega} \cdot \vec{n})\vec{n}) - \left(\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - (\vec{\omega} \cdot \vec{n})^2)}\right)\vec{n} \tag{2.25}$$

In the case of negative value in the square root all the light is reflected, giving a mirror effect. This can happen when light travels from a medium with low η_1 to a media with high η_2 at the critical angle Θ_c.

2.4 Rendering equation

The render equation was first introduced by Kajiya [16] and is at the heart of the different illumination methods. It gives the outgoing radiance $L_o(x, \vec{\omega})$ at any point in a model. Here the equation is presented in a slightly different form than that of Kajiya's original definition :

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + L_r(x, \vec{\omega}) \tag{2.26}$$

where L_e is the emitted light and L_r is the reflected light at position x with outgoing direction $\vec{\omega}$. How to determine L_r is not straightforward and will be described in section 2.3.1. This can be expressed using BRDF's, which using Equation (2.12) gives :

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_\Omega f_r(x, \vec{\omega}', \vec{\omega}) L_i(x, \vec{\omega}')(\vec{\omega}' \cdot \vec{n}) d\vec{\omega}' \tag{2.27}$$

The render equation is dependant on the BRDF and the choice of BRDF significantly effects the visual appearance of an object.

2.5 Ray tracing

Ray tracing is a point sampling technique for calculating global illumination. The basic rendering technique was introduced in 1980 by [2]. Ray tracing calculates the path of a light ray trough a scene and samples the intersection points along that path. In nature light rays travel from the light source to the viewer. Simulating this approach has been researched, it however requires a large amount of rays to be traced and even with many rays traced will likely still produce a noisy image. The popular ray tracing technique however traces rays from the viewer out into the scene, thus reducing the number of rays required to equal the resolution of the resulting image and eliminating a need for multiple samples per pixel.

The start of a trace path is defined by the view position and direction including

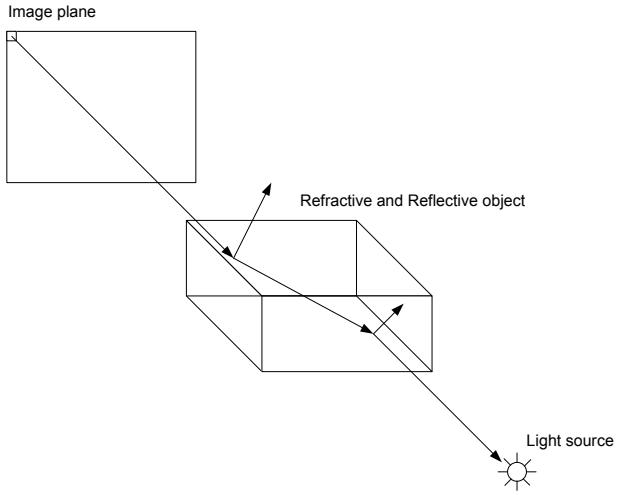

Figure 2.5: Illustration of tracing a ray from a pixel in the view plane into the scene. Maximum recursion depth is 3.

naturally a scene with light sources. The nearest intersection between ray and object is found.

At an intersection point the local lighting is calculated for each light source and global lighting is added. Reflected and refracted rays are also emitted if the surface is specular and/or transparent.

The algorithm is recursive and uses two functions. Where `trace` find the nearest (shortest distance, d) intersection and `shade` calculates the lighting at the intersection. The ray tracing algorithm uses shadow ray to test whether the intersected point is in shadow. It does so by tracing a ray from the point to the light source in question. For this reason hard shadows are easily added to the ray tracer. The basic ray tracer itself cannot produce full global illumination. But other than easy hard shadows the advantages are :

- Natural hidden surface removal
- Natural reflections and refractions on entire scene.

```
for each pixel
    color = trace{ray}

trace(ray)
    find nearest intersection with object
    find point and normal of intersected surface
    color = shade(point,normal)
    return color

shade(point,normal)
    color = 0
    for each light source
        emit shadow ray
        if shadow ray does not intersect
            color = local color
    if(surface is specular)
        color = color + trace(reflected ray) + trace(refracted ray)
    return color
```

- Support for any geometry
- Support for advanced lighting models (BRDF's etc.)
- Blackbox nature of scene and objects allow for optimization of individual sections of the algorithm. (Which will be discussed later).

Some of the global illumination further elements that have been implemented for the ray tracer are :

- Number of calculations
- Memory consumption for extremely large scenes

More advanced features of global illumination can be added to the ray tracer. Some of these are :

- Depth of field
- Motion blur
- Caustics, which is the topics of this thesis
- Indirect illumination

2.6 Photon-map

Photon-mapping was introduced by Henrik Wann Jensen in [5] as a means of handling global illumination effects (caustics, color bleeding and such). The method uses two passes:

1. Photon emission
2. Rendering

Photon emission is normally handled by a ray tracer, which means how complex a scene, objects, BRDF's etc. are handled is limited purely by the ray tracer.

Emission

As mentioned emission is usually accomplished by tracing photons path through a scene. Photons are emitted from light sources and carry their energy (power) through the scene until stored. The power can be determined as a fraction of the light source energy. The light source energy is equally divided amongst photons emitted from the light. Photons are traced through a scene in the same way as a ray, but at the intersection points the two are handled differently. The cause of this is the fact that a ray gathers radiance, where a photon delivers flux. A photon does not scatter and this means that, at an intersection, it must either be reflected, refracted or absorbed. A probabilistic technique called Russian roulette is used to decide what action to take. If reflection is chosen for a non-diffuse surface the reflection is handled with the BRDF. If reflection is chosen for a diffuse surface a random direction is chosen. For caustics it is usual to eliminate photons that do not hit a specular surface as the first interaction. The photons that hit a specular surface are the ones that are likely to contribute to a caustic. See Figure 2.6 for an illustration of photon emission.

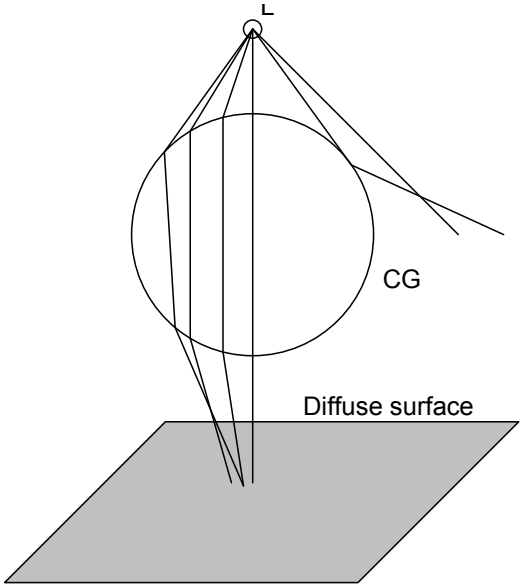

Figure 2.6: Illustration of photon paths, from a light source L, into a scene with a caustic generating sphere, CG, that is both refractive and slightly reflective.

Storage

Photons are stored in a single data structure. The important aspects of the structure is that finding points in a radius around another point is fast and that storing many photons is as cheap as possible. The number of photons needed depends on the scene, for caustics the number is usually not as great as for full global illumination. A photon is classically represented by

```
struct photon {
```

```
    float x,y,z;    // Position of the photon
    float[4] power; // Power of the photon
    char phi,theta; // Incident direction
    short flag;     // Used for kd-tree
}
```

The data structure chosen by Wann Jensen is a Balanced Kd-tree. The Kd-tree uses axis aligned planes to divide the scene into voxels, which makes it possible to search for photons around a point efficiently.

Radiance estimate

We are interested in evaluating the outgoing radiance for a surface, which is given by :

$$L_r(x,\vec{\omega}) = \int_{\omega} f_r(x,\vec{\omega}',\vec{\omega})\frac{d^2\Theta_i(x,\vec{\omega}')}{dA_i} \approx \sum_{p=1}^{n} f_r(x,\vec{\omega}',\vec{\omega})\frac{\Theta_p(x,\vec{\omega}_p)}{\Delta A} \tag{2.28}$$

where Θ_i is the incoming flux, estimated from n photons in a radius from the point x. Each photon p has power $\Theta_p(x,\vec{\omega}_p)$. Density estimation of the photons stored is used to evaluate the equation. Density estimation is done by finding the N-photons that are nearest to the point in space at which one wants to evaluate the equation. An area is give, usually by a sphere containing the N nearest photons. The energy of the photons is summed and divided by the area of the sphere.

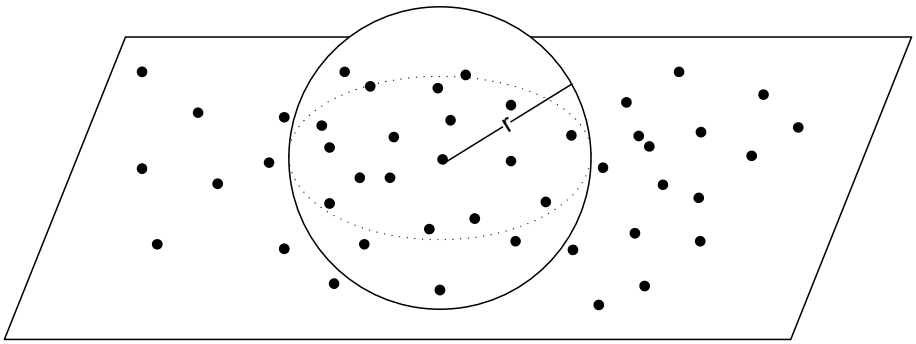

Figure 2.7: Illustration of a sphere volume with radius, r, used for density estimation. The volume has been expanded so it contains a desired number of photons.

Optimizing photon-mapping has been the topic of some research and in the appendices will be a short overview of some the methods can be found.

2.7 Halton sequences

A sequence random numbers in does not necessarily distribute evenly in the interval they are chosen from. This is not always desirable and other number

sequences could be considered and one that is often used is a from Halton Sequence. A Halton sequence is a quasi-Monte Carlo sequence, which means that it is not truly random. A Halton sequence in one dimension consist of numbers generated by dividing an interval uniformly. A Halton sequence is a also called a reversed radix-based sequence. This is because it uses a radical inverse function to pick a value in the interval $[0; 1[$ from an integer. A sequence value is found by evaluating

$$\Phi_b(i) = \sum_{j=0}^{\infty} a_j(i) b^{-j-1} \Leftrightarrow i = \sum_{j=0}^{\infty} a_j(i) b^j \tag{2.29}$$

for value i and base b, where a_j is the sequential digits of the value i. In plain language what happens can be explained as :

1. Expressing the value i in base b.
2. Taking the value found in step 1 and reversing the order of the digits.
3. And finally adding a floating point in front of the value.

An example would be the radical inverse of $i = 1234$ in base $b = 10$ would give the value 0.4321.

The bases, b, that the Halton sequence is built from is a chosen from the prime numbers. This means that if one uses several bases there is no or little correlation between the sequences.

2.8 Mip-maps

Textures are rarely displayed in their natural size and the pre-calculation method, Mip-mapping, was introduced by Williams in [19] to improve sampling. Mip-mapping generates a series of textures from an original texture. The resolution is halved at each level, so a 128x128 texture will have a pyramid (with level 0 being the highest resolution) of texture with resolutions 64x64, 32x32, 16x16 and so forth. The mip-map levels are created using a square averaging filter

Figure 2.8: The mip-map levels of an image. Level 0 to 5, left to right.

with height and width, 2^k, where k is the level in the pyramid.
There are different methods for doing texture lookups. Trilinear is the most common and it utilizes bilinear filtering. A value is determined bilinear filtering

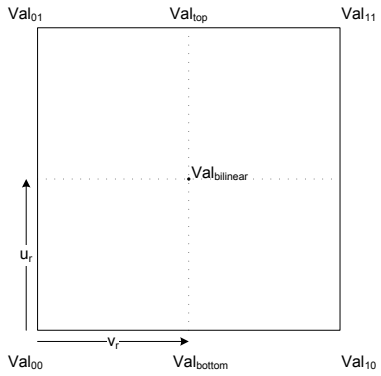

Figure 2.9: Illustration of bilinear sampling between four values.

on 4 pixel values as follows :

$$value_{bottom} = val_{00} + u_r(val_{10} - val_{00}) \quad (2.30)$$
$$value_{top} = val_{01} + u_r(val_{11} - val_{01}) \quad (2.31)$$
$$value_{bilinear} = value_{bottom} + v_r(value_{top} - value_{bottom}) \quad (2.32)$$

where u_r and v_r are u, v coordinates relative to the target u, v. To smooth the transition between different levels of detail trilinear interpolation is used. The value is determined from the chosen level and the two surrounding levels, of coarser and finer detail. First bilinear interpolation is used on the levels followed by linear interpolation between the values.
Determining the level of detail is difficult and different application use different methods (for more information see [18] for a description of some possibilities).

2.9 Perspective transformation

The perspective transformation is a part of the rendering pipeline and in practice the details of the implementation may differ from the basic theory presented here. To take a point from object space (x, y, z) to screen (x_s, y_s, z_s) space, the coordinates are first transformed into view space (or eye space, x_e, y_e, z_e) and then screen space. It is the last part of this transformation that's named the perspective transformation.
A perspective transformation can be defined in several ways, here we will examine a definition using the parameters :

Field of view, θ, which is the angle giving the height of the view plane.

Ratio, r, which is the ratio between height and width of the view plane.

Near, N, which is the distance in view space to the near plane (or view plane).

Far, F, which is the distance in view space to the far plane.

The screen space is perceived to be a projection of everything contained within a view frustum. Only objects contained within this frustum are rendered (this occlusion takes place in the rendering pipeline, sometimes after the perspective transform). We are looking to project the eye coordinates into screen coordi-

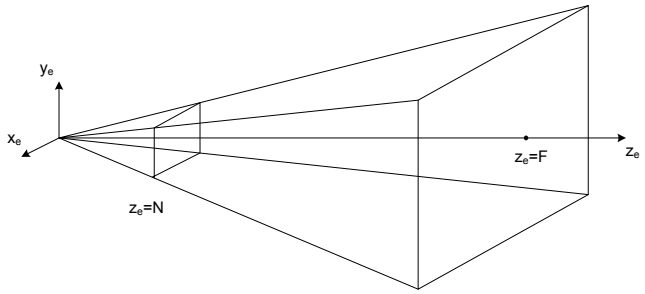

Figure 2.10: Illustration of a view frustum with Near plane (N) and far plane (F).

nates given in the intervals :

$$x\epsilon[-1,1] \quad y\epsilon[-1,1] \quad z\epsilon[0,1] \tag{2.33}$$

The screen coordinates x_s and y_s are determined in a straightforward manor by :

$$x_s = \frac{1}{r}N\frac{x_e}{wz_e} \tag{2.34}$$

$$y_s = N\frac{hy_e}{hz_e} \tag{2.35}$$

where the height $h = Ntan(\theta/2)$ and view ratio, r of the image plane, are used to scale the coordinates into the desired intervals.
The z-transformation was proven by [20] to take on the form :

$$z_s = A + \frac{B}{z_e} \tag{2.36}$$

Looking at a view frustum with $F = 1$ one can use the two equations

$$0 = A + \frac{B}{z_{min}} \tag{2.37}$$

$$1 = A + B. \tag{2.38}$$

To determine complete transformation

$$z_s = F\frac{1 - \frac{N}{z_e}}{F - N} \tag{2.39}$$

This transformation however is non-linear and cannot be performed using matrices. In the rendering pipeline the transformation is separated into two steps

by introducing homogenous coordinates. Homogenous coordinates (x, y, z, w) containing an additional fourth coordinate w :

$$x = x_e \tag{2.40}$$
$$y = y_e \tag{2.41}$$
$$z = \frac{hFz_e}{N(F-N)} - \frac{hF}{F-N} \tag{2.42}$$
$$w = \frac{hz_e}{N} \tag{2.43}$$

These transformations are linear and can be performed as such :

$$[x \;\; y \;\; z \;\; w] = [x_e \;\; y_e \;\; z_e \;\; 1]P \tag{2.44}$$

using the projection matrix, P :

$$P = \begin{bmatrix} r & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{hF}{N(F-N)} & \frac{h}{N} \\ 0 & 0 & -\frac{hF}{F-N} & 0 \end{bmatrix} \tag{2.45}$$

The screen coordinates are the determined from homogenous coordinates by the non-linear perspective divide

$$x_s = \frac{x}{w} \tag{2.46}$$
$$y_s = \frac{y}{w} \tag{2.47}$$
$$z_s = \frac{z}{w} \tag{2.48}$$

2.10 Ray differentials

A common issue in imaging is aliasing and several methods have been developed for anti-aliasing ray traced images. In [21] Homan Igehy presents a fast and robust method for estimating a rays footprint. A rays footprint is an estimate of the size of the ray. Here only the general theory will be presented and we refer the reader to Igehy's article for the anti-aliasing example.

Ray differentials are an estimation of the deviation in ray direction caused by different phenomenon (such as reflection and refraction). Ray differentials are cheap because tracing extra rays is not necessary, instead some variables need to be calculated at each intersection. The ray position and one or more offset ray are used to calculate a rays footprint.

The foundation of the method is that phenomenon can be presented as differentiable functions and the traversal of a ray through a scene can be represented as a series of these functions.

$$v = f_n(f_{n-1}(\ldots(f_2(f_1(x, y))))) \tag{2.49}$$

Equally is the differentiability of these functions, that using the chain-rule gives :

$$\frac{\partial v}{\partial x} = \frac{\partial f_n}{\partial f_{n-1}} \cdots \frac{\partial f_2}{\partial f_1} \frac{\partial f_1}{\partial x} \tag{2.50}$$

In the following a ray, $\vec{R}$ will be presented by a point, P and direction, D on the form :

$$\vec{R} = \langle P \;\; D \rangle \tag{2.51}$$

The function is recursive so for different usages one can parameterize ones initial functions by different values. For ray tracing the x, y-coordinates in the view plane are used. The initial direction is given by the function :

$$d(x, y) = View + xRight + yUp \tag{2.52}$$

where $View$ is the view plane position, $Right$ is the right vector contained by the plane and Up is the up vector of the view plane. The initial values for ray tracing are thus given by the ray origin, P and normalized ray direction D.

$$P(x, y) = Eye \tag{2.53}$$

$$D(x, y) = \frac{d}{\sqrt{d \cdot d}} \tag{2.54}$$

One or several ray differentials, that are offsets of R, can be tracked. The ray differentials are given by two different partial derivatives, one in each offset direction :

$$\frac{\partial \vec{R}}{\partial x} = \left\langle \frac{\partial P}{\partial x} \;\; \frac{\partial D}{\partial x} \right\rangle \tag{2.55}$$

$$\frac{\partial \vec{R}}{\partial y} = \left\langle \frac{\partial P}{\partial y} \;\; \frac{\partial D}{\partial y} \right\rangle \tag{2.56}$$

Each offset rays that one needs to estimate are represented by these two differentials. From here on we will focus on the x-offset ray differential, the y-offset ray differential is treated equally. At the core of ray differentials lies the decision to use a first order Taylor approximation to estimate the offset ray.

$$\left[\vec{R}(x + \triangle x, y) - \vec{R}(x, y)\right] \approx \triangle x \frac{\partial \vec{R}(x, y)}{\partial x} \tag{2.57}$$

$$\left[\vec{R}(x, y + \triangle y) - \vec{R}(x, y)\right] \approx \triangle x \frac{\partial \vec{R}(x, y)}{\partial y} \tag{2.58}$$

A higher order Taylor approximation is possible, but Igehy states that generally a first order approximation is sufficient. The initial ray differential values for ray tracing are given by

$$\frac{\partial P}{\partial x} = 0 \tag{2.59}$$

$$\frac{\partial D}{\partial x} = \frac{(d \cdot d)Right - (d \cdot Right)d}{(d \cdot d)^{3/2}} \tag{2.60}$$

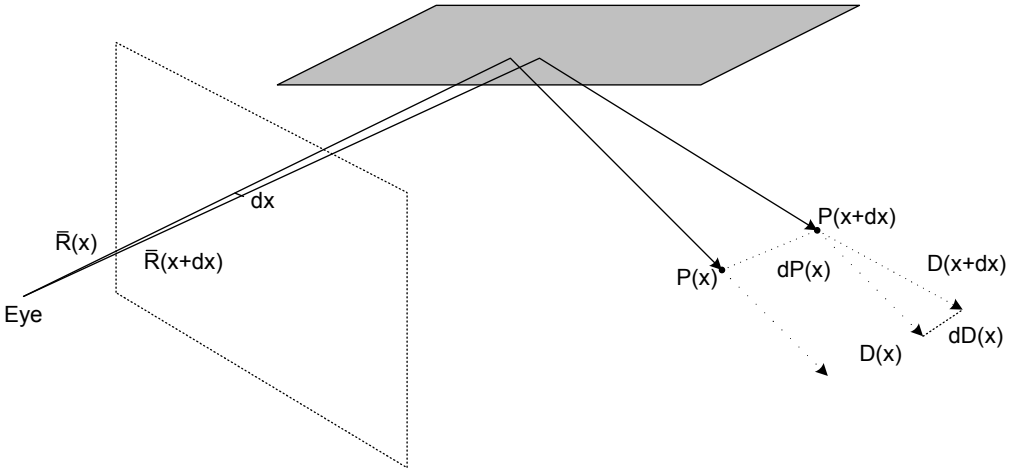

Figure 2.11: Illustration of a ray path and its ray differential, offset in screen space, x. The letter d in the image is equal to ∂.

Propagation

The phenomenons most commonly used in ray tracing are the simple reflections and refractions described in section 2.3.3 and 2.3.4. At intersections the direction of the ray change, but the position changes during transfer as will become clear. Reflections and refractions occur after a transfer.

Transfer

Transfer is the act of a ray travelling unhindered through a medium. The position is given by the function of a straight line.

$$P' = P + tD \tag{2.61}$$
$$D' = D \tag{2.62}$$

Simple differentiation gives the values needed to describe the ray differentials.

$$\frac{\partial P'}{\partial x} = \left(\frac{\partial P}{\partial x} + t\frac{\partial D}{\partial x}\right) + \frac{\partial t}{\partial x}D \tag{2.63}$$
$$\frac{\partial D'}{\partial x} = \frac{\partial D}{\partial x} \tag{2.64}$$

where the distance t is given as follows. For a planar surface containing P', where $P' \cdot N = 0$, the distance t is given by:

$$t = -\frac{P \cdot N}{D \cdot N} \tag{2.65}$$

Both the point P and direction D are both projected onto the same vector N meaning that any arbitrary N will give the same ratio. Therefore instead of $P' \cdot N = 0$ one can decide to use the surface normal. The differential of t is:

$$\frac{\partial t}{\partial x} = -\frac{(\frac{\partial P}{\partial x} + t\frac{\partial D}{\partial x}) \cdot N}{D \cdot N} \tag{2.66}$$

Reflection

Reflection is given by the simple reflection equation (2.18).

$$P' = P \tag{2.67}$$
$$D' = D - 2(D \cdot N)N \tag{2.68}$$

The ray differential is given by :

$$\frac{\partial P'}{\partial x} = \frac{\partial P}{\partial x} \tag{2.69}$$
$$\frac{\partial D'}{\partial x} = \frac{\partial D}{\partial x} - 2\left[(D \cdot N)\frac{\partial N}{\partial x} + \frac{\partial (D \cdot N)}{\partial x}N\right] \tag{2.70}$$

where :

$$\frac{\partial (D \cdot N)}{\partial x} = \frac{\partial D}{\partial x} \cdot N + D \cdot \frac{\partial N}{\partial x} \tag{2.71}$$

The derived normal, $\frac{\partial N}{\partial x}$, will be described later in this section.

Refraction

Refraction is given by the simple refraction equation (2.25).

$$P' = P \tag{2.72}$$
$$D' = \eta D - \mu N \tag{2.73}$$

where η in this notation is the ratio between the refraction indices of the two media. The ray differential is given by :

$$\mu = [\eta(D \cdot N) - (D' \cdot N)] \tag{2.74}$$
$$D' \cdot N = -\sqrt{1 - \eta^2[1 - (D \cdot N)^2]} \tag{2.75}$$

where :

$$\frac{\partial P'}{\partial d} = \frac{\partial P}{\partial x} \tag{2.76}$$
$$\frac{\partial D'}{\partial x} = \eta\frac{\partial D}{\partial x} - \left(\mu\frac{\partial N}{\partial x} + \frac{\partial \mu}{\partial x}N\right) \tag{2.77}$$

$$\frac{\partial \mu}{\partial x} = \left[\eta - \frac{\eta^2(D \cdot N)}{D' \cdot N}\right]\frac{\partial (D \cdot N)}{\partial x} \tag{2.78}$$

The derived normal is described below and

$$\frac{\partial (D \cdot N)}{\partial x} = \frac{\partial D}{\partial x} \cdot N + D \cdot \frac{\partial N}{\partial x} \tag{2.79}$$

Differential normal of triangles

The surface normal is determined in different way for different surface representations, however a mesh surface using triangles is the most common type. A triangle consists of 3 points, $(P_\alpha, P_\beta, P_\gamma)$ and 3 normals, $(N_\alpha, N_\beta, N_\gamma)$. If one looks at a point P contained within the triangle (on the triangle plane), as

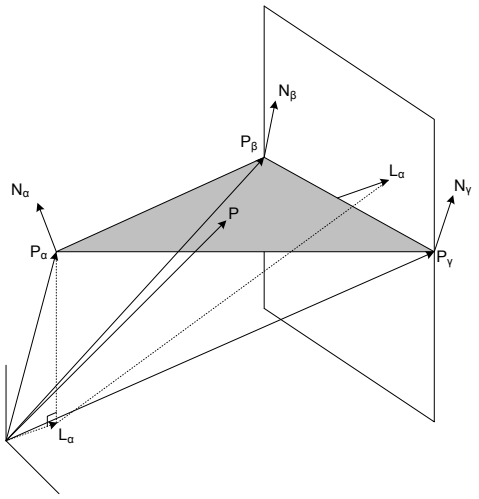

Figure 2.12: Illustration of a triangle and the values need in the following.

shown on Figure 2.12. The point can be determined by linear interpolation of the three points in the triangle.

$$P = \alpha P_\alpha + \beta P_\beta + \gamma P_\gamma \tag{2.80}$$

The barycentric weights, α, β, γ, complies to

$$\alpha + \beta + \gamma = 1 \tag{2.81}$$

assuming that P lies within the triangle. If P is already known the barycentric weights can be determined by using the planes L_α, L_β and L_γ. The planes are determined in the same way, for L_α the plane is any plane containing P_β and P_γ, while also being perpendicular to the triangle. L_α is normalized in a way so $L_alpha \cdot P_alpha = 1$. The barycentric weights are then determined by:

$$\alpha(P) = L_\alpha \cdot P \tag{2.82}$$
$$\beta(P) = L_\beta \cdot P \tag{2.83}$$
$$\gamma(P) = L_\gamma \cdot P \tag{2.84}$$

The normal, N is then also determined by linear interpolation:

$$n = (L_\alpha \cdot P)\, N_\alpha + (L_\beta \cdot P)\, N_\beta + (L_\gamma \cdot P)\, N_\gamma \tag{2.85}$$
$$N = \frac{n}{\sqrt{n \cdot n}} \tag{2.86}$$

The differentials are:

$$\frac{\partial n}{\partial x} = \left(L_\alpha \cdot \frac{\partial P}{\partial x}\right) N_\alpha + \left(L_\beta \cdot \frac{\partial P}{\partial x}\right) N_\beta + \left(L_\gamma \cdot \frac{\partial P}{\partial x}\right) N_\gamma \tag{2.87}$$
$$\frac{\partial N}{\partial x} = \frac{(n \cdot n)\frac{\partial n}{\partial x} - \left(n \cdot \frac{\partial n}{\partial x}\right) n}{(n \cdot n)^{3/2}} \tag{2.88}$$

Chapter 3

Problem Analysis

The problem is creating a fast and robust method for simulating caustics.

Some brute force methods could be considered. Increasing the number of CPU's would be possible, and the photon emission part of the algorithm it lends itself naturally to parallelization. However the building of the photon-map cannot be parallelized and thus after emission all information about photons would have to be distributed to all CPU's placing a large bandwidth. The method has been applied for caustics in [8]. This implementation test on 9 AthlonMP 1800+ CPU's, does not produce what would be considered real-time frame rates. Furthermore we are targeting a single CPU implementation.

Another brute force method, was attempted in [23], by implementing the photon-mapping technique entirely on the GPU. This method is however too slow for real-time usage. It does however include all global illumination effects.

In this thesis we will implement and expand upon the algorithm presented by Bent D. Larsen in [3]. This method uses a fast ray tracer to emit photons and a pixel shader[1] is used to filter a screen-sized texture containing the rendered photons.

Some advantages of this method of emission are that the ray tracer handle advanced BRDF's. The ray tracer can also be implemented to handle any geometry including meshes or parametric surfaces. Building a fast ray tracer is challenging and good implementations already exist. In this thesis the focus will not be on the ray tracer, and building a ray tracer, with all the existing optimizations is considered beyond the scope of this work. However a section in the appendices gives a summary of the optimizations for photon tracing that was considered for the thesis. What is needed for photon distribution is a ray tracer that can handle any meshes to test different scenes, but it is not required that the ray tracer is fast.

Focus will be on filtering. Bent D. Larsen has worked on some of the problems with screen space filtering and provide improvements for the basic filter.

[1] In the shading language HLSL each pixel is processed by a pixel shader. In another common shading language, Cg, this shader is called a fragment shader.

The basic filter, which will be described in more depth in Section 4.4, is a filter kernel that weighs the photon count of a pixel by the distance from the center of the kernel. This does not solve the density estimation from classic photon mapping and is not physically accurate. We will present a new filtering equation that better approximates the, classic radiance estimate.
Each pixel contains the number of photons that was projected into the pixel. The number of pixels in a screen at given resolution is naturally constant, thus when the eye point moves closer to the caustic the greater the number of pixels covering the caustic will be. Therefore fewer photons will be filtered per pixel and the intensity of the image will decrease. The solution suggested by Bent is to used the area of the projected into the pixel, we will implement this and describe this improvement.
The caustic is created from randomly distributed photons and redistributing each frame or often means that photon positions will change. This change leads to flickering in the caustic and Bent suggests using Halton sequences instead of random numbers to improve on the stability of the appearance of the caustic. This will be implemented.
The filtering algorithm used is rather expensive and Bent suggest an optimization that efficiently can decide what pixels contain the caustic and should be filtered. This method is however not built to handle close-ups and a new version will be presented.

The greatest remaining unsolved issues of this method is regarding the coherency of the caustic. When zooming in on the caustic, the distribution of photons disperse and with a filter radius of around 4 (giving a filter kernel of 9x9) the caustic quickly looses the appearance of a coherent lighting phenomena.
The first thought is to simply increase the filter kernel radius. This would achieve results similar to the classic density estimate (which increases the filter radius until a number of photons are found). However this expensive since the number of samples (or area of the kernel) is given by

$$k_{area} = (1 + 2r)^2 \tag{3.1}$$

with kernel radius r. This means the radius for the first radius sizes gives

r	$samples(k_{area})$
1	9
2	25
3	49
4	81
5	121

When the camera is close to the caustic, data needs to be provided to compensate for the decreasing amount of filtering data, that in the areas of interest will be due to the inaccuracy of emitting a limited number of photons. Compensation should still be as physically accurate as possibly and ideally the following goals should be achieved.

- The coherency solution should preserve the amount of energy in stored photons, ie. the total amount of energy in the caustic should remain the same for each zoom level.

- The shape of the caustic should also be preserved.
- The cost of achieving the optimization should not prevent real-time application.

Three methods are considered candidates, two uses mip-maps and one uses ray footprints.

Mipmaps

Instead of using a larger filter, using pre-filtered data might be an option. The pre-filtered data needs to be generated effectively, so using manual filtering is out of question (and would probably be slower than simply filtering on the GPU). A feature in modern graphics hardware is automatic generation of mipmap levels. This is a very efficient mipmap filtering that takes place during rendering. Using mipmap levels from a texture containing the photons would provide a pixel with a color value that had been filtered with a larger kernel. However this kernel is different from the basic caustic filtering kernel we use, since mipmap generation uses a uniform averaging filter. This means that the method differs from the caustic filtering in that it does not weigh the photons with regards to radius (this leads to a perfectly square filter). The energy contained at the different levels of the photons is close to equal, but due to the values being stored in an 8-bit structure values will be rounded (if the uniform average filter produces floating point values, such as 0.5f) causing some imprecision. Also one should be careful using the highest levels of the mip-map. Figure 3.1 illustrates what can go wrong. The conclusion must be that one should be careful what maximum level one uses.

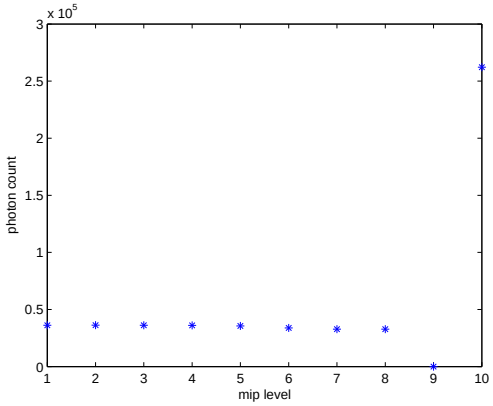

Figure 3.1: This is a histogram showing the sum of 700x700 at different levels of an image. The image used was a photon texture containing a caustic generated with a refractive sphere. The sum is, as expected, unchanged for the mip-levels until one reaches the last levels with resolution 2x2 and 1x1. It is seen that at the photon count fluctuates wildly for those two levels. The cause of this is that level 8 has the color value (0,0,0) in its four pixels and level 9 has the color value 1 in its pixel. This is possible because all the images are generated from the original image.

The shape of the caustic should roughly be maintained, but will naturally become more blocky due the filter shape. However since we are basically filtering the same photon texture the receiver surface can still support any arbitrary receiver surface shape. The blocky appearance of the outer shape of the caustic could be rounded using ray differentials, which could be used as an estimate radius at the storage position. The ray differentials should be a cheap addition to the ray tracing cost (no extra intersection tests are needed) and if rendering the resulting discs can be cheaply achieved, the addition to the filter itself will be one additional texture lookup.

Much like with mip-maps used for anti-aliasing it is tricky to decide how to blend the mip levels and in this case also the normal filtering. This decision should be based on the density of the photon distribution. The density however depends on many factors such as the number of photons, the screen resolution, the geometry of the scene (both caustic generators and receiver surfaces) and the camera position. We present two methods.

The first method we use is a purely empirical method of blending that simply uses a set of functions to blend in the different levels. The functions are approximations of the perspective transformation of the x and y values, thus the blending of levels occur in a pace according to the change in the density of the distribution.

The second method we test is more closely related to the way the classic photon-map method works. In this method the sub sampling filter of the mip-maps is considered to be the approximately the same as the caustic filter. (this is not true, since mip-mapping uses uniform average sub-sampling and caustic filtering uses convolution with a rounded filter) This means that using a different level of the filter is the same as expanding the search radius, and sampling the level gives the average value. Choosing levels is accomplished using one or more samples and looking at the values.

Other methods where considered as well. Another empirical method would be to use the u,v-area contained by the pixel, which has already been calculated, to decide the zoom level. This would require a scale factor but, would naturally take into account the perspective transformation. This is a method that is sometimes applied when using mip-maps for anti-aliasing. This is very closely related to the pixel area method, which is already included.

The ray footprint achieved by ray differentials could also be applied to this problem. The ray footprint estimates how far from a photon, a slightly offset photon, would be at the point of storage. This method is not completely based on the geometry of the scene because the initial offset direction of a ray differential is chosen arbitrarily. To support more than one caustic in a scene it is necessary to have a value per pixel. An attempted method that was dropped was to splat a value onto a texture and passing that texture to the shader. The value was the radius of the ray differential projected into screen space and the splattering pattern was discs with the radius of the ray differential. This would mean that the projected radius was available at the points where it was estimated that photons would be. However concentrated photons with small radii

and widely spread photons will have large radii. These will likely overlap and in the areas, not covered by the concentrated photons will have large radii. This leads to sharp differences in the power of the caustic.

An issue with using ray differentials are critical angles. Marking the ray differentials that exceed a critical angle along a differential rays path would render these rays useless with regards to making decisions. The physically correct thing to do, would be to reflect a ray differential that exceeds the critical angle, however this would not be useful since the reflected ray would not add to the refracted caustic (which is what the original photon is a part of since critical angles only occur during refraction). Three options considered was:

1. One could draw the photon discs with a fixed size, which could be the mean size or perhaps an arbitrarily chosen size. The problem with the mean size is that more than one caustic may be in the image. These caustics may have very different mean sizes and thus the flagged photons would not fit into either caustic. The arbitrarily chosen size is even less general.

2. One could draw the photons using the radius of another foot print in the distribution. With this method one runs the method of picking photons, which radius differ greatly from the drawn photon discs.

3. The safe method, which we will choose, is to simply ignore photons that have been flagged. The drawback to this method is that photons are lost, and thus depending on usage could never produce a perfect approximation.

The pixel area optimization that is applied to normal filtering, as mentioned earlier, is also applied to the mip-map image, thus adjusting energy with regards to the orientation of the receiving geometry, without change the energy in normal or mip-map filtering. The part of the pixel area optimization that takes the distance from the surface into consideration is kept due to the fact that the amount of energy for the normal level of the photon texture and the mip levels are assumed to be relatively close.

Pre-filtered footprints

In the classic algorithm the radiance estimate includes, the expanding of a volume until it contains a satisfying number of photons. This could be done in screen space if the number of texture lookups was not a concern, but as stated previously this is the case. A fundamental restriction to this algorithm is that it must perform the density using a fixed volume (or in our case area, since screen space is 2 dimensional). This means that the number of photons used in the estimate can change resulting in the possibility of empty areas.

The third proposed method uses ray footprints estimated with ray differentials to calculate a pre-filtered density estimate. The idea is that using ray differentials we essentially have an circular area which can be used to divide the power of the photon. The reason why this might be a possible method is that the area varies according to the density of the caustic at the position of the photon, ie. the smaller the area, the more focused the part of the caustic is.

Circular discs will be drawn to a texture. The power of the disc will be the power of a photon divided by the area of the disc. Overlapping disc will be added together. This method would preserve the total amount of energy in the caustic, where it not for photons lost to critical angles.

The advantage this method should have over the other is avoidance of the blocky appearance of the mip-map. The shape would naturally be rounded and several levels could be generated by using more ray differentials. This method however could possibly be too expensive for real-time, because we are drawing discs instead of points. A cheap method of drawing a disc might be to draw a square using 4 indexed vertices and alpha blending with a texture containing the circle. This however is expensive compared to a single point that requires no shading, and if many photons are used in a scene the workload could be too great for real-time applications. It should be noted that in the Section 2.10 the ray differentials are parameterized by x,y in the image plane, instead we use Θ, ϕ offsets from the light direction.

To summarize we will attempt implementing of 3 methods, two empirical and one that attempts an approximation of the classic photon-map method. These will be described in Section 4.4.5 We will also improve upon the parts of the original algorithm proposed by Bent D. Larsen. A change will be made to the filtering part described in Section 4.4.1 and to the quad optimization in Section 4.4.3.

Finally a short discussion on what graphics system to use for implementation. The main concern with regards to implementation is how much needs to be implemented and ease of implementation. The combination Managed DirectX and C# has free implementations of the algebra needed and mesh classes with a built in intersection method. A drawback might be the relative youth of Managed DirectX, which may limit the detail level of the documentation available.

Chapter 4

Algorithm

The focus of the algorithm will be on how hardware optimized caustics filtering can be achieved, with optimal visual appearance. Much effort has been put into developing very efficient ray tracers and optimizations that complement each other now exist. Ray tracing as a more important part of real-time rendering is becoming more feasible. In this implementation focus will be on the coherency of the caustic and an accelerated ray tracer will not be included in this implementation.

4.1 Overview

Before we get into details on each part of the algorithm we will give a brief overview. As described in 2.6 there are two passes in a classic photon map algorithm, namely emission and rendering.

Emission

For emission we use a simple ray tracer and store the photons as positions and offset positions in an array. The ray tracer supports simple meshes and meshes from .x files. It has no optimizations and utilizes the built-in intersection method of DirectX Mesh objects to test for intersection against objects (more detail on this method can be found in Section 5.3). There is the option of either emitting a full number of photons each frame or only emit the photons once and then use the same photons for rendering. For a discussion of some of the optimizations that could be considered for this implementation see the appendices.

Rendering

Rendering of the caustic will be composed of several rendering passes to achieve the different improvement to the quality of the caustics that where discussed in the analysis section. Before describing each part of the algorithm, short descriptions of each render pass that happens during a frame rendering will be given :

1. The first render pass is there to create the photon texture. During this pass the mip levels are also auto generated. This pass needs to render the

entire scene, without lighting or effects and the photons as points. This is needed to get the right occlusion of photons.

2. If the ray footprint optimization is enabled an extra pass is needed. This pass needs to render the scene, black with no light and then it needs to render the photons as discs instead of points.

3. The second pass is to generate the quads for the quad speed optimization. This pass need to either render the same scene as the first pass followed by a render of a number of quad shaped polygons. Alternatively it needs to render the same scene as the ray footprint pass followed by the quads.

4. The third pass is used to generate the texture with the information for the pixel area optimization. This will only be generated and used if the option is checked.

5. The final render pass is caustic rendering itself. This pass renders the full scene and then blends in the caustic using filtering and either mip levels or ray footprint texture.

4.2 Photon emission

For photon emission a ray tracer is used, which has the advantage of supporting any geometry. Ray tracers are being optimized for full global illumination, which requires a large amount of rays to be traced. For global illumination a very large photons must be spread into the entire scene, which is not necessary for caustics. Solutions for real-time global lighting using ray tracing has been suggested, two such are found in [3] and [7]. Disitrubtion and speed is important to the algorithm. The speed depends on the ray's traversal through a medium (we assume no participating media, see [4] for information on this topic) and interaction with surfaces. The final distribution of photons is primarily decided by the geometry of the scene, a scene built from parameterized geometry would give a more accurate photon distribution than the mesh representation. How great the difference is also depends on the tessellation of the mesh geometry. In this implementation we will stick with mesh objects. A factor that affects the final photon distribution, which we can control is the initial ray directions. We use either a spot or point light source to distribute pick the initial directions. The pseudo code for distributing from a point light source is given below on Figure 4.1.

4.2.1 Distribution

The caustics generated are an approximation created by filtering a distribution of points in both the classic algorithm in the one presented here. The distribution of photons is very important to both appearance of the caustics and performance of the algorithm. It should also be noted that the application is set up to emit photons from a spot, but is also capable of emitting from a simple point light source, the results appearance wise would be the same it would just take longer to calculate.

```
emit_photons(scene,irradiance)
{
    for (no_of_photons) {
        theta = random between [0,1]
        phi = random between [0,1]

        direction.x = sin(theta)*cos(phi)
        direction.y = sin(theta)*sin(phi)
        direction.z = cos(theta)

        lightsource = choose random lightsource from scene
        ray(lightsource.position, direction)
        trace_photon(scene, ray, irradiance, 0)
    }
}

trace_photon(scene, ray, irradiance, level)
{
    if(level < maxlevel)
    {
        if(level == 0)
            incident = nearest non-diffuse intersection
        else
            incident = nearest intersection

        if(diffuse surface)
            photonmap.store(incident.position)
        else
        {
            phenomenon = use russian roulette to determine phenomenon
            if(phenomenon == reflection)
                trace_photon(scene, spawn_reflected_ray(ray,incident),
                             ray.irradiance, level++)
            else
                trace_photon(scene, spawn_refracted_ray(ray,incident),
                             ray.irradiance, level++)
        }
    }
}
```

Figure 4.1: Pseudocode for photon emission for a point light using explicit sampling.

Number of rays

The number of rays needed to generate caustics in a scene differ from scene and can be adjusted, this is a weakness of the algorithm in that it cannot handle every scene equally well, without adjusting settings. If one shoots in random directions from light sources a lot of noise might be the result of a high number of rays. Photons, that do not add to the appearance of the caustic or caustics in a scene, will be considered to be noise (this would not be the case in full global illumination). Noise is unwanted in regards to the appearance of caustics and will also make it more difficult to optimize the filtering operation (which will become clear in Section 4.4.3). Noise is also costly with regards to time spent calculating ray traversal, since the photons stored are unwanted.

To reduce noise in the scene the algorithm will only calculate ray traversal if the first intersection is a caustics generator. The advantage is the elimination of unwanted photon paths after one level of traversal. This assumes that the only photon paths that add to a caustic are the ones that hit a caustic generator in their first step of traversal and this assumption is not entirely correct. Diffuse reflections can also add to caustics, but this will in most scenes be an unlikely

occurrence. The result of the assumption is also that caustics generated after several reflections and/or refractions will not be captured as seen on Figure 4.2.1. Another possible implementation would be to not emit all the photons

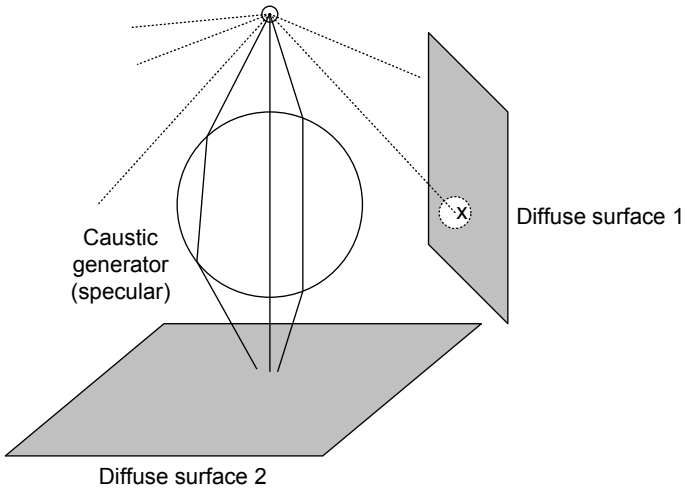

Figure 4.2: Illustration of a with two diffuse surfaces and a specular surface (caustic generator). The caustic is likely to form on diffuse surface 2 due to the position of the light source. However light may hit a point x on diffuse surface 1. Diffuse reflection might be chosen and the direction is chosen at random and light from this reflection may or may not add to the caustic. The most likely outcome is that it wont and instead will strike a random diffuse surface, thus adding noise to the scene.

every frame. This would work by dropping a percentage of the photons from the distribution from the previous frame one will save a lot of ray traversal calculations, which will likely be the bottleneck of an application. The result of this will depend on how dynamic the scene is ie. for a dynamic scene with much animation the reuse may be apparent even if a large number of photons are shot, but for a static scene a caustic might flicker less, since only a smaller number of random positions are removed and re-emitted rather than creating a new set of photons. In our application we will not be going for this optimization since we wont be testing in real-time and shooting once will suffice for the purpose of this thesis. An application of this optimization can be found in [17].

Halton sequences

The visual appearance of caustics is dependent on the storage position of the photons. For a caustic to look coherent it is preferable that the photons be uniformly distributed. When dealing with animated caustics, a quasi-random distribution will be preferable to a completely random distribution The reason is that a quasi-random distribution can be more uniformly distributed. The result is a more slightly more coherent caustic and less variance from frame to frame resulting in less fluctuation in the caustic (less flickering).
Both random numbers and Halton sequences generate two values, r_1 and r_2, in

the interval $[0;1]$, which can be used to generate two angles, $\varphi \in [0, 2\pi]$ and $\Theta \in [0, \pi]$. These angles are spherical polar coordinates and represent a directional vector.
The Halton sequences are generated on startup and simple lookups are performed during rendering. We do a random lookup in the distribution, which should still have less variance.
This is not a solution to the problem of coherent caustics, but an improvement that does not add any cost to the algorithm during rendering. Calculation of the distribution takes place once at program start, during runtime look ups are used.

4.2.2 Traversal

The traversal algorithm is the basic algorithm from the classic photon-map method using Russian roulette to decide, whether to reflect or refract photons on intersection with non-diffuse surfaces. We assume that between objects there is vacuum and not participating media (such as milk, marble) so travelling between objects does not require special attention or incur a computational penalty. This ray tracer implementation is simple and thus rays are checked for intersection with all objects in the scene, for large scenes this is expensive and scene trees are common in advanced ray tracers. BSP-trees in particular are popular. Intersections are handled by the black box method in the DirectX Mesh class. An examination of the most popular optimizations for ray tracers with photon tracing in mind, can be found in the appendices.

4.3 Photon storage and representation

Photon storage and photon representation is dictated by the photon filtering part of the algorithm. Filtering will be done in image space because we wish to take advantage of the possible hardware acceleration by using the gpu instead of the cpu. The only way to pass large amounts of data on to the shaders are through textures. This means that advanced data structures (such as Henrik Wann's classic photon-map) are not possible and for we will only be able to display 8-bit values ie. 8-bit (alpha,red,green,blues) values. The 8-bit texture

0	0	0	4	11
0	5	0	5	7
0	0	0	0	8
0	2	1	7	0
0	0	0	0	0
1	0	3	0	0
0	0	0	0	0
0	0	0	0	0

Figure 4.3: Illustration of a photon texture after the photons have been rendered additively as points.

means that there is a limit to the number of values that it is possible to store,

ie. from 0 to 255. In the future the precision of the display buffer may change allowing greater numbers if needed. Whether 255 is enough depends on the geometry, number of photons emitted and the screen resolution. The greater the screen resolution the more precisely the photons will be projected onto the screen ie. photons that might be stored at the same position at lower resolution will now be stored in different pixels. This is a problem that occurs due the screen space filtering which means that the area of the filter in world space changes, so it's size relative to the photon distribution changes. That is not the case with the classic density estimation, which fully takes place in world space.

Due to technical issues we will be limited to less than 255 photons per pixel in this implementation, when using mip-maps. This is caused by auto mip-map generation only working for 8-bit textures. Fractions has to be either floored and ceiled. It seems that the hardware used for implementation floors the values, meaning that low values are quickly eroded. Therefore we scale the photon value by 20, which means that a pixel can contain 12.8 photons, which of course is not physically accurate, but sadly necessary. We adjust for this scaling in the shading and for the rest of this thesis, this will be transparent.

One of the drawbacks of the single texture method is that we are forced to make the assumption that all photons emitted have the same color (or power), which could be especially limiting in scenes with several light sources.

For use with occlusion culling, which will be explained in next chapter, the photon storage points are also stored as 3d points in an array.

4.4 Photon rendering

Filtering will take place in a pixel shader to take advantage of hardware acceleration. This is likely to be a good solution with regards to speed and will become better as graphics cards are improved. GPU speeds are evolving faster than speeds of the CPU's, which is promising. However more and more algorithms are moved to the GPU, so it hard to predict what the future results on the GPU in a full application setting (such as a game) will be.
As mentioned earlier we are working in screen space and we will need a texture with viewport size.

The most precise caustics are achieved using the classic algorithm for photon-maps by Henrik Wann. In his algorithm filtering takes place in world space, which as stated previously is not possible in a shader. In this algorithm filtering takes place in screen space and the loss of depth information leads to several issues, that produce less accurate caustics. This is a sacrifice that might be acceptable for real-time applications (the assumption that this is the case is the motivation for the research in the area).

Note that in the following, the descriptions of the parts of the algorithm will not be described in the order they are run. See the overview for that information.

4.4.1 Caustic rendering

The rendering of a caustic is achieved by filtering the texture containing the stored photons. The basic filtering equation suggested by [3] is as follows :

$$c_{empirical}(x,y) = s \sum_{i=-k}^{k} \sum_{j=-k}^{k} t(x+i, y+j)\sqrt{1+2k^2-(i^2+j^2)} \qquad (4.1)$$

where c is the caustic contribution to the color of the pixel, that comes from reading the finest level of the texture.The value s is scale factor, t is the texture value read from the photon texture and $\sqrt{1+2k^2-(i^2+j^2)}$ is a radius based scale (where k=4 means a 9x9 kernel) to round the caustic and preventing a blocky appearance. It is assumed that all light sources has the same color. Bent's is not based in physical reality, a method that closer resembles the was used by [17]. They use a square filter and simply divide

$$c_{square}(x,y) = \frac{s}{A} \sum_{i=-k}^{k} \sum_{j=-k}^{k} t(x+i, y+j) \qquad (4.2)$$

by the area, $A = (1+2r)^2$. This closer approximates the radiance estimate given by equation 2.28. However the square shape produces blocky looking textures except at a distance. Instead we suggest using a rounded filter, in the same way a Bent, but present a version that resembles the radiance estimate more closely.

$$c_{round}(x,y) = \frac{s}{A} \sum_{i=-k}^{k} \sum_{j=-k}^{k} ratio * t(x+i, y+j) \qquad (4.3)$$

where c is the final color, s is a scale factor, A is an area and ratio is the ratio between distance from the kernel center to the pixel, and the length diagonal. A and ratio will now be explained. We will not divide by the exact area in pixels, and note that the pixels positions are already approximated (which happens when going from world to screen space ie. rasterization). Perhaps a using the (u,v)-area of contained by the pixel one would make a more precise approximation. Different possible kernel areas are shown on 4.4.1. We could choose to use

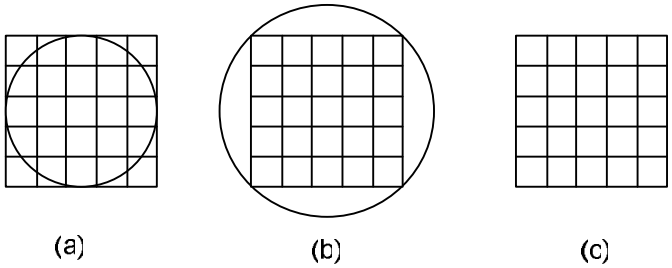

Figure 4.4: Illustration of (a) kernel with area contained by circle (b) kernel with area containing circle and (c) a square filter kernel.

the area contained by a circle to get an accurate result, but we would be wasting

samples. Using all samples and then using the area containing the circle would be inaccurate, so instead we approximate the area, by using the following area :

$$A = \pi(k + 1.5)^2 \tag{4.4}$$

Where k is the kernel radius, we adjust by 0.5 because we want to use the center of the pixel and add 1.0 to increase the area (how good an approximation this makes depends on filter size) and the ratio value is given by:

$$ratio = 1 - r/d \tag{4.5}$$

This value will

$$d = (1 + r)\sqrt{(2)} \tag{4.6}$$

and r is the radius from the center of the central pixel of the kernel :

$$r = \sqrt{(abs(i) + 0.5)^2 + (abs(j) + 0.5)^2} \tag{4.7}$$

This way all the pixel will be weighted, but the filter will be round and be linearly gradient.

4.4.2 Occlusion

First step of the caustic reconstruction as suggested by [3] is rendering the photons to the texture which will be filtered in the final step. First a black version of the scene is rendered to fill the z-buffer with depth values. Afterwards the photon positions is rendered as points. The purpose of this is to get the right depth occlusion for the photons. Rendering the black scene should be relatively cheap, of course depending on the scene, however no lighting or effects need to be calculated.

4.4.3 Quad filtering

The filtering shader is a large shader as it needs to sample a texture for use with the kernel. If a 4x4 filter is used that means 81 samples is needed for the kernel. For each of the samples equation 4.17 is calculated. In addition to this, a sample from either mip-mapping or ray footprint may be used. To reduce the number of pixels for which the shader is run [3] introduces an acceleration method that works by using the stencil buffer in the graphics pipeline.

The idea is to segment the screen with a grid. Each grid cell containing photons needs to be rendered. Finding out what cells needs to be drawn is achieved in two passes.

1. The photons are rendered with the stencil buffer enabled and set to increment on z-pass.

2. Now the stencil buffer is set to always keep the current values. The stencil buffer is also set to only let pixels, where the stencil value is greater than 0, pass. For each grid cell a polygon that fills out the screen space of that grid cell is drawn. During drawing occlusion queries are used. An occlusion query returns the number of pixels that was changed during the beginning and the end of the query.

The information from the occlusion queries is then used during the final rendering stage, where a blend is made between the scene rendered to a texture (without caustics) and the photon texture. Quads are drawn where pixels are to be filtered and the blend between the two textures is therefore only calculated in pixels that covered by these quads. Bent D. Larsen uses points to decide

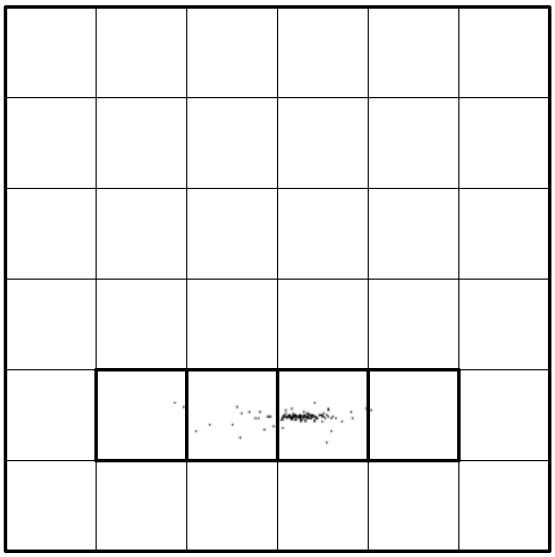

Figure 4.5: Illustration of a grid of quads.

which grid positions contain pixels. This solution is not ideal when zooming in, because as photons disperse one might have grid cells inside the caustic containing no photons, thus these areas will not be filtered at all. We suggest using ray footprints to solve this issue. Rendering discs at pixel positions with radius given by the ray footprint will give a better idea of what grid cells needs rendering. This is important when using coherency improving methods suggested in this thesis, because unfiltered areas will be very noticeable. A constant radius was considered, but with thin caustics with large area, the ray footprints will give a better solution. The drawback to this method is that rendering discs instead of points can be very costly. Two different methods was tried, one using a polygon disc and one using a polygon square with a texture containing a circle (combined with blending). Surprisingly, with this implementation, the polygon method was cheaper and is the one used. This optimization is best when the caustic only takes up a small part of the screen space, which will be the case most of the time. When the caustic takes up most of the screen space, the quads optimization will turn into an extra rendering cost instead. Also noise in the scene will hurt the efficiency of the optimization.

4.4.4 Pixel area

Due to the sampling being in screen space the closer the camera is to the caustic the less photons will be contained by the view frustum. This means that less photons are filtered and the power of the caustic thus is affected by the relative

position of the camera to the caustic. Furthermore the orientation of the caustic in the frustum will also affect the power of the caustic. This is due to the fact that the greater the angle between the surface normal and eye direction is, the more likely a single pixel is to contain a high number of photons.

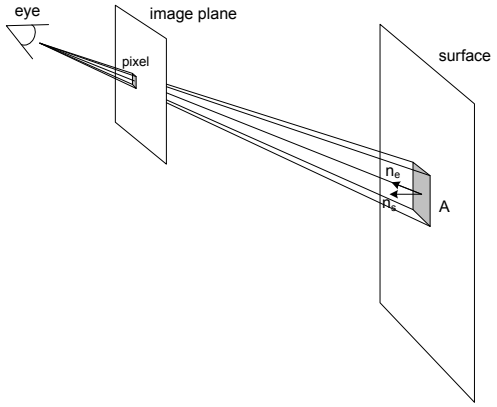

Figure 4.6: Illustration of a situation containing the variables used for the calculation of dot product.

The solution suggested by [3] is to calculate the world space area, A, of the pixel projected onto geometry contained by that pixel. The area is given by :

$$A = (n_s \cdot n_e)\left(4d^2 tan\left(\frac{f_x}{2p_x}\right) tan\left(\frac{f_y}{2p_y}\right)\right) \tag{4.8}$$

where n_s is the surface normal, n_e is the direction from geometry to the eye position, d is the distance from the eye point to the geometry that is contained by the pixel.

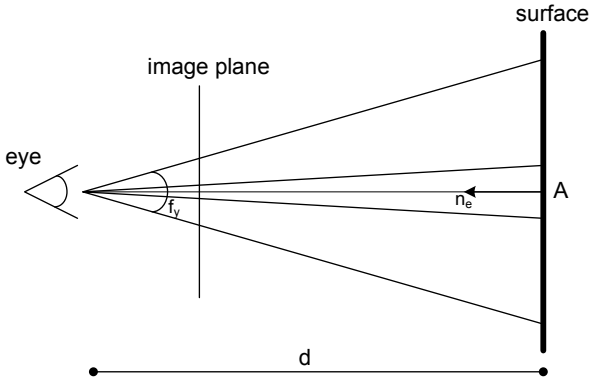

Figure 4.7: Illustration of a situation containing the variables used for the calculation of distance.

This area is used by the final caustic rendering, but we can calculate them in a rendering pass of its own, where the scene is rendered black. The eye point

and scene is naturally the same for both render passes (ie. this pass and the final pass). Due to the fact that we use segmented filtering (which will be described in depth later), the geometry values of the scene is not available at the time of caustic filtering and this is the reason for this pass. The values are rendered to a texture (dot product to one color value and area to another), which is then passed to the final caustic render pass. The calculations are implemented as follows.
Both of the following calculations use the screen space position p_s and this is calculated in a vertex shader by applying the model view projection matrix, M_p, to the world coordinates of the vertex, p_w, as such:

$$p_s = M_p p_w \tag{4.9}$$

This will produce coordinates in the intervals $(x_s, y_s, z_s) \in [-1, 1]$. The rasterizer will interpolate the coordinates for use with the pixel shader. The following calculations will be performed in a fragment shader.

The rotation

The dot product of n_e and n_s is calculated in a pixel shader to achieve better surface precision of n_e. The value, n_e, is the normal from the world position to eye position (which is a constant value, that is passed to the shader).

The area

The depth value, z_s, of p_s after perspective division is given in the interval $z_s \in [-1, 1]$. We will find a new depth value, z, by applying

$$z = \frac{\frac{p_s.z}{p_s.w} + 1}{2} \tag{4.10}$$

The result is a value, $z \in [0, 1]$.

Application

In the final caustic filter the distance value, d, from eye to pixel is brought from the interval [0,1] to world space by calculating

$$d = -\frac{nf}{z(f-n) - f} \tag{4.11}$$

The reason why this calculation is deferred to the final shader is that float point textures hold values between [0,1] so we can transfer the depth value in this way without loss of precision.

The pixel area shader is applied for all pixels in the scene, and one might consider using the quad optimization (described in Section 4.4.3 on page 42) to reduce the number of pixels. This shader is however very simple and the time saved by shading fewer pixels might not outweigh the time spent on rendering the quads.

This method alone does not solve the problems due to loss world position information. When one zooms in on the caustic less photons are filtered and

they are farther apart in screen space positions. This method increases the power of the photons that are filtered with the kernel in the final shader and improves the appearance of the caustic while the eye position is still at a some distance. This optimization simply gives a better looking caustic at different distances, but when zooming in to close the caustic will appear very bright an unfiltered.

4.4.5 Coherency optimization candidates

There are three implementations in the final version of the program.

Empirical Mip-Map method

The empirical mip-map method requires that mip-levels are auto generated for the photon texture upon rendering. It should be noted firstly that this is not a physically accurate method. A fixed number of levels is sampled, we use 6 and these are weighted using functions on the form :

$$f = a\frac{1}{x^b} \tag{4.12}$$

This is the form that the transformations of x,y-takes see Section 2.9. The exact functions used are:

Level 0: $c_0 = \frac{1}{dist^{0.6}} t(x, y)$

Level 1: $c_1 = \frac{1}{dist^{0.8}} t(x, y)$

Level 2: $c_2 = \frac{1}{dist^{1.0}} t(x, y)$

Level 3: $c_3 = \frac{1}{dist^{1.4}} t(x, y)$

Level 4: $c_4 = \frac{1}{dist^{1.8}} t(x, y)$

Level 5: $c_5 = \frac{1}{dist^{2.2}} t(x, y)$

Where the dist value is the same value that was calculated for the pixel area method. These are summed together.

$$c = c_0 + c_1 + c_2 + c_3 + c_4 + c_5; \tag{4.13}$$

As we know from the perspective division equations 2.34 for the screen x and y positions are non-linearly determined by the distance (and it is these functions we are approximating). This means that the dispersion will accelerate the closer the camera is to the object. The slope, b, controls the pace at which the different levels are blended in. Level 0 needs to be filtered, since this is the original level.

However this means that at some point the photon power will saturate the image. Finally using mip-maps in this manner will lead to squarely pre-filtered values ie. blocky appearance. All the calculations take place in a Pixel Shader.

Empirical Splatter method

This is another empirical method that uses a different set of pre-filtered data, which we will generate. The ray differentials are not based on (x,y)-offset as in the theory, but rather a (Θ, ϕ)-offset, which is chosen arbitrarily. Photons are traced and then discs at photon positions are splattered on to a texture with additive blending, so that overlapping photon discs add their energy together. The reason for this is that a pixel that would be the center of a filter kernel would receive energy from the photons contained within the kernel. If we instead considered every photon as the center of a kernel and every pixel contained within these kernels was incremented for each kernel. If the value written to the pixels, p, where the photon count, n, of the pixel divided by area of the kernel A_k combined with additive rendering, it would be the same as using a uniform filter if the kernel radius was the same for both the filter and photon disc. The value, p, is given by:

$$p = s_s \frac{n}{A_k} \tag{4.14}$$

and the final color of a pixel, c, in the texture when all discs where rendered would be

$$c = \sum_i p_i \tag{4.15}$$

Where i represents an index of a photon disc that covers the pixel. In the final algorithm we only use one look-up in the texture so c is the value used. The idea is illustrated on Figure One could use a fixed size radius, but this would lead to very inaccurate shapes. Instead use the ray footprint given by the ray differentials to give the radius of the photon discs. This not perfect either, since the final value is given by the initial choice of (Θ, ϕ). The varying size leads to inaccurate results. In classic photon mapping the photons, that contribute to an area, is divided by that area. In this method each photon is divided by their own area estimate. Also the method we use weights the photon count by their distance from the kernel center, we could consider using the inverse of that weight here, which would give something similar, but we leave this possibility for future work. Several images (varying (Θ, ϕ)-offsets) could be used in the same way as mip-maps, but a photon splatter texture is more expensive to generate. We are rendering discs (using polygons, which is not optimal) instead of points and this can get quite costly with many photons emitted.
A drawback to this method is that we are working in world space with discs that rendered independently from geometry. This results in the method not working correctly for receiving surfaced to which the entire disc cannot adhere.
Weighting is handled in a simple manor, where the normal filtered color and the sampled value from the splatter texture are linearly interpolated by an arbitrary fraction. The pixel area will saturate the image with the color, which means that at some point the splatter value will become more and more apparent in the image.
The splatter texture is passed to Pixel Shader which handles the blending.

Variable Area by Mip-Maps

The classic photon map technique varies the area in its radiance estimate to find a desired number of photons, due to the speed requirement we are not

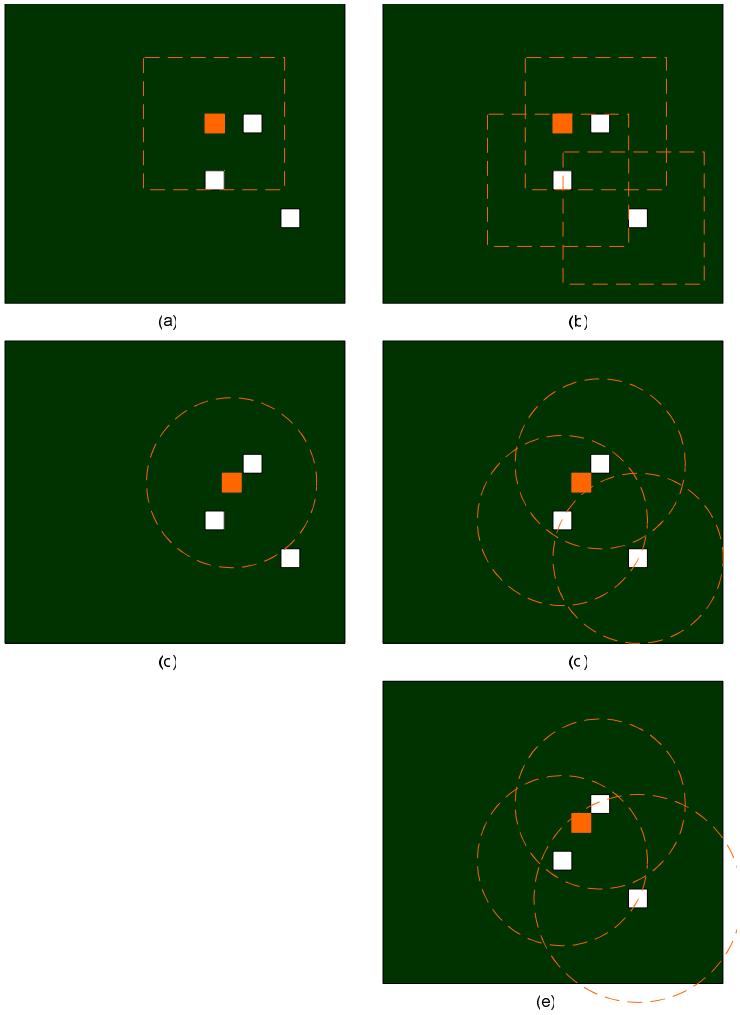

Figure 4.8: Illustration of the idea, where the white square a pixel containing photons. The red squares are the kernel centers and the red stippled shape is the filter outline. The images contain (a) a square filter kernel, (b) a square splatter area, (c) a round filter kernel, (d) a round splatter kernel and (e) a round splatter kernel with varying radius.

able to just scale the filter kernel. Mip-maps provide information about how many photons are contained in the area of size depending on the Mip-level. The original texture level is 0 where pixel's holds values of an area we consider 1 to 1. At level 1 the area is 4 to 1, at level k the area is 2^{2k}. These level have been down sampled using a uniform average. The photon texture will have mip-map levels auto generated possibly as show on 4.4.5, however whether fractions are floored or ceiled is dependant on hardware. The idea is then that going up a level could be the same as expanding the search radius. If we expand the search radius, until a maximum radius is reached or we have found a photon value.

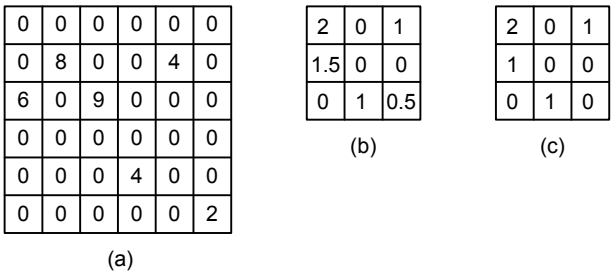

Figure 4.9: Illustration of mip-mapping of photons. Part (a) is the level0 image, (b) is the level1 image in floating points and (c) is the level1 image in 8-bit, where the values are floored.

The photon value returned by a texture lookup (with bilinear sampling) is

$$c(x,y) = \frac{1}{A}\sum_{i=-k}^{k}\sum_{j=-k}^{k} t(x+i, y+j) \tag{4.16}$$

Where A is the is the area of the filter used to create the level given by 2^{2n} for level n. By using trilinear interpolation for texture lookups between the levels one can get blended values of the different levels. The color value of trilinear value is

$$c(x,y) = wc_{level(n)}(x,y) + (1-w)c_{level(n-1)}(x,y) \tag{4.17}$$

where w is the weighting factor (or depth). The value of w is determined by looping through the levels of the mip-map until values are found, if any are found at all. The current method skips an entire level at the time. This will produce a crude appearance. One does not have to skip one level at a time, 10 tests per level where tested, this means 50 samples are need for 10 levels, which is unacceptable. Another method that was considered was using several loops, where the first loop skips an entire level to find the two levels between the interpolation takes place, then another loop uses a finer division between only these two levels. This would accomplish the same precision except it would "only" cost 20 samples, this is quite a bit considering that the standard filter of 4x4 uses 81 samples. The algorithm is as show on Figure 4.4.5.

```
texcoords.w = 0
maxlevel = chosen max level
desired = chosen desired value

value = sample level 0 using texcoords
while(w less than level && value < desired)
{
    texcoords.w++;
    value = sample level++;
}

color = sample the tex using filtering and LOD samples with texcoords
color *= scale
```

Chapter 5

Implementation

5.1 Overview

An overview of the implementation is given here in two UML diagrams. In Figure 5.2 the diagram shows the class structure of the main functionality (ray-tracing, rendering and photon-mapping) centered around the RayView class. In Figure 5.1 the diagram shown is the class structure of the Graphical User Interface. Below the figuret's are short descriptions of the purpose of each class.

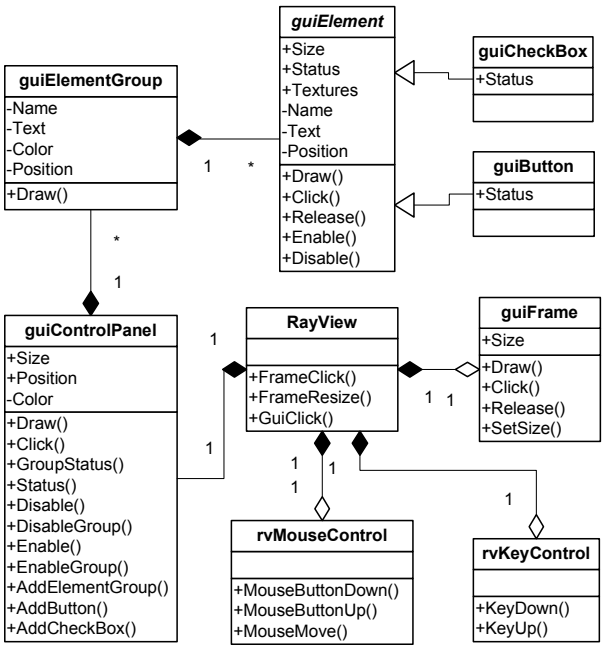

Figure 5.1: UML diagram for the GUI of RayView

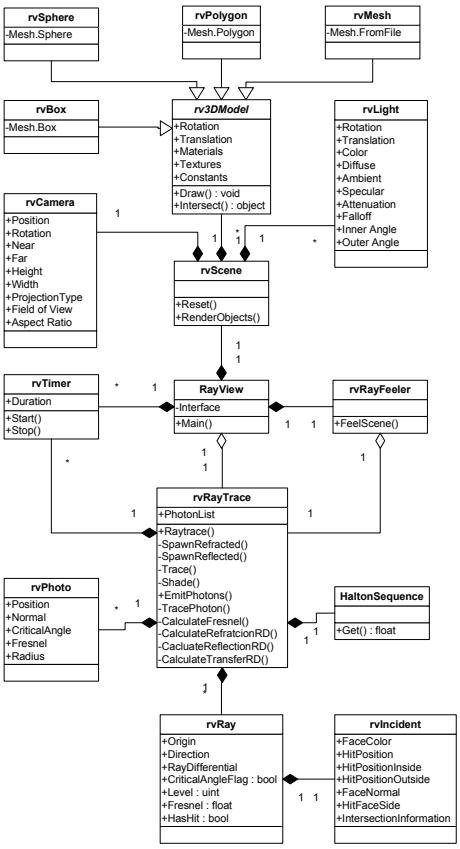

Figure 5.2: UML diagram for the main functionality of RayView

5.1.1 Main classes

The main classes are considered the classes pertaining to the generation of the resulting image. Also the Ray Feeler class has been included here.

RayView

RayView is the class that contains the Main() function and the Render loop. It also acts as the binding link between the different Interface event and the Graphic User Interface.

rvRay

The rvRay class contains RayViewt's representation of a ray used for raytracing. It is used in Trace calls and if intersection occurs the information pertaining to incidents are written to the ray and used when spawning a new ray, if max depth has not been reached. The rvRay also contains information related to Ray Differentials ie. a differential Position and Direction. The Fresnel coefficient, is accumulated during a rays traversal through a scene.

rvPhoton

The rvPhoton class is RayViewt's representation of a photon. This is used when a photon is stored as surface. It contains the storage position and surface normal at the stored position. It also contains the accumulated Fresnel coefficient and final values of the Ray Differentials.

rvIncident

The rvIncident class is created upon a rays intersection with a surface. It contains all the information needed to calculate spawned rays, shading and Fresnel coefficients at an incident.

rvTimer

The rvTimer is the timer class used for generating the results. It is able to measure time with a precision of less than 1 ms. Start begins the measurement, Stop end the measurement and Duration is the resulting time span.

rvCamera

The rvCamera class is RayViewt's representation of a camera, it holds and calculates information needed to set up projection and perspective transformations.

rvLight

The rvLight is the basic light class. The support light types are spot and point light sources.

rv3DModel

The rv3DModel class is an abstract class used to define the different Mesh classes. Each class contains information regarding transformation, appearance and surface constants (including reflection index).

rvSphere

A mesh class derived from rv3DModel. This class contains a Mesh.Sphere object.

rvMesh

A mesh class derived from rv3DModel. This class contains a Mesh.FromFile which is able to load .x files. The .x format is Microsoftt's object format and plug-ins for importing and exporting this format exist for most major graphics programs such as Maya and SoftImage.

rvPolygon

A mesh class derived from rv3DModel. This class contains a Mesh.Polygon object.

rvBox

A mesh class derived from rv3DModel. This class contains a Mesh.Box object.

rvRayTracer

This is the class that supplies a standard Monte Carlo Ray Tracer and also contains the photon tracing capabilities of RayView. After a call to EmitPhotons, the resulting photons are contained by the rvRayTracer class.

rvRayFeeler

The rvRayFeeler class was implemented and used during the development and has been left in. It can trace a ray or photon through a scene and show the resulting path of the ray and its ray differentials.

HaltonSequence

This is RayViewt's implementation of Halton Sequences as discussed earlier. It provides the ray tracer with random values in the interval [0,1] for use with photon emission.

5.1.2 Graphical User Interface classes

The classes used in the GUI and for creating the user interface.

rvMouseControl

This contains the mouse event handler for the Windows Form ie. RayView. It also contains RayView and upon receiving an event it calls the methods in RayView to handle the event.

rvKeyControl

This contains the key event handler for the Windows Form ie. RayView. It also contains RayView and upon receiving an event it calls the methods in the RayView class that corresponds to the pressed key.

guiFrame

This is a GUI element that controls the area of the screen that will ray traced. It is used only in the Ray Tracing mode of RayView.

guiElement

This is an abstract class that defines the different guiElements (guiCheckBox and guiButton). The Status method of guiElements are used by RayView during the rendering loop to check different settings.

guiCheckBox

This is a check box.

guiButton

This is a button.

guiElementGroup

This class contains a list of guiElements and when drawn to screen the guiElementt's contained by this guiElementGroup are drawn appropriately as well.

guiControlPanel

This is the main GUI class. It contains a list of guiElementGroups, which in turn contain the guiElements. When the guiControlPanel all parts of the GUI is drawn. The elements are drawn in the order added to the guiControlPanel and guiElementGroups.

5.2 Direct3D and HLSL

The DirectX 9.0 SDK that was used during development and testing was the October 2005 release. The shader version targeted was Shader Model 3.0. It would have been nice to implement the shaders to support Shader Model 2.0. However Pixel Shader 2.0 is limited to 32 texture instructions and 64 arithmetic instructions, which is insufficient considering that with a filter of radius 2, 25 texture lookups would be needed. Pixel Shader 2.x promises the possibility of unlimited texture instructions, but can only guarantee 96 arithmetic instructions, which i not enough for our complex filtering algorithm. Pixel Shader 3.0 guarantees a minimum of 512 arithmetic instructions as well as unlimited texture instructions.

Due to Managed DirectX being new, the normal DirectX re-distributable installer does not install the files needed for Managed DirectX support. If the command line switch

```
/InstallManagedDX
```

is used the files will be installed. The Managed DirectX environment at this point is also undergoing large changes and support between versions is far from guaranteed. This was developed with the October 2005 release of the DirectX SDK. The dllt's used are

- .../1.0.2902.0/microsoft.directx.direct3d.dll
- .../1.0.2909.0/microsoft.directx.direct3dx.dll
- .../1.0.2902.0/microsoft.directx.dll

When using C# and .NET the .NET Framework 1.1 (in this case) is required.

The Managed DirectX 9.0 October release (which includes the DirectX Redist MSI installer), which was used is included on the cd that is supplied with this paper as well as the .NET Framework 1.1 installer.

5.3 Mesh class

The Mesh class, along with ease of coding, was the main reason that Managed DirectX and C# was chosen for this project. The Mesh class delivers two important things, support for any mesh (such as the Cognac glass we use) and an Intersection method. Mesh classes can represent standard models such as a Sphere, Box, Cone, Polygon (read: mesh surface) etc. with standard implementations in DirectX. It can also load .x files into a mesh and will support intersections for these. The Mesh class has it's own draw function and also has advanced functions such as generation of tessellation levels and cloning for instancing.

5.4 Microsoft .x file format

The .x file format is Microsoft 3D model class. It can contain a mesh model, animations, materials, texture and more. Most major graphics illustration software, such as Maya, SoftImage and 3d Studio Max either has standard support for the file format or have plug-ins available to add support.

5.5 AutoMipMapGeneration

AutoMipMapGeneration is key to the use of Mip-Maps, this feature is not currently supported for 32-bit floating point textures, but only 8-bit texture. The result in stored photons can quickly erode completely from an image, which will happen as soon as a value is less than 1. How quickly depends on how dense the distribution of photons is since mip-mapping uses an averaging filter on values. In computer graphics greater color resolution is a active topic of development and in the future the author hopes auto mip-map generation will be available for 32-bit textures.

5.6 Graphical user interface

Here descriptions of the different options in the graphical user interface will be given.

Modes

The first element group is modes, where RayView is set to the desired mode. The mode of interest is photon-map, the others where necessary steps in development.

Rendering

This mode set RayView to ordinary rendering (using the standard pipeline) of the selected test scene.

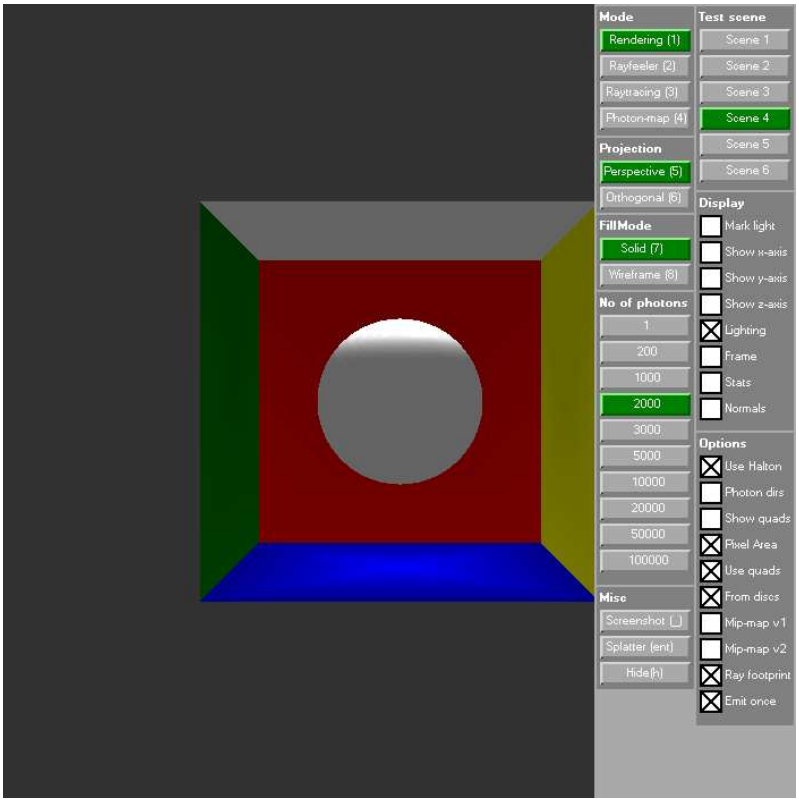

Figure 5.3: Image of the application and GUI.

Rayfeeler

This mode uses the ray tracer to emit a single ray into the chosen test scene. The resulting path is displayed in the scene (which is rendered in a standard fashion) and the ray differential paths are also drawn.

Raytracer

This mode ray traces an image. It takes a long time and no status is shown, so one should be aware that the program has not frozen if it appears so after selecting this mode, it is simply working hard. The frame can be used to decide how much of the window area is to be rendered.

Photon-map

This mode blends a caustic into a scene rendered in a standard fashion. This is the where the results of the thesis can be seen.

Projection

The projection category is used to decide on the type of projection used, the two options are perspective and orthogonal projections. This also affects standard ray tracing.

Fillmode

This chooses the fill mode. Solid or wire frame. This does not affect ray tracing.

No of photons

Since a text input was not implemented, options for the number of photons emitted for the photon-map has been added. The number of photons range from 1 to 100,000.

Test scene

Chooses from the different test scenes that have been used for testing. In the result section each of these scenes will be described in more detail.

Misc

Under misc the are options to take a screenshot, save the current splatter and hide the gui. The screenshot is generated by saving the current backbuffer to file, which means that everything rendered to the window is captured (including the GUI). The splatter is the texture generated for use with one of the caustic generation methods.

Options and Display

There are several options in these two categories:

Mark light if checked the light sources contained by the chosen scene will be marked with a sphere to indicate position and a line to indicate a direction.

Show x,y and z-axis if either of these are checked the relevant axis are shown.

Lighting if checked will enable the render pipelines Phong illumination for the scene.

Frame if checked the frame for ray tracing, will be shown and can be resized by click and dragging it.

Use Halton if checked Halton sequences will be used to generate directions, otherwise the system random generator will be used.

Photon dirs if checked photon paths of stored photons will be shown. This must be enabled at the time of emission to occur.

Initial dirs if checked the initial photon dirs of photons will be shown. This must be enabled at the time of emission to occur.

Normals if checked the normals of surfaces will be shown.

Stats if checked different statistics for the scene will be shown in the top left corner. The time measurements are averages of 10 (this value is a constant in the program) renderings.

Show quads if checked the quads, used for accelerating the filtering, will be drawn as black squares over the scene.

Pixel Area if checked the pixel area method will be used.

Use quads if checked the quad acceleration method will be used.

From discs if checked the quads will be generated using discs, rather than photons.

Use mip-map if checked the mip-map method will used.

Ray footprint if checked the splatter (ray footprint) method will be used. Will only be used is mip-map is unchecked.

Emit once if checked photons will only be emitted once and then reused the following frames.

Chapter 6

Results

The emission part of the algorithm can be disconnected from the filtering part of the algorithm since at some point most emission methods generate intersection points in space. This enables us to use a simple ray tracer quality testing and evaluate how the algorithm would work with an advanced ray tracer by using results from a different implementation.

The implementing and testing will take place on a system with the following characteristics:

- The CPU is a Pentium M 1.9 GHz.
- The graphics card is an nVIDIA GeForce Go6800
- The OS is Windows XP
- The editor is Microsoft Visual Studio .NET 2003
- The SDK is Microsoft Managed DirectX 9.0 (see more details in the Implementation chapter)
- The programming language is C# and the shader language is HLSL
- The resolution is fixed at 700x700.

We will use the performance measurement tool PIX supplied with the DirectX SDK to measure most values. The rvTimer class will be used to measure some times in milliseconds.

6.1 Emission

In our tests we will use our own results and extrapolate what results would be possible with a faster Ray Tracer.

6.1.1 Reference Ray Tracer : OpenRT

The ray tracer we will use as a reference is Wald's OpenRT as described [7]. The OpenRT system uses a selection of the optimizations some of which are

described in short in the Appendices. The OpenRT ray tracer has been tested on a 2.5 GHz Pentium IV notebook processor. Using scenes, with the following polygon count, with an unknown, but a seemingly high amount of objects (see Wald's paper for images). The complexity of the scene is given in Figure 6.1. Wald includes some performance tests for these scenes, we've plucked a few,

Name	#Triangles
ERW6	804
ERW10	83.600
Office	34.000
Theater	112.306
Conference	282.801
Soda Hall	2.247.879
Cruiser	3.637.101
Power Plant	12.748.510

Table 6.1: The scenes used by Wald for testing the OpenRT ray tracer.

which are shown on Figure 6.2. Here we only include the results for pure ray

Scene	FPS @ 1024x1024
ERW6 (static)	8.95
ERW6 (dynamic)	4.00
ERW10	5.82
Office (static)	4.68
Office (dynamic)	2.61
Theater	2.68
Conference (static)	4.40
Conference (dynamic)	3.17
Soda Hall (in)	3.68
Soda Hall (out)	4.47
Cruiser	3.38
Power Plant (in)	1.43
Power Plant (out)	1.59

Table 6.2: These are some of the results of the OpenRT ray tracer. The values are Frames Per Second.

tracing (ie. no shading), but results for shaded ray tracing are also given in Wald's thesis. The average of these value is :

$$FPS_{avg,openrt} = 3.9123 \tag{6.1}$$

The algorithm scales quite well for increasing scene complexity so using the average should be acceptable. However Wald is testing ray tracing and shooting a large amount of rays, that is 1.000.000 rays. This is very likely more than is needed to generate caustics and in the test program the maximum is 100.000. If we assume that cost scales linearly with the number of rays an average for the number of photons we can test are given on Figure 6.3. These are very fast

No of rays	∼ FPS	∼ time to trace
1	$30.0123 \cdot 10^5$	0.0003
200	$15.0650 \cdot 10^3$	0.0664
1000	$30.0123 \cdot 10^2$	0.3332
2000	$15.0615 \cdot 10^2$	0.6639
3000	$10.0400 \cdot 10^2$	0.9960
5000	600.0246	1.6666
10000	301.23	3.3197
20000	150.6150	6.6494
50000	60.2460	16.5986
100000	30.123	33.1972

Table 6.3: These are some of the results of the OpenRT ray tracer. The values are Frames Per Second and the time it takes to trace the rays in milliseconds.

tracing times as we shall discuss, when we below compare OpenRT to the ray tracer created for this thesis.

6.1.2 Thesis Ray Tracer

The implementation of the ray tracer for this thesis is a simple one, and the interesting thing about it, is the performance of the Mesh.Intersect method. The test scenes have between 1 and 5 objects and thus scene hierarchies would do little to improve our purpose.
In Figure 6.1 images of the test scenes rendered using standard rendering with Phong shading, are shown in Figure The complexity of the test scenes are shown in Figure 6.4. The results of emitting a tracing 2000 rays in the different rays

Name	#Triangles
Single sphere	4900
Sphere & Box	1768
Cognac glass	5700
Sphere in Box	4920
Brass Ring	644
Wave plane	2010

Table 6.4: The scenes used in this thesis and their polygon count.

through the scene are given on Figure 6.5. Column shows the average level reached by the rays. The ray level can average below 1 because the ray starts at level 0 and level 1 is reached only after a ray is either reflected or refracted. For a perhaps more natural measure add 1 to the values in column one. Our emission scheme means that rays that hit a non-specular surface on the first bounce will be eliminated, therefore many rays are eliminated early on. It should also be mentioned that we are using a spot light source to emit the photons, but the spots are not perfectly adjusted.
The second column is the average time of the tracing the full ray paths. This value changes roughly with the complexity of the objects, but it also seems that

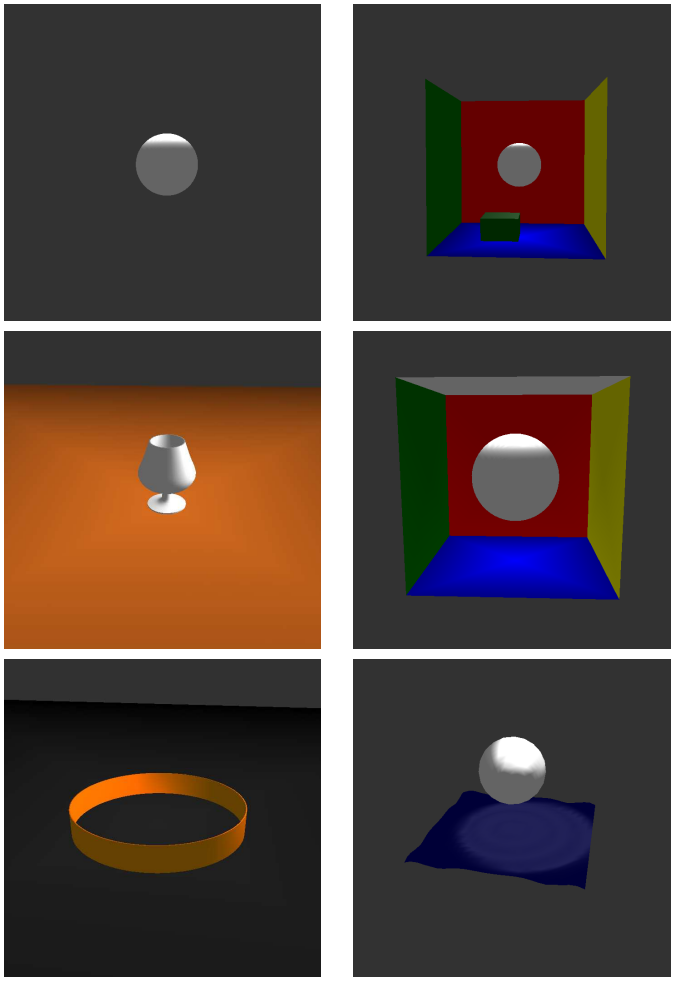

Figure 6.1: Top-left: Single sphere, Top-right: Sphere & Box, Center-left: Cognac glass, Center-right: Sphere in Box, Lower-left: Brass Ring, Lower-right: Wave

the object type has an effect. The cognac glass is only a bit more complex than the single sphere (from triangles, 116%), but it takes much more time to trace (448%). The cognac scene also contains a plane, but this plane is only 2 flat polygons.
The third column shows the total time to trace a ray, while also calculating ray differentials for two offset rays (none where calculated in column). These values follows the values from column two as expected, and are only slightly lower

$$RD_{cost} \approx 0.0025 = 1.1353\% \tag{6.2}$$

The fourth column is the average time of the intersection tests in the scene, no data to compare these with where available for this thesis. These are not pure measures of the Mesh.Intersect method, but includes inverse transformations of the ray using the vector and matrix algebra package that comes with DirectX.

Name	ray level	trace ray	no ray diff	intersect	total
Single sphere	1,0336	0, 1059	0,1016	0, 0828	140,3572
Sphere & Box	0,6662	0, 0900	0,0814	0, 0222	344,9263
Cognac glass	0,5185	0, 4740	0,4718	0, 3809	718,7494
Sphere in Box	0,0401	0, 0074	0,0072	0, 0622	284,7446
Brass Ring	0,0625	0, 0178	0,0176	0, 1029	65,3781
Wave plane	1.2583	0, 3325	0,3252	0, 0627	275,5466

Table 6.5: The results of testing the ray tracer is given in the table. 2000 rays where emitted and the values are averages of the times measured. 'Ray level' is the average depth of a ray, 'trace ray' is the average time to trace a single ray, 'no ray diff' is the same without calculation ray differentials, 'intersect' is the average time of the intersection tests and 'total' is the total cost of tracing the 2000 rays. Times are measured in milliseconds.

The fifth column is the total time of tracing 2000 rays with the thesis ray tracer. From previously (Table 6.3) we have the estimated value 0.6639 ms. This result is far smaller than any of our values as seen in Table 6.6. What we are interested

Scene	Name	%
1	Single sphere	99.53
2	Sphere & Box	99.81
3	Cognac glass	99.91
4	Sphere in Box	99.77
5	Brass Ring	98.98
6	Wave plane	99.76

Table 6.6: How many percent faster OpenRT would likely be compared to the thesis tracer. Of course the OpenRT value used is an interpolation and in practice a direct interpolation is not realistic.

in for the remainder of the results are how much time, after emission, is left for filtering. FPS is a frequency measure and means Frames Per Second. FPS is a common value for measuring the speed of an graphics application, however we will measure in ms. FPS is calculated by

$$FPS = 1/t \tag{6.3}$$

Where is divided by the time in seconds. If we set a goal of 30fps we calculate the milliseconds at our disposal as

$$t_{max} = 1/30 \approx 0.0333s = 33.3ms \tag{6.4}$$

After emitting photons we should thus in theory have the disposable times given in Table 6.7. These values are based on the values in Table 6.3. These values are the basis of comparison for the rest of the thesis. For different applications there will be different expectations with regards to speed. In computer gaming a single part of the full algorithm cannot be allowed to take up a 33.3ms. Caustics is considered just an effect and would probably not be used in a framework, where it was the only effect.

No of rays	$t_{disposable}$	$\sim$ time to trace
1	33.2997	0.0003
200	33.2336	0.0664
1000	32.9668	0.3332
2000	32.6361	0.6639
3000	32.3040	0.9960
5000	31.6334	1.6666
10000	29.9803	3.3197
20000	26.6506	6.6494
50000	16.7014	16.5986
100000	0.1028	33.1972

Table 6.7: How many percent faster OpenRT would likely be compared to the thesis tracer. Of course the OpenRT value used is an interpolation and in practice a direct interpolation is not realistic. Time values are in milliseconds.

6.2 Filtering

The two main categories that we consider when evaluating the caustic filtering algorithm are performance and appearance. First we will look at the standard filtering method with no optimizations. Then we will review the results for each part of the algorithm, including each of the suggested methods. We use 5 of the 6 test scenes from previously, we drop Scene 1 because the scene contains no diffuse surfaces. Since filter takes place in image space and we do not use quads we will always be filtering the same amount of pixels ie. 700x700 pixels. For this reason we need only test filtering speed with one scene.

6.2.1 Basic filtering

Basic filtering means that the Pixel Area area will be disabled. So will all of the coherency optimizations. Quads will be enabled. For the irradiance estimate we need to generate the caustic texture and then filter it. We emit photons once and then filter with only quads enabled in some cases. The basic filter is primarily affected by how much of the screen area is to be filtered, we can see the percentage filtered by showing the quads. However when generating the photon texture, we are rendering the scene and the cost of this may very depending on camera position. Also the number of photons would affect the rendering time of the photon discs used to determine the quads. We chose to use only Scene 4 and two different amounts of photons with filter kernel of size, 4x4. on figure 6.2. The results of the test are shown in 6.8. On Figure 6.3 we show the images saved at the positions, where the stats was gathered. As expected the number of photons have minimal impact on the performance of the filtering algorithm. The quad optimization is already hinting that it is valuable. Table 6.9 shows the values if we add the appropriate OpenRT tracing times from Table 6.3. These are the estimates of the frame rates the algorithm should run at using OpenRT for photon emission. With the added cost the cost of the basic filter is reasonable.

Quality of the caustic is fine given an appropriate amount of photons and

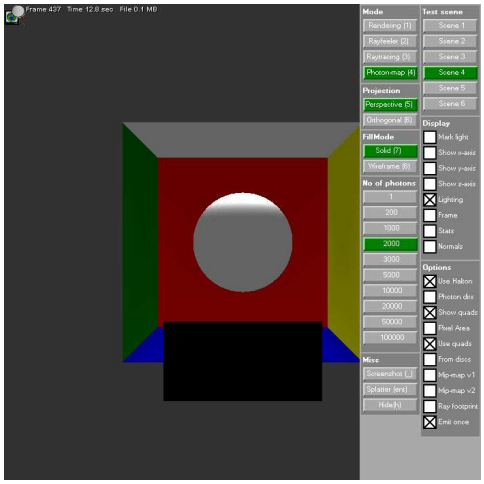

Figure 6.2: The setup for the basic filter test.

# Test	# Photons	Quads	Screen%	# Pixels	FPS	Duration (ms)
1	2000	2	5.56	27440	53.12	18.82
2	2000	8	22.22	108878	49.68	20.13
3	2000	36	100.00	490000	19.04	52.52
4	20000	2	5.56	27440	46.21	21.64
5	20000	8	22.22	108878	47.05	21.25
6	20000	36	100.00	490000	20.05	49.87

Table 6.8: The screen percentage is calculated from the number of quads, which is set to 6x6 during the tests and the screen size is 700x700. The FPS is measured using PIX, and averaged over the frames, and the rest is calculated from the other two values.

#Test	~ Duration	~ FPS
1	19.49	51.31
2	20.79	48.10
3	53.18	18.80
4	28.29	35.35
5	27.90	35.84
6	56.52	17.70

Table 6.9: The estimated runtime values using the quad optimization in some cases.

distance. If one zooms in the expected results is a visible change in the power of the caustic. From a distance many photons are considered in a few pixels and thus, while up close few photons are filtered. Some caustics in scene 2 is shown on figure 6.3. These images also show how this method in a natural way handles non-planar surfaces like the corners of a box. Another problem is rotation of

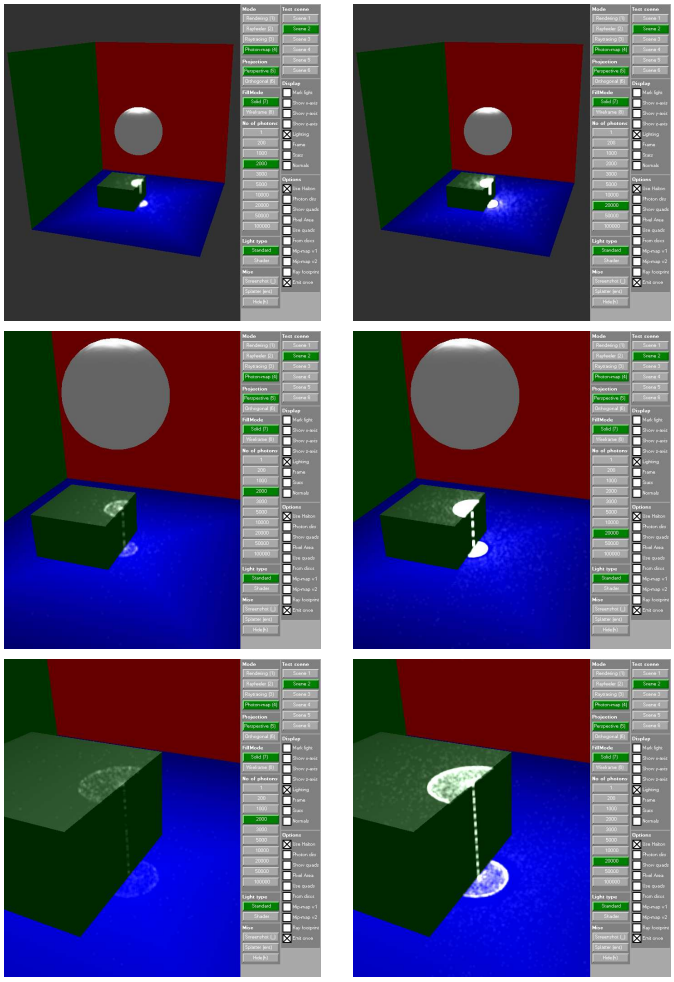

Figure 6.3: (Distance,Photons) Top-left:(1,2000), Top-right:(1,20000), Center-left:(2,2000), Center-right:(2,20000), Lower-left:(3,2000), Lower-right:(3,20000)

the camera about the caustic, this will be shown in the next section.

6.2.2 Pixel Area

The pixel area part of the algorithm is an empirical method used to scale the caustics power with regard to distance. Ideally this would be replaced by the coherency optimizations, but this has not been done in this implementation. Figure 6.4 shows It is clear that the power of the caustic appears less affected by distance and rotation. It however only scale the filter values, meaning that it does not solve coherency problems.

We will test the cost of pixel area by using standard filtering and quads generated from points. The pixel area algorithm should be impacted by the number of pixel that contain geometry, we will test this by zooming in on the

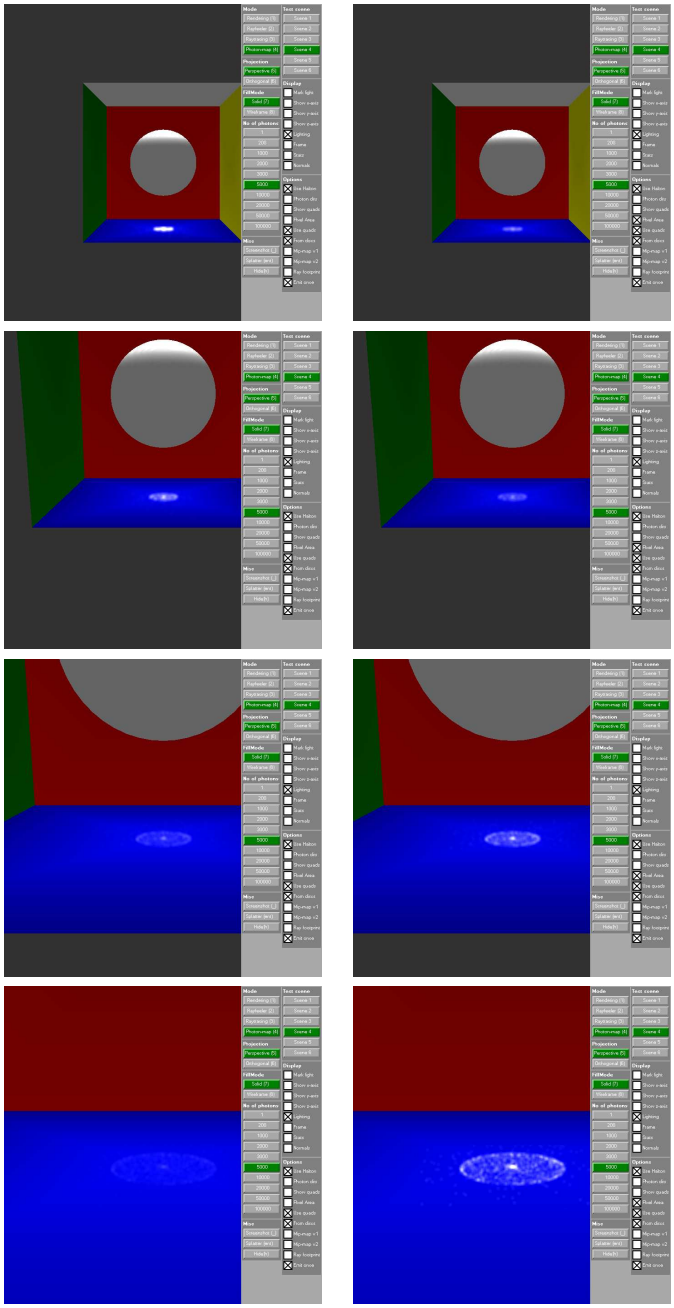

Figure 6.4: Are series of images where the left column contains images, where pixel area was not used and in the right column the images are rendered using the pixel area optimization.

contents of Scene 4 and measure the speed with and without pixel area enabled. The setup is shown on Figure 6.5. Table 6.10 shows the results of test at the

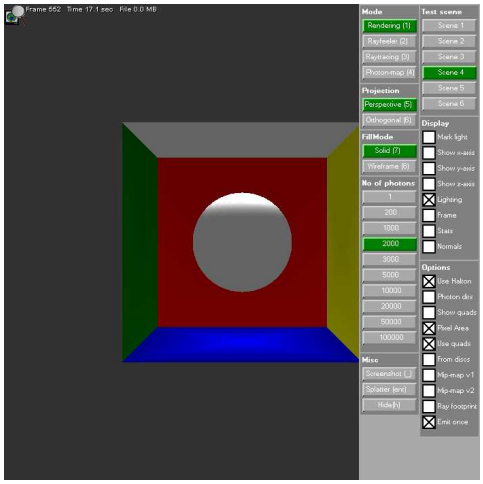

Figure 6.5: The setup for the basic filter test.

top (Test 1) and bottom (Test 2) positions show in Figure 6.3. It should be

#Test	~ Duration w PA	~ Duration wo PA	Difference
1	32.69	24.46	8.23
2	27.49	23.79	3.70

Table 6.10: The estimated runtime cost (in milliseconds) of using the pixel area optimization.

noted that the values are averages of 500 samples each. It seems clear and reasonable that there is a cost associated with using the pixel area. However it does not seem a high cost and the variance in the rendering calls add great noise to the data. Test 2 should in theory cost more to calculate than Test 1, but the variance seems to be too great to get a clear measurement. If we check the rvTimer in the program it estimates that the pixel area pass costs around $0.1ms$ and this would naturally not be noticeable with the values we found with PIX.

6.2.3 Quads

The quad optimization is expected to increase the speed of the filtering process, without harming the appearance of the caustic. An issue with using single points to generate quads is that at some point they will become incoherent and if grid cells are small enough they may be empty. Figure 6.6 illustrates an image filtered using original quads and the quads rendering as suggested in this thesis. Figure 6.6 shows how a combination of finely divide quad grid and point rendering can cause unfiltered holes in a caustic, when zooming in. Using photons discs does not guarantee a perfect solution, since the filtered area depends on kernel size, rather than the ray differential size. It should however be sufficient for most situations.

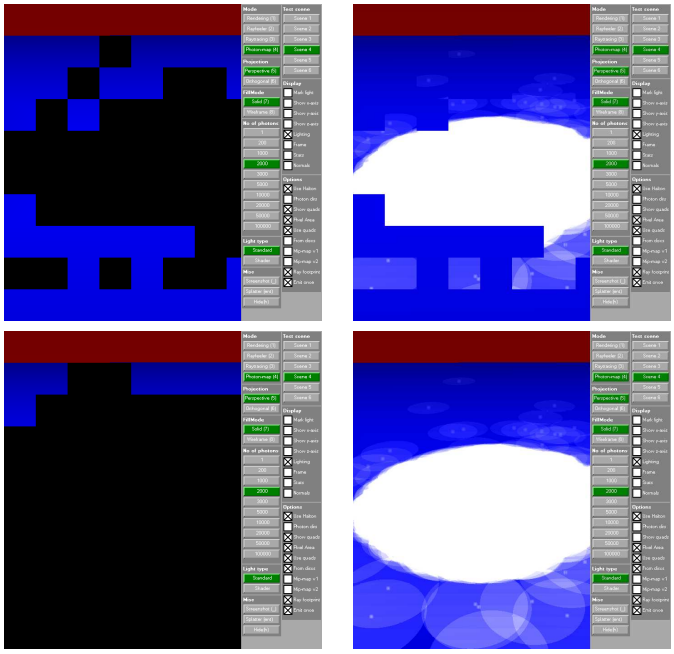

Figure 6.6: The four images contain (topleft) contains the quads generated with points, (topright) contains the image generated using the quads from topleft, (bottomleft) contains the quads from photon discs and (bottomright) contains the image rendered using bottomleft quads.

Quads also improve speed and Figure 6.7 shows how the amount of screen space is rendered depends on the distance from the caustic.

We will test the effectiveness of the optimization by rendering Scene 4 with quads enabled with discs, photons and disabled. We will disable pixel area and coherency optimizations. We will test with a different amount of photons in the scene, since this should have a great effect on whether using discs is actually an optimization in it is current implementation. Table 6.11 shows the results. The

#Test	# Photons	Duration wo Quads	~ Duration w Photons	~ Duration w Discs
1	2000	49.95	27.72	32.14
2	20000	49.97	28.75	107.94

Table 6.11: How quad affects the FPS in at different settings. The values are times of total rendering in ms. Emission takes place once.

values indicate that discs are more expensive than photons. Photons for quad generation will be an optimization over no acceleration in most cases ie. it seems to handle large amounts of photons fine. Discs however become very expensive at a relatively small amount of photons (most scenes would likely require more than 2000 photons to create caustics in), this would have to be solved for the method to be truly useful. It is not perfect either, if using a different filter that

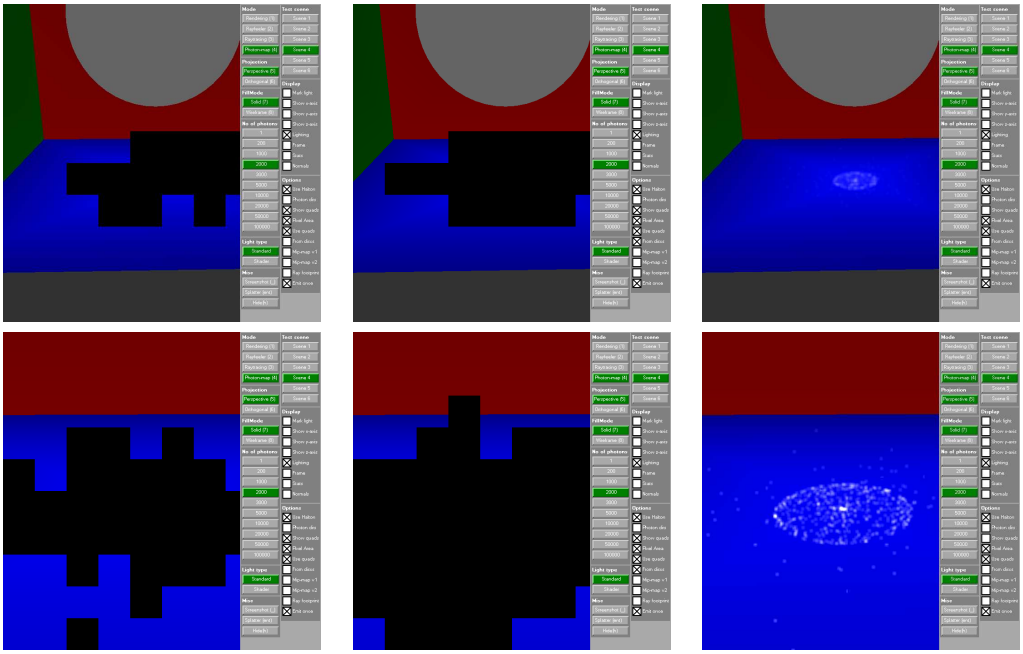

Figure 6.7: The four images contain (topleft) contains the quads generated with points, (topright) contains the image generated using the quads from topleft, (bottomleft) contains the quads from photon discs and (bottomright) contains the image rendered using bottomleft quads.

produce a different (larger) caustic it could still be broken by the quads. This can be the case with the mip-maps, but the method is naturally ideally suited for the splatter method, since they use the same ray footprints.

6.2.4 Empirical Blend with Mip-Maps

Empirical mip-maps was a simple method used to see what the auto generated mip-maps would look like. The result are given for Scene 1 to 6 in Figure 6.8. The result are caustic with increased power, which is caused in part by the blending functions used and in part by the application of the pixel area optimization (saturation would occur without pixel area, just slower). Also the shape is not accurate, whether this is acceptable depends on the user. The method does create coherency, but the images are not physically accurate. However they do treat the surface more correctly, since it is based on the original photons. However it is not perfect since the mip-map does not use depth to see if it overlaps other geometry as seen with the brass ring on Figure 6.8. The cost of the rendering will be measured using quads, pixel area and empirical mip-maps (Mip-maps v2) enabled. The different scenes are tested in a close up situation as shown on 6.8. The results are shown on Table 6.12. The being close results in the quad optimization not giving much of a speed up, this results in a lower FPS. The FPS is not considered real-time, but is still interactive. At a distance the FPS is real-time with the method turned on.

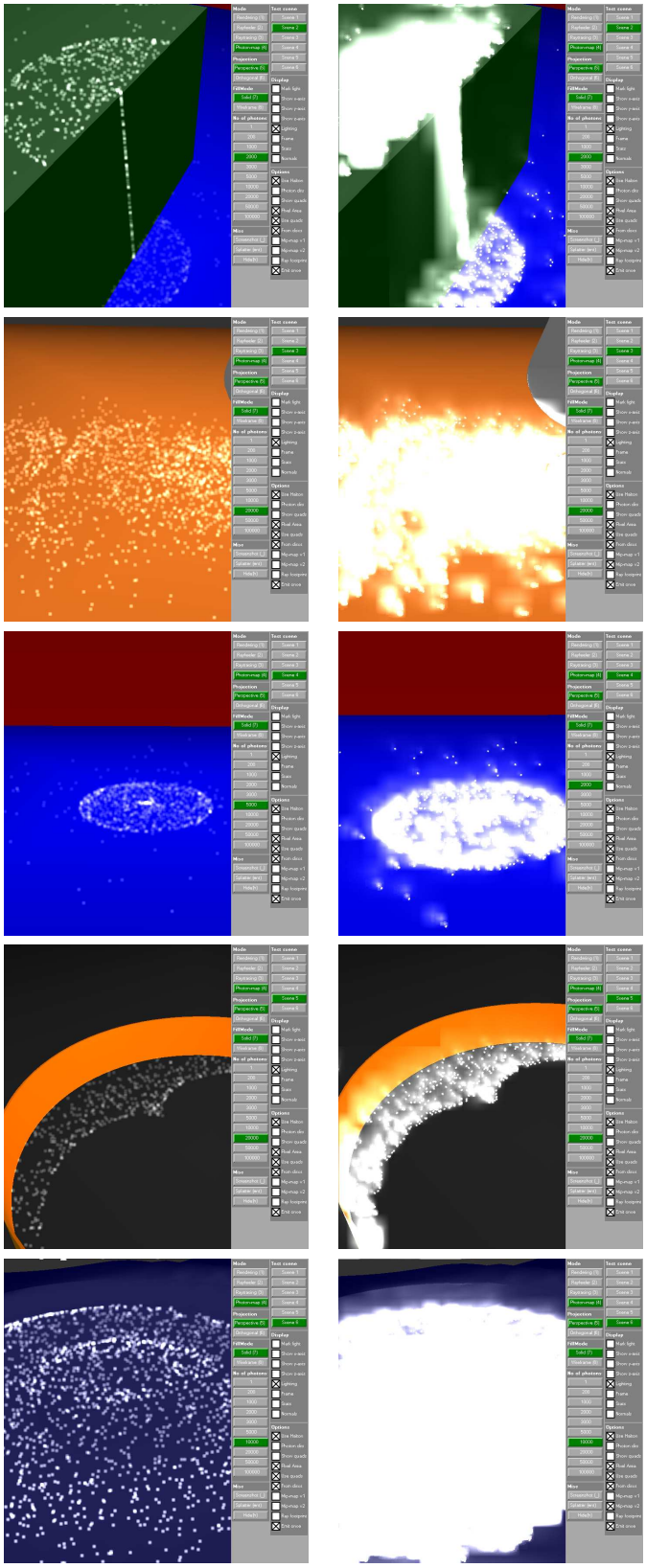

Figure 6.8: The quality test for empirical mip-maps.

# Scene	# Photons	FPS	Duration	FPS w OpenRT
2 (near)	5000	14.99	66.71	14.62
2 (far)	5000	28.98	34.50	27.65
3 (near)	5000	15.01	66.62	14.64
3 (far)	5000	32.56	30.71	31.48
4 (near)	5000	14.56	68.68	14.21
4 (far)	5000	29.60	33.78	28.21
5 (near)	5000	19.48	51.33	18.87
5 (far)	5000	30.62	32.66	29.13
6 (near)	5000	14.84	67.39	12.41
6 (far)	5000	29.33	34.09	27.97

Table 6.12: The rendering times of running the application with the Empirical Mip-map method

6.2.5 Empirical Blend w Photon Discs

The empirically blended splatter map uses photon discs rendered to a texture. The quality tests are shown on Figure 6.9 for scenes 1 to 6. This method produces nice results in some cases, but cannot handle edges and bumpy surfaces as seen, since the discs will stick out and also through objects. The method produces nice looking caustics for open planar surfaces, which means that it is not a very general method. The of the splatter method was measured using pixel area and quads. And was measured at a distance and close up. The results are shown on Table 6.13. The method, with it is current implementation is not able

# Scene	# Photons	FPS	Duration	FPS w OpenRT
2 (near)	5000	11.36	88.28	11.11
2 (far)	5000	11.72	85.32	11.63
3 (near)	5000	14.25	70.18	13.92
3 (far)	5000	28.88	34.63	27.55
4 (near)	5000	14.44	69.25	14.44
4 (far)	5000	17.74	56.37	17.23
5 (near)	5000	22.51	44.42	21.70
5 (far)	5000	34.04	29.38	32.21
6 (near)	5000	11.73	85.25	11.50
6 (far)	5000	14.91	67.07	14.63

Table 6.13: The rendering times of running the application with the Empirical Splatter method

to what is considered real-time frame rates, interactive frame rates are realistic.

6.2.6 Level adjusted Mip-Maps

The last method considered gives the quality tests shown on Figure 6.10. The quality is unacceptable and not an improvement. There seems to be unresolved issues with using a mip level that is unique to each pixel. during develop this was also seen using a splatter texture to give values for blending. We attempted

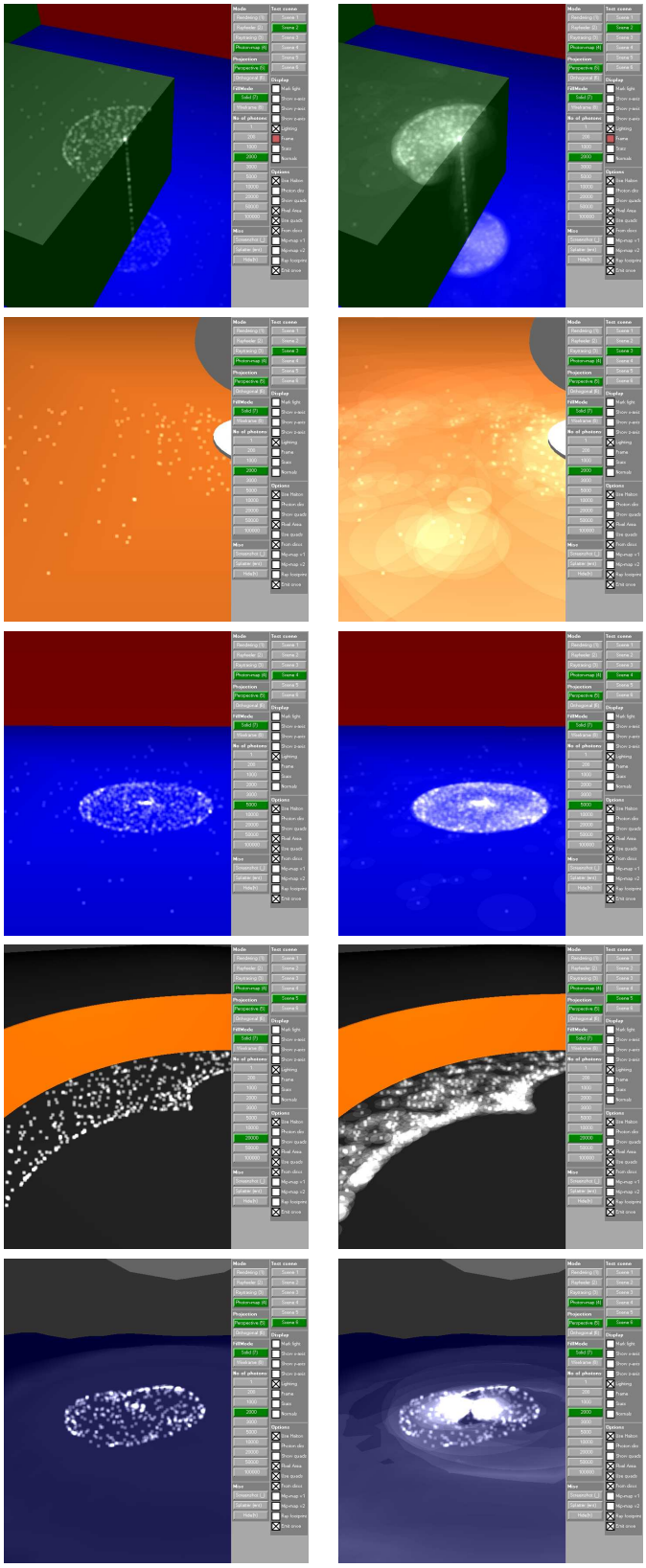

Figure 6.9: The quality test for empirical splatter

to use loops to get a finer resolution for the trilinear sampling, but the results are not satisfying. This however is far from the expected result and the author has of yet not been able to explain the appearance. So it is possible that there are flaws in the implementation.

6.2.7 Various observations

Here some general observations will be reviewed.

Filtering method

The different filter types described in Section 4.4.1 give different results. A caustic from a sphere filtered using the different filters is shown on Figure 6.11. The square filter produces noticeably blocky images at almost any distance. Bent's filter produces images similar to the new filter, except as one zooms in the filter shape changes towards a square filter. The author, whose opinion might be colored, prefers the image produced with the new filter. It utilizes the entire filter kernel and produces nice rounded filtered photons.

Critical Angles of Ray Differentials

Using ray differentials leads to an issue with critical angles for ray differentials, when the photon path does not surpass the critical angle. An image of the photon paths of a ray traced scene is shown on Figure 6.12. Figure 6.12 shows that in some scenes it can be quite a lot of photons, whose ray differentials exceed the critical angle. Of course this only happens in scenes with refractive objects.

Texture coordinates for Mip-map sampling

There seem to be unresolved issues with the filter coordinates as can be seen on 6.8. The photon is not positioned at the center of the filtered square. We do adjust the screen coordinate, before using it for the loop-up. The adjustment is needed since pixels are defined by a point in the center of the pixel, which is not the case with texels. The adjustment is given by [25] as :

$$t(u, v) = c_s(x, y) - (0.5, 0.5) \tag{6.5}$$

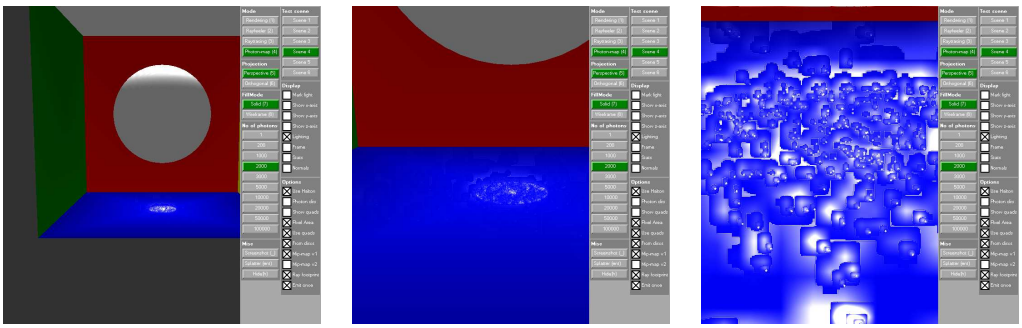

Figure 6.10: The quality test for empirical splatter

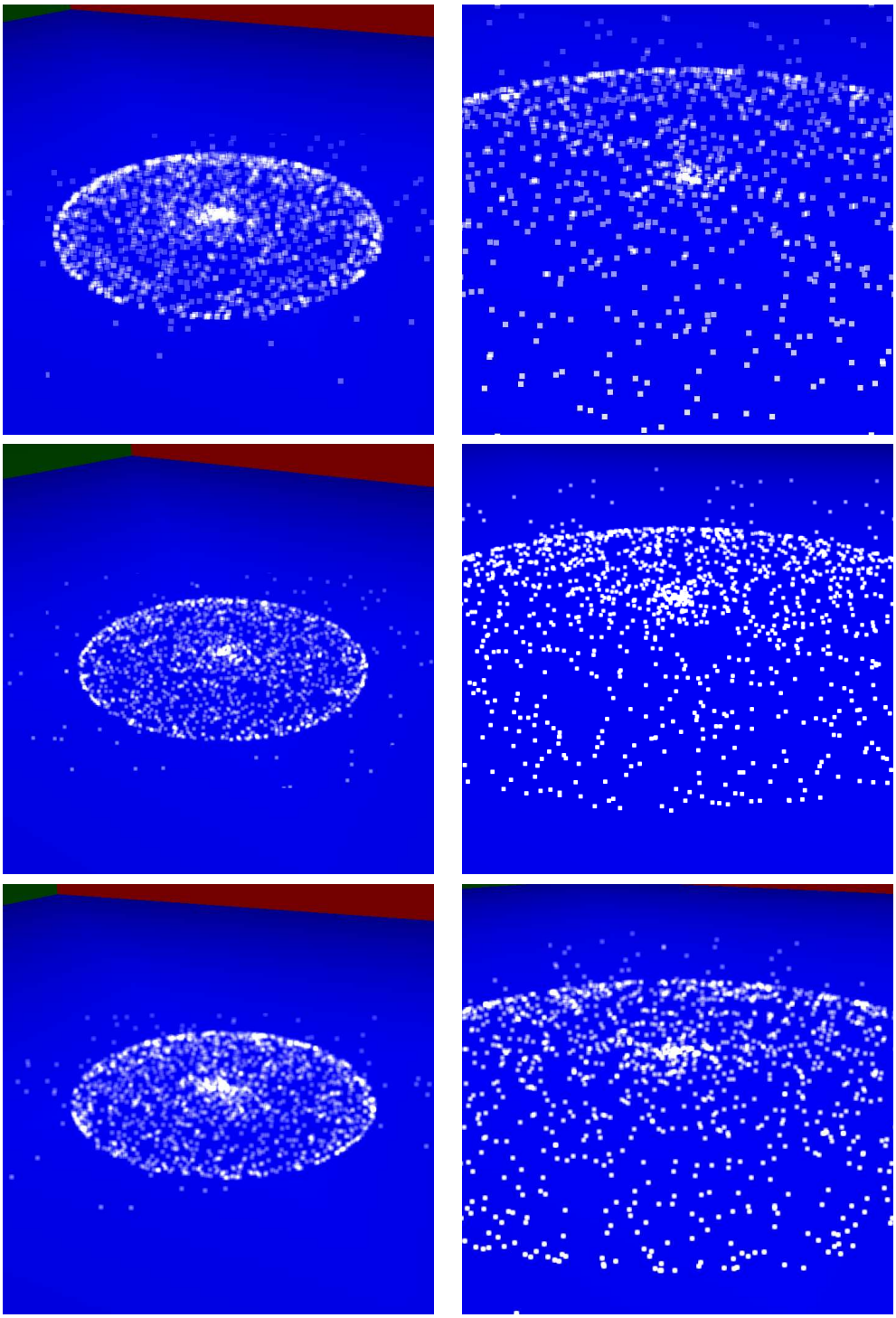

Figure 6.11: The different filter types. Top row are the square filtered images, the center row is Bent's filter and the bottom row is the filter presented in this thesis.

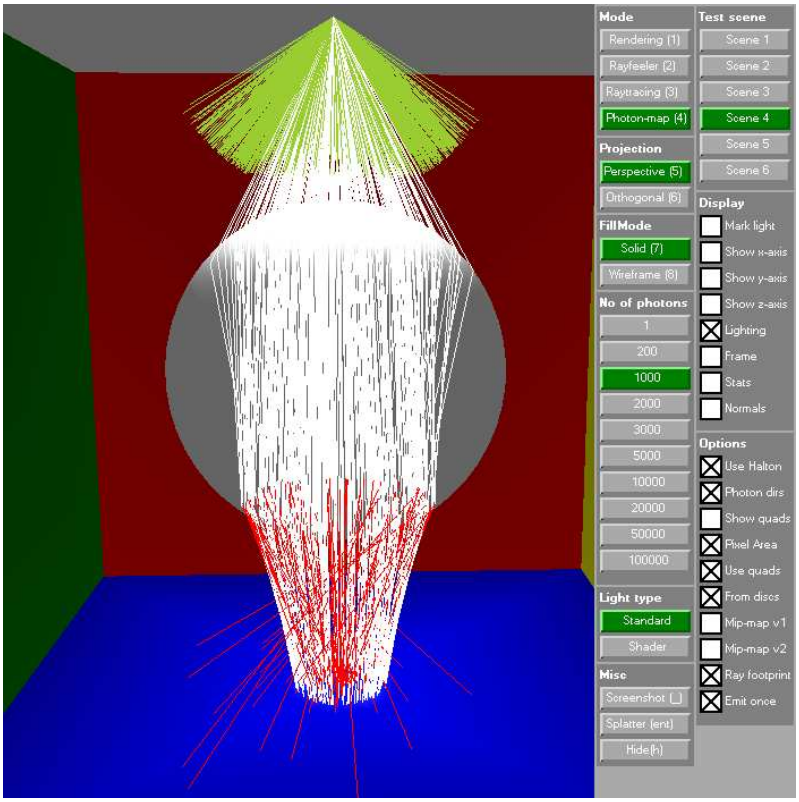

Figure 6.12: This is test scene 4, where the green ray are the initial directions, the white paths are the photons paths of the stored photons and the red rays are the photon paths of stored photons, where the ray differential exceeded a critical angle at some point.

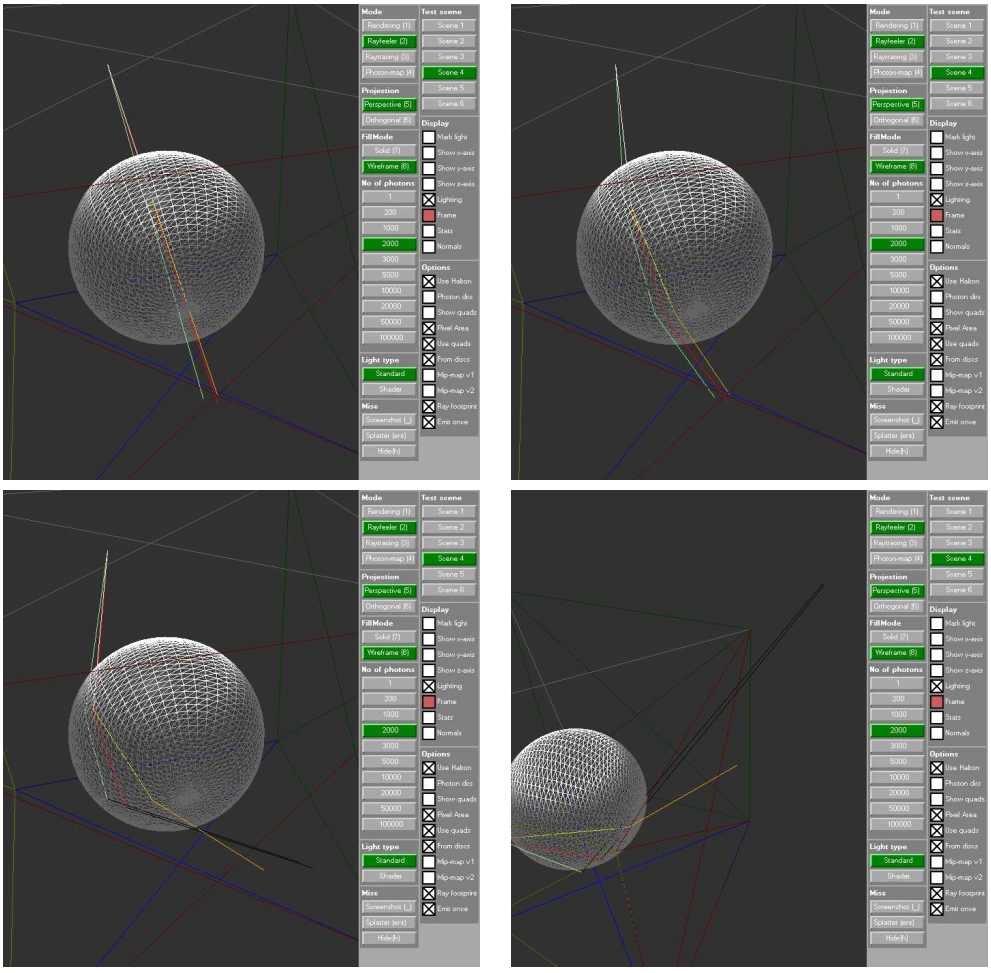

Figure 6.13: The path that changes color is the photon path, the red and green paths are ray differential paths and the black path segments are used to indicate that this ray differential has exceeded a critical angle.

Chapter 7

Discussion

7.1 Conclusion

Three methods have been presented for adding coherency to a caustic, a new filter kernel has been presented as well as a new method for improving the generality of the quad optimization.

The filter kernel presented creates a softer and more round appearance. It also utilizes all the samples in a kernel.

The improvement to the generation of quads is necessary. Normally the quads would cover the filtered area, but when we add coherency the quads generated using photons are no longer sufficient as seen en the results section. The method using quads improves on the quality of the method, but it is rather slow in its current implementation.

The quality of the caustics generated using the empirical methods are crude and due to the blending method the caustic saturates the image, thus appearing more powerful. There also exist a need for adjusting the lookup since the center of the filter used for sub sampling the mip-map is not actually place in a center pixel. The center on the test machine is instead positioned in the topleft pixel of a square kernel. However the method handles arbitrary surfaces like the standard filtering method and is also cheap.

A good implementation of the Level adjusted mip-map level for simulating dynamic radius in the filter kernel was not reached and the caustic is ruined by the current implementation. The idea was included in this thesis because it could possibly be developed further.

Mip-maps are crude and alone would probably not give nice looking caustic. They are however generated fast and offer support for arbitrary surfaces. Therefore the idea should not be abandoned completely. If combined perhaps with ray differentials or a combination of filtering and blending it may become useful.

The last method used a splattered texture and produced nice looking caus-

tics if emitted onto planar open surfaces. There is however issues concerning speed, this could perhaps be remedied with a better implementation. A sprite implementation of photon discs was tried, but was even slower than the polygon based discs, the reason for this could be inexperience and the idea should not be abandoned. Arbitrary surfaces are not handled well by this method either.

The estimated speeds of the mip-map method was very good and combined with a fast ray tracer such as OpenRT, fast real-time caustics are not out of reach.

Managed DirectX is very new and may improve, which could affect the speed of the implementation. Is noted that normally OpenGL or Unmanaged DirectX with C++ is used. The author had only had experience with OpenGL at the beginning of the project and better implementations could also affect the speed in a significant way. In particular parts with many draw calls, such as photons or more importantly discs, there could be much to gain from reducing the number of draw calls, buffer locks and more according to the Microsoft SDK Documentation. The author however did not become aware of this in time to effectively put the knowledge to use.

7.2 Future work

Future work could use a ray tracer and as computers become more powerful the fast ray tracers will be more useful, for generating effects. The ray tracer can handle surfaces and geometry more physically accurate than the rasterized techniques at the moment.

The splatter method presented would have to be sped up. Drawing photon discs would have to be optimized. A fast implementation might use a square (defined with 4 vertices) and a texture containing a circle to draw fast photon discs. However there are many drawbacks to this method that would have to be solved and it seems unlikely that it will succeed.
Also the splatter method currently overlaps objects, this might be render for occlusion as was done when drawing photons to a texture. This is done by simply drawing the black scene before, drawing the discs. However this would add further cost to the method.

Using ray tracers would likely include the use of a fast filtering method, but a method would need to be more robust than the methods presented here in their current states. Mip-maps promises fast generation of data levels and conserves the values of the texture to a certain level. Fast mip-level generation for floating point textures would enable the method to handle more photons, this is however a hardware problem. A better blending method might be developed using ray differentials.
Also for mip-maps the texture coordinate look up issue should be examined.

Finally more research is likely to be focused on Caustic Maps, mentioned in related work, which is promising method, that however is less accurate when distributing photons. If the accuracy is increased this method might become dominant.

Appendix A

Ray tracing and photon-map optimizations

Here we will give a overview of some of the possible methods for accelerating ray tracing in regards to photon emission and photon mapping that was considered during the planning of this thesis. An overview of many more methods can be found in [7], this encompasses methods that are not directed at caustics, but ray tracing in general.

A.1 Intersection

Intersection tests are where much of the calculation in a ray tracer takes place. Making sure that these are optimized is therefore very important and several optimizations can that complement each other.

Primitive intersections

At the base level is the intersection between primitives, which could be parameterized or meshed depending on implementation. SIMD implementations would probably be used, to take advantage of modern computers multi threading capabilities. Some papers on the subject primitive intersection testing are [26, 27, 28, 29, 30, 31].

Bounding volumes

Bounding volumes where introduced in [32] and applied in [2]. A bounding volume is used to encapsulate complex objects with a simpler shell. Intersection testing with the shell is faster, because of the lower complexity and can therefor be used to quickly eliminate objects that a ray path does not intersect with in a complex scene. Common bounding volumes are a box or a sphere. A box is faster to test intersections against, but a sphere might give a more accurate intersection test by enclosing the encapsulated object tighter.

Spatial subdivision

With spatial partitioning the object is divided in to voxels containing pointers to faces they contain. For rigid objects a BSP-tree containing the information needed is generated once. Walking through a tree to the voxel containing faces and intersecting the contained faces is likely faster than checking intersections against all the faces of the object. Itt's important to decide on how fine a division is used is important, since too many voxels will increase the cost of walking the tree.

A.2 Initial rays

If one can find methods to cheaply reduce the number of initial rays, it could greatly reduce the number of intersections that needs to handed.

Projection maps

A simple method for determining in what directions shooting rays would be useful is using projection maps as is done in [17]. Projection maps are 6 images containing the scene rendered in each direction from the light source. Only caustics generators would be rendered and when emitting one can check whether or not to emit a photon with a lookup in the projection map using the direction. Whether this is an optimization depends on the scene. In a scene with complex geometry and movement of the generators, the cost of rendering for the projection maps might be too great. We use scenes that are sparsely populated and ray's are only checked for intersection a few times, so elimination is quick as is.

Simple sample reuse

In [17] a simple method for reducing the number of photons emitted is suggested and tested. This involves elimination of a percentage of the photons emitted last frame and shooting the same amounts of new photons. This was not tested in animated scenes, but is assumed to work in relatively static environments. In static environment the authors found it to be acceptable. The author of this thesis imagines that it would reduce flickering in the caustic. Since this thesis is not going for an

Sample reuse

For ordinary ray tracing of dynamic scenes an optimization has been suggested by Formella in [11], in which the entire path of a ray is saved. Thus when this ray is intersected by moving objects it is retraced. It is possible that this could be adapted for photons, although it would require extra memory storage.

A.3 Traversal

Eliminating a ray, that does not contribute to the final image (here under caustic), can possibly save many intersection calculations. Ray tracing a scene could lead to explosive growth of the ray tree, with photon emission each intersection

leads to a maximum of 1 emitted ray, so there is less gain, but it still should be noticeable.

Spatial and Hierarchical subdivision of scenes

The scene, much like the objects themselves, can be segmented into voxels using splitting planes. This would be saved in a tree structure that, if balanced, could save a lot of intersection tests. This is the case since once a ray is know to travel on one side of the plane, things on the other side can be eliminated. There are several different tree types and many of these where included in a test by Havran et al. in [33]. BSP-tress and Kd-trees are generally considered the best options. These methods where originally developed for static scenes where the trees can be build on startup thus not incurring a penalty of building during rendering. However for dynamic scenes the tree would have to be updated, which could mean that it is not a speed-up. An issue that can occur with scene divided by a large number of small voxels is that an object can be contained by several voxels and thus need several intersection tests against. Mail boxing was suggested by Amanatides et al. in [34] for use with ray tracing. In mail boxing every object is given an ID and the ray stores the last ID it was tested for intersection with. A concern might be the added memory usage.

Appendix B

Math

B.1 Taylor approximation

A Taylor approximation can be used to approximate a complex function, $f(x)$, with a sum of polynomial functions. This requires that the $f(x)$ is n times differentiable. The precision of the approximation is dependant on the order, n, of the approximation given by

$$P_n(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \ldots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \tag{B.1}$$

the higher the order, the better the approximation is.

B.2 Spherical polar coordinates

Spherical polar coordinates can be used to describe a vector in in cartesian coordinates by two angles, φ and Θ. We can use this to generate a vector from two random numbers. The positions are given by:

$$x = rsin\Theta cos\varphi \tag{B.2}$$
$$y = rsin\Theta sin\varphi \tag{B.3}$$
$$z = rcos\Theta \tag{B.4}$$

where $r \in [0, \infty]$ is the vector length, and $\varphi \in [0, 2\pi]$ and $\Theta \in [0, \pi]$. For normalized vectors $r = 1$.

B.3 Quaternions

Rotation is a common operation in computer graphics and a natural way to think of rotation is by three angles ie. one for a rotation around each axis. This is not perfect however. The rotations in that notation will be dependant on each other (which is hinted at by the non-commutative nature of rotation matrices) and can produce problems such as Gimble lock ([18]).
Quaternions are an extension of complex numbers (introduced by Sir William

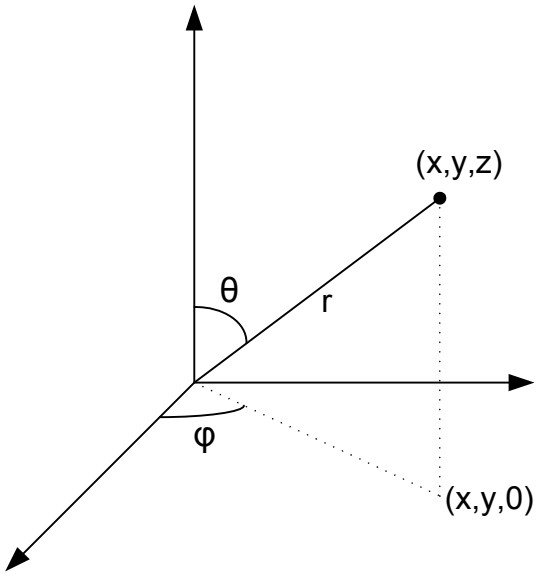

Figure B.1: Illustration of polar coordinates.

Hamilton in the 1830s) that has found application in computer graphics as a representation for rotations. A quaternion, q, can be written as such :

$$q = a + bi + cj + dk \tag{B.5}$$

with a being the real part and i, j, k being complex. For our purposes we use the notation :

$$q = (s, v) = s + v_x i + v_y j + v_z k \tag{B.6}$$

where s is a scalar and v is a vector on axes i, j, k. The conjugate of q is $\overline{q}$:

$$\overline{q} = (s, -v) \tag{B.7}$$

Assuming unit vector, n, we can write the q on the form :

$$q = (cos(\Theta/2), sin(\Theta/2)n) \tag{B.8}$$

where $|n| = 1$. A point p can be written on the form :

$$p = (0, r) \tag{B.9}$$

with $r(x, y, z)$ being the vector to the point. Rotating p is achieved by the operation :

$$p_{rotated} = qpq^{-1} \tag{B.10}$$

which if the p is a unit vector n is :

$$n_{rotated} = qn\overline{q} \tag{B.11}$$

The general rotation operation can be written on matrix form :

$$\begin{bmatrix} 1-2{v_y}^2-2{v_z}^2 & 2v_xv_y-2sv_z & 2v_xv_z-2sv_y & 0 \\ 2v_xv_y+2sv_z & 1-2v_x^2-2v_z^2 & 2v_yv_z-2sv_x & 0 \\ 2v_xv_z-2sv_y & 2v_yv_z-2sv_x & 1-2v_x^2-2v_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.12}$$

using the general notation for q.

Bibliography

Thesis

[1] Musawir Shah and Sumatra Pattanaik. *Caustics Mapping : An Image-space technique for Real-time Caustics*, 2005.

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

[3] Bent Dahlgaard Larsen *Real-time Global Illumination by Simulating Photon Mapping*, 2004.

[4] Henrik Wann Jensen *Realistic Image Synthesis Using Photon Mapping*, 2001.

[5] Henrik Wann Jensen *Global illumination using photon maps*, 1996.

[6] M. Wand, et al. *Real-Time Caustics*, 2003.

[7] Ingo Wald, *Realtime Ray Tracing and Interactive Global Illumination*, 2004.

[8] Ingo Wald, et al. *Realtime Caustics Using Distributed Photon Mapping*, 2004.

[9] James Arvo. *Backward ray tracing*, 1986.

[10] Andrew Glassner, *An introduction to ray tracing*, 1989.

[11] Arno Formella, Christian Gill and V. Hofmeyer. *Fast Ray Tracing of Sequences by Ray History Evaluation*, 1994.

[12] Kavita Bala, Julie Dorsey and Seth Teller. *Radiance Interpolants and Accelerated Bound-Error Ray Tracing*, 1999.

[13] Andrew Woo. *Fast Ray-Polygon Intersection*, 1990.

[14] Ingo Wald, Carsten Benthim, Markus Wagner and Philipp Slusallek. *Interactive Rendering with Coherent Ray Tracing*, 2001.

[15] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg and T. Limperis. *Geometric considerations and nomenclature for reflectance.* Monograph 161, National Bureau of Standards (US), 1977.

[16] James T. Kajiya. *The rendering equation*, Computer Graphics (Proc. SIGGRAPH '86), 1986.

[17] Martin Valvik and Peter Jensen. *Real-time Caustics*, 2004.

[18] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques*,Addison-Wesley,ISBN:0-201-54412-1

[19] L. Williams. *Pyramidal Graphics*, 17(3), 1-11, (Proc. SIGGRAPH '83).

[20] W. Newman and R. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, 1973.

[21] Homan Igehy, *Tracing ray differentials*, SIGGRAPH '99 Proceedings, 1999.

[22] Christophe Schlick, *A customizable reflectance model for everyday rendering*, Fourth Eurographics Workshop on Rendering, 1993.

[23] Timothy J. Purcell et al. *Photon Mapping on Programmable Graphics Hardware*, SIGGRAPH/EUROGRAPHICS, 2003.

[24] Randima Fernando et al.. *The Cg Tutorial*, nVIDIA press, Addison Wesley.

[25] Microsoft Corp., *Directly Mapping Texels to Pixels*, $http://msdn.microsoft.com/library/en-us/directx9_c/DirectlyMappingTexelstoPixels.asp?frame=true$.

Appendices

---

[26] Didier Badouel. *An Efficient Ray Polygon Intersection*, 1992.

[27] Jeff Erickson. *Pluecker Coordinates*, 1997.

[28] Tomas Möller and Ben Trumbore. *Fast, minimum storage ray triangle intersection*, 1997.

[29] Ken Shoemake. *Pluecker Coordinate Tutorial*, 1998.

[30] Andrew Woo. *Fast Ray-Polygon Intersection*, 1990.

[31] Olivier Devillers and Philippe Guigue. *Faster Triangle-Triangle Intersection Tests*, 2002.

[32] H.J.Clark, *Hierarchical Geometric Models for Vsiible Surface Algorithms*, Comm. ACM, 1976.

[33] Vlastinil Havran, Jan Prikryl and Werner Purgathofer. *Statistical Comparison of Ray-Shooting Efficiency Schemes*, Techical Report, 2000.

[34] John Amanatides and Andrew Woo. *A Fast Voxel Transversal Algorithm for Ray Tracing*, EUROPGRAHICS 87, 1987.