Multi-sampled Photon Differentials

Incorporating the View Ray Differential in the Radiance Estimate

Lasse Jon Fuglsang Pedersen

(fuglsang@diku.dk)

Supervisor: Jon Sporring

(sporring@diku.dk)

October 2011

Abstract

Using the fundamental theory of photon mapping, photon differentials, and ray differentials, I explore the possibility of incorporating the ray differential of a view ray in the radiance estimate for photon mapping with photon differentials.

By treating the view ray differential as a beam that occupies the width and height of single pixel in the final image, I arrive at two different methods: *Coplanar intersection-weighted photon differentials*, which is based on explicitly computing the coplanar intersection between the footprints of the respective ray differentials, using the coverage and centroid of the intersection to weigh the contribution of the photon differential, and *multi-sampled photon differentials*, which uses the footprint of the view ray differential to define a set of sampling points, using this set of points to multi-sample the photon differential.

Based on the hypothesis that it is possible to surpass the accuracy of the regular filtered radiance estimate by taking into account the footprint of the view ray differential, I subject each of the two new methods to a common test case, comparing them to two configurations of the regular filtered radiance estimate; one using a single view ray per pixel, and one using 3×3 view rays per pixel.

My evaluation shows that, while both new methods are improvements over the regular filtered radiance estimate with one view ray per pixel in terms of accuracy, coplanar intersection-weighed photon differentials does not perform well enough in terms of rendering time to be useful in practice without further optimization.

Multi-sampled photon differentials, however, presents a practical alternative to obtaining more accurate results without tracing more view rays per pixel. With 8×8 sampling points, the results of the method clearly rival those of the regular filtered radiance estimate with 3×3 view rays per pixel, and in addition it is also three times faster.

Contents

1 Introduction 1

2 Photon mapping 2
2.1 Photon tracing . . . 2
2.1.1 Emission . . . 2
2.1.2 Scattering . . . 3
2.2 Rendering . . . 5
2.2.1 Computing the reflected radiance . . . 6
2.2.2 Hybrid approaches . . . 8
2.3 Data structure . . . 8
2.3.1 Kd-trees . . . 8
2.3.2 Octrees . . . 9
2.4 Summary . . . 10

3 Photon differentials 11
3.1 Ray differentials . . . 11
3.1.1 Initialization . . . 13
3.1.2 Transfer . . . 14
3.1.3 Reflection . . . 15
3.1.4 Refraction . . . 16
3.2 Emission and scattering . . . 18
3.2.1 Footprint of photon differential in Q . . . 19
3.3 Computing the reflected radiance . . . 20

4 Incorporating the view ray differential 22
4.1 Initialization of view ray differential . . . 23
4.2 Coplanar intersection-weighted photon differentials . . . 25
4.2.1 Projection and rotation . . . 25
4.2.2 Computing the intersection . . . 27
4.2.3 Coverage and centroid . . . 29
4.2.4 Computing the reflected radiance . . . 30
4.3 Multi-sampled photon differentials . . . 30
4.3.1 Generating the sampling points . . . 31
4.3.2 Computing the reflected radiance . . . 32

5 Implementation and evaluation 33
5.1 Image comparisons . . . 34
5.2 Performance discussion . . . 40

6 Conclusion 42

References 43

A Source code 44

1 Introduction

In synthethic computer graphics, there are generally two overarching branches of rendering algorithms, where each branch caters to a specific type of application. One of these branches is concerned specifically with real-time applications, which require that images are rendered at interactive framerates, while the other branch is more concerned about photorealism, which requires that the light in the scene is simulated as accurately as possible.

In terms of photorealistic rendering, most rendering algorithms made for this purpose revolve around the concept of approximating the solution to an equation known as *the rendering equation*. The rendering equation was introduced in 1986 by Kayija [6]. Without going into the formal description this early in the text, let me just state that the rendering equation defines the visible light in some point on a surface as a function of the viewpoint, the surface itself, and the surfaces that are visible from that particular point.

Algorithms that solve the rendering equation are also commonly referred to as *global illumination algorithms*. This is because the rendering equation treats each surface as a potential source of light, which implies that surfaces can be illuminated both indirectly by other surfaces, and directly by dedicated sources.

The topic of this report has its roots in a global illumination algorithm known as *photon mapping*. Photon mapping is a two-stage algorithm, where the first stage involves tracing *photons* from the light sources, storing these at each intersecting surface in an intermediate data structure. The second stage involves tracing rays from the viewpoint, sampling the intermediate data structure in each point of intersection to reconstruct the light reflected towards the viewpoint. This reconstruction is referred to as the *radiance estimate*.

More recently, Schjøth et al. [8] proposed an extension to photon mapping known as *photon differentials*. With photon differentials, each photon is associated with a *ray differential*, which keeps track of the size and shape of the light represented by the photon. Using the extra information provided by photon differentials, Schjøth et al. [8] define a new radiance estimate, which proves to be better at preserving the shape and contours of the reconstructed light.

Based on the observation that also view rays can be associated with ray differentials, I explore how to incorporate the *view ray differential* in the radiance estimate for photon mapping with photon differentials, with the purpose of improving the accuracy of the radiance estimate. Specifically, using the radiance estimate defined by Schjøth et al. [8] as the foundation for my work, the purpose is to arrive at a new radiance estimate that is more accurate than the radiance estimate defined by Schjøth et al. [8].

Prior to incorporating the view ray differential in the radiance estimate, I describe the concepts and theory of photon mapping, photon differentials, and ray differentials, all of which are topics that contain terms, principles and definitions that are fundamental to my work.

This is a master's thesis at DIKU – the Department of Computer Science at the University of Copenhagen. In this regard, I would like to thank to my supervisor Jon Sporring for the advice that I have received during the project.

2 Photon mapping

Photon mapping is a global illumination algorithm that solves the rendering equation. It revolves around the concept of using an intermediate data structure, the so-called *photon map*, to estimate the irradiance for any given point on any given surface, and consequently also the reflected radiance for any given viewpoint. It was introduced by Henrik Wann Jensen in 1996 [4].

The basic photon mapping algorithm consists of two stages: The *photon tracing* stage and the *rendering* stage. The photon tracing stage populates the photon map by tracing rays from the light sources, while the rendering stage generates the final image by tracing rays from the viewpoint, sampling the photon map.

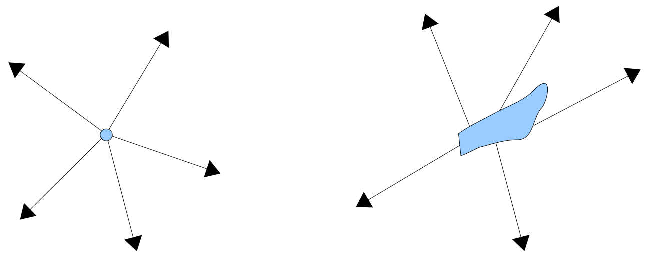
Sections 2.1 and 2.2 describe each of the two stages in more detail, while section 2.3 discusses the underlying data structure for the photon map.

2.1 Photon tracing

In the photon tracing stage, a finite number of *photons* are emitted from the light sources in the scene. Each photon can be thought of as a particle that represents some radiant power Φ_p travelling in a specific direction. The scattering and distribution of light through the scene is then simulated by tracing the path of each photon, and along each path the relevant photon-surface intersections are stored in the photon map.

2.1.1 Emission

As shown in Figure 1, the initial position and direction of each photon depends on the type of light source that it originates from. If the light is a point light, then the starting point is already known, and the direction is randomly chosen within the unit sphere. For complex lights, i.e. lights with a surface, the starting point is a randomly chosen point on the surface, and the direction is randomly chosen within the unit hemisphere in the direction of the corresponding surface normal.



(a) Point light (b) Complex light

Figure 1: Illustration of photons being emitted from two different types of light sources. When emitted by a point light, the emitted photon all assume the same starting point. When emitted by a complex light, the starting point is a random point on the emitting surface. In this example the emission occurs in 2D, meaning that the surface in (b) is the boundary of the blue area.

The radiant power of each photon is initialized to a fraction of the corresponding light source's radiant power Φ. To avoid adding or removing energy from the system, the sum of the radiant power of the photons emitted by a given light source must always equal the radiant power of the light source itself. A simple solution is to divide the radiant power of the light source evenly among its emitted photons, i.e.:

$$\Phi_p = \frac{\Phi}{N} \tag{1}$$

where N is the number of emitted photons for that particular light source.

Note that the number of emitted photons is not a direct parameter of the individual light sources. Rather, this value is inferred from the specified number of photons for the entire scene N_{scene}, as well as each light source's radiant power Φ, in relation to the total radiant power of all light sources in the scene Φ_{scene}. Thus, the number of emitted photons for a single light source is defined as:

$$N = \frac{\Phi}{\Phi_{\text{scene}}} N_{\text{scene}} \tag{2}$$

Jensen remarks [5, p.61] that all photons having approximately equal radiant power is important for the quality of the radiance estimate. The radiance estimate is described in section 2.2.1 on page 6.

For photons emitted using the model described above, this is easily verified by substituting the right hand side of Equation 2 into Equation 1, and then reducing:

$$\begin{aligned} \Phi_p &= \frac{\Phi}{\left(\frac{\Phi}{\Phi_{\text{scene}}} N_{\text{scene}}\right)} \\ &= \frac{\Phi_{\text{scene}}}{N_{\text{scene}}} \end{aligned} \tag{3}$$

which states that the radiant power of any emitted photon is just a function of the total radiant power of all lights in the scene, as well as the total number of emitted photons.

2.1.2 Scattering

After a photon has been emitted, then it is traced through the scene to simulate the scattering of light. This is achieved using conventional ray tracing techniques. Each photon is essentially treated as a ray that is tested for intersections with surfaces in the scene, where, for each photon-surface intersection, the material properties of the surface are used to decide the next course of action.

Based on the material properties of an intersecting surface, a photon can be reflected, refracted, or absorbed in some capacity. In general, when a photon is reflected or refracted, then its radiant power Φ_p and incident direction ω_p is stored in the photon map, essentially adding to the incident radiant power in the point of intersection. As shown in Figure 2 on page 4, a photon can be stored multiple times along a path. A photon trace continues until the photon is absorbed, until some maximum number of intersections is reached, or until no further intersections can be found.

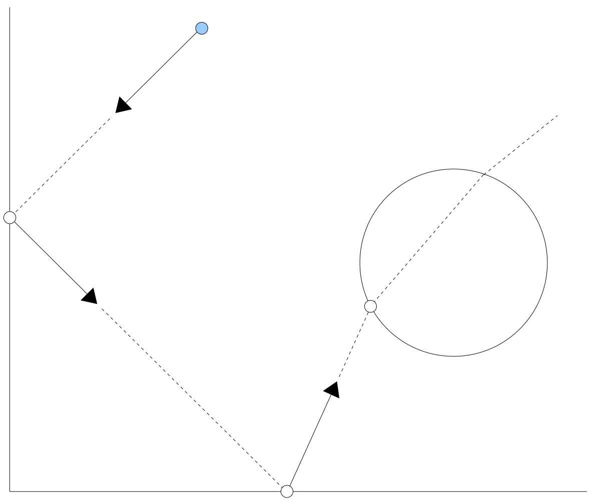


Figure 2: An example of one of the possible paths that a photon can take through a scene. The blue dot denotes the emitting light source, a point light in this case, and the white dots denote where the photon is stored in the photon map for this particular path. In this case the photon is reflected twice, first by a specular surface and then by a diffuse surface, and then refracted. The trace terminates as the photon exits the scene in a direction where no more intersections can be found.

Obviously, most surfaces are not perfect reflectors nor fully transparent. They absorb some of the light that hits them, and they typically also reflect light in multiple directions, diffuse as well as specular. For these reasons, an explicit implementation of scattering normally requires that the radiant power of each photon is adjusted when reflected or refracted, and that more photons are created at each photon-surface intersection, in order to evaluate all possible paths.

However, evaluating all possible paths for each emitted photon is also expensive and impractical. For this reason, photon tracing does not implement scattering explicitly, but instead uses a probabilistic selection scheme known as *russian roulette*.

Using the russian roulette scheme, only a single path is traced for each photon, and the radiant power of each photon is retained for the entire length of its path, regardless of the number of reflections and refractions. Instead, the number of reflections and refractions that each photon is subjected to is probabilistically chosen based on the surface properties along the path.

More specifically, the russian roulette scheme lines up the possible choices as numeric intervals proportional to the material properties of the intersecting surface, and then picks a random number that points to one of them. For example, if a surface is 75% specular, then the russian roulette scheme will cause 75% of the photons hitting that surface to be reflected in the specular direction with unchanged radiant power, while the remaining 25% will be completely absorbed and discarded.

Given enough photons hitting the same surface, it should be clear that reflecting

100% of the radiant power of 75% of the incoming photons is equivalent to reflecting 75% of the radiant power of 100% of the incoming photons. Russian roulette therefore produces a result that is similar to that of a non-probabilistic approach, although at the cost of some added variance in the photon map.

Listing 1 shows how photon scattering can be implemented statistically using the russian roulette scheme, eliminating the need to trace all possible paths for each photon.

```
// photon scattering using russian roulette
void trace(photon)
{
    surf = find_next_surface(photon)
    rand = rand_unit()

    // test if within interval for specular reflection
    if ((rand -= surf.k_s) < 0)
    {
        store_photon(photon, surf)
        return trace(reflect_s(photon, surf))
    }

    // test if within interval for diffuse reflection
    if ((rand -= surf.k_d) < 0)
    {
        store_photon(photon, surf)
        return trace(reflect_d(photon, surf))
    }

    // test if within interval for refraction
    if ((rand -= surf.k_t) < 0)
    {
        store_photon(photon, surf)
        return trace(refract(photon, surf))
    }

    // absorb (end trace)
}
```

Listing 1: Pseudo-code for photon scattering using russian roulette.

2.2 Rendering

During the rendering stage, a finite number of rays are cast from the viewpoint and traced into the scene. These rays pass through points on the view plane that correspond to pixels in the final image, hence they are referred to as *view rays*.

Each view ray is traced recursively, but the recursion only occurs in the case of refraction. This is to support transparent surfaces, and this particular part of the algorithm can be likened to a simplified version of classic recursive ray tracing [11].

Because the photon map contains an approximation of the incident light for every point on every surface in the scene, direct as well as indirect, it is not necessary to pursue reflection rays, or trace rays back to direct light sources, in order to compute the reflected radiance for a given view ray.

For each ray-surface intersection, the reflected radiance in the direction of the view ray is computed directly based on the k-nearest photons in the photon map. Jensen refers to this computation as the *radiance estimate*. Obviously, the closer the k-nearest photons are to the point of intersection, the more accurate the estimate of the reflected radiance will be.

Figure 3 illustrates the paths of example view rays as they are traced into an example scene, as well as how the distance may vary to the photons that contribute to the radiance estimate in each ray-surface intersection, assuming $k = 3$.

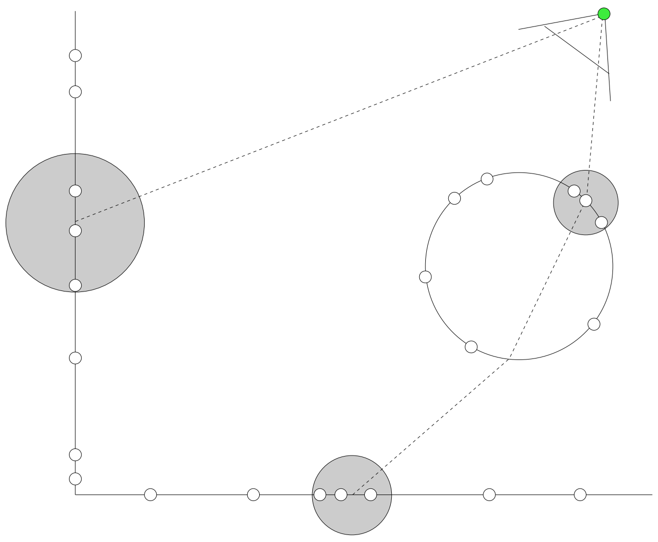


Figure 3: View rays are traced from the viewpoint and into the scene. For each ray-surface intersection, the k-nearest photons are retrieved from the photon map and used as input to the radiance estimate, which is detailed in section 2.2.1. In this example $k = 3$. Notice how the support region (marked with grey circles) expands around the point of intersection to encompass the k-nearest photons. Obviously, the smaller the support region is, the more accurate the radiance estimate will be. In general, emitting more photons into the scene during the photon tracing stage will increase the chance of a smaller support region during the rendering stage.

The radiance estimate is described in more detail in the the following section, 2.2.1, and section 2.2.2 discusses some of the optimizations that are relevant in terms of practical application of the basic photon mapping algorithm.

2.2.1 Computing the reflected radiance

Computing the radiance for a given view ray is synonymous with solving the rendering equation:

$$L_0(x, \omega) = L_e(x, \omega) + L_r(x, \omega) \tag{4}$$

where x is the point of intersection, ω is the outgoing direction, L_e is the emitted radiance, and L_r is the reflected radiance. Typically, ω is defined as the normalized and inverted direction of the view ray's actual direction v, i.e.:

$$\omega = -\frac{v}{\|v\|} \tag{5}$$

Assuming that the emitted radiance can be read directly from the intersecting surface in x, the main point of interest is to compute the reflected radiance. The reflected radiance can be described by the following integral:

$$L_r(x, \omega) = \int_{\Omega_x} f_r(x, \omega', \omega) L_i(x, \omega')(n_x \cdot \omega') d\omega' \tag{6}$$

where Ω_x is the hemisphere in the direction of the surface normal n_x in the point of intersection x. The integration variable ω' is a direction on the hemisphere, and L_i is the incident radiance from ω'. Finally, f_r is the bi-directional reflectance distribution function (BRDF), which describes how much of the incident light from ω' is reflected in the outgoing direction ω.

As described in section 2.1.2, each photon stored in the photon map represents some incident radiant power Φ_p from a specific direction ω_p. According to Jensen [5, p.14], incident radiance L_i can be expressed in terms of incident radiant power Φ_i as follows:

$$L_i(x, \omega') = \frac{d^2\Phi_i(x, \omega')}{(n_x \cdot \omega')d\omega' dA_i} \tag{7}$$

where $d\omega'$ is the solid angle and dA_i is some infinitesimal area on the surface around x. Plugging this back into Equation 6 yields:

$$\begin{aligned} L_r(x, \omega) &= \int_{\Omega_x} f_r(x, \omega', \omega) \frac{d^2\Phi_i(x, \omega')}{(n_x \cdot \omega')d\omega' dA_i}(n_x \cdot \omega')d\omega' \\ &= \int_{\Omega_x} f_r(x, \omega', \omega) \frac{d^2\Phi_i(x, \omega')}{dA_i} \end{aligned} \tag{8}$$

Finally, the continuous integral is approximated by a sum over the k-nearest photons in the photon map. Jensen writes:

$$L_r(x, \omega) \approx \sum_{p=1}^{k} f_r(x, \omega_p, \omega) \frac{\Delta\Phi_p(x, \omega_p)}{\Delta A} \tag{9}$$

where ω_p is the incident direction stored with each photon, and $\Delta\Phi_p(x, \omega_p)$ is the stored radiant power, assuming that the photon was actually stored in x. This is usually *not* the case, and the farther a photon lies from x, the less accurate its contribution to the radiance estimate will be.

In Equation 9, ΔA denotes the projected area of a sphere expanded around x to encompass the k-nearest photons. This is also illustrated in Figure 3 on page 6. Assuming a flat surface around x, ΔA is defined as the area of a circle with the same radius r as the encompassing sphere, i.e.:

$$\Delta A = \pi r^2 \tag{10}$$

One of the issues with the raw radiance estimate – Equation 9 – is that the k-nearest photons typically have varying distance to the actual point of interest x. As the accuracy of a particular photon's contribution to the radiance estimate increases as its distance to x decreases, it makes sense that photons close to x should contribute more than photons far from x.

Thus, Jensen suggests weighing the contribution of each photon by some function K of its distance to x, i.e.:

$$L_r(x, \omega) \approx \sum_{p=1}^{k} f_r(x, \omega_p, \omega) \frac{\Delta\Phi_p(x, \omega_p)}{\Delta A} K(\|x_p - x\|) \tag{11}$$

where x_p is the position of the photon in the photon map. Notice how K is essentially an isotropic filter kernel, since it relies on the euclidean distance between x_p and x.

2.2.2 Hybrid approaches

Jensen [5, p.95] suggests that, in practice, mixing photon mapping with more elements of classic recursive ray tracing [11] can be beneficial, especially in terms of improving the quality of specular reflections without having to greatly increase the number of photons. The primary reason for this is that specular reflections, unlike diffuse reflections, are highly viewpoint-dependent, while the photon map is viewpoint-agnostic. Thus, by ignoring specular reflections in the photon tracing stage and instead tracing these backwards in the rendering stage, it is possible to get considerably better image quality without increasing the number of emitted photons.

Similarly, Jensen also suggests using a separate photon map to render caustics, generated by emitting photons specifically towards highly curved specular surfaces, since these are most likely to cause the formation of caustics. Caustics usually have sharp features, such as clearly defined edges, which can be difficult to reconstruct due to the averaging nature of the radiance estimate, which, using the k-nearest photons, blurs details in favour of less variance.

Neither of these optimizations are requirements of the basic photon mapping algorithm, and one can also think of them as sampling problems, i.e. problems that become smaller as the resolution of the photon map is increased. In this chapter I have chosen to focus on the photon mapping algorithm in its most basic form, and I refer the interested reader to Jensen's book [5] for further details on the various optimizations.

Section 3 examines an extension to photon mapping, namely photon differentials, that improves the ability to render caustics without using a separate photon map or increasing the number of photons.

2.3 Data structure

The photon map is a data structure that allows insertion of photons as well as searching for the k-nearest photons in any given location in a three-dimensional space. In principle, any type of data structure can be used, even a simple list, but for all but the most simple cases we will want to use an acceleration structure that allows searching for the k-nearest photons in less than linear time. This is because the rendering stage will be querying the photon map for the k-nearest photons quite often.

2.3.1 Kd-trees

Jensen [5] suggests using kd-trees to implement the photon map. Since kd-trees happen to subdivide space by axis-aligned splitting planes that intersect points in space, see Figure 4, this choice makes quite a lot of sense as long as photons have no size or shape associated with them.

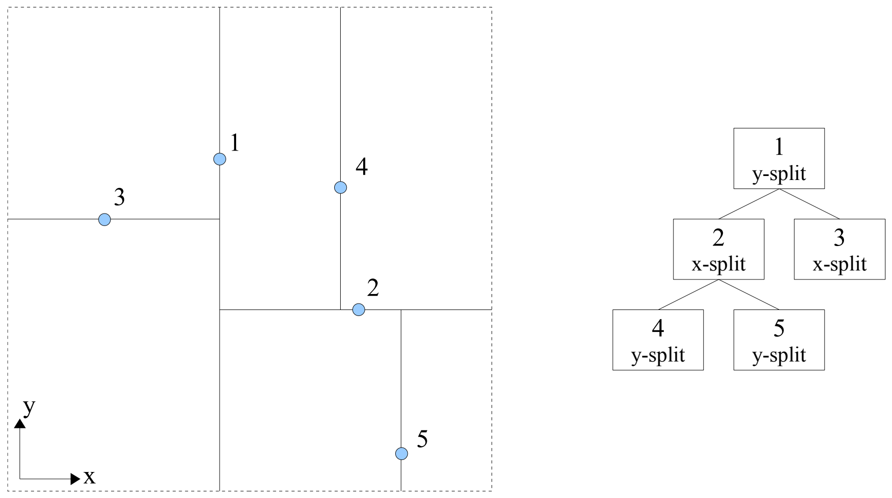


Figure 4: An example of a two-dimensional kd-tree with 5 nodes. Kd-trees subdivide space by axis-aligned splitting planes chosen in rotation. The subdivision of space is shown on the left, and the structure of the corresponding tree is shown on the right. In this case there are only two axis, hence the nodes partition the space along y, then x, then y, and so forth. Because each node is a point, the kd-tree is well suited to storing photons with no size or shape associated with them.

By implementing the photon map using a kd-tree, the photons themselves can act as the nodes in the tree, reducing both complexity and memory consumption. This data structure also allows searching for the k-nearest photons in logarithmic time; $O(k+\log n)$, where n is the number of photons and k is the desired number of neighbours.

2.3.2 Octrees

Another possibility is to use octrees, a three-dimensional variant of quadtrees [1]. Octrees are an interesting alternative as they employ fixed spatial subdivision, see Figure 5 on page 10, rather than the splitting-plane approach used by kd-trees.

The fixed subdivision of space makes octrees quite easy to grasp and visualize, and therefore also relatively easy to implement. For example, searching in an octree can be accomplished by tracing rays into the geometry of the tree, testing for intersections with the axis-aligned bounding boxes that define the nodes of the tree. Octrees also inherently support entities with actual volume and/or shape, which could potentially be useful with regards to photon differentials.

Perhaps the most obvious disadvantage of using octrees is that the tree cannot be rebalanced, and as such the tree will only be balanced if the data stored in it already has a uniform spatial distribution. This follows as a result of the fixed spatial subdivision.

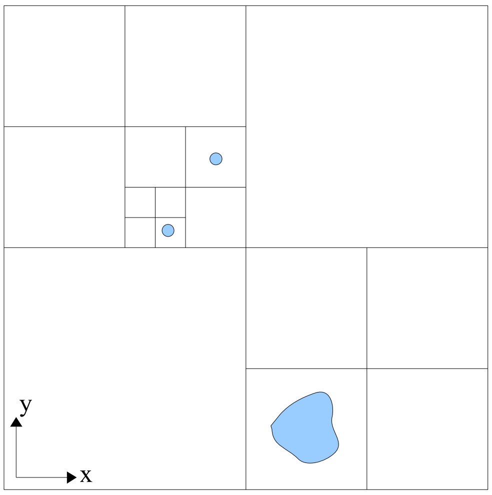


Figure 5: An example of a quadtree storing two points (upper left) and some complex geometry (lower right). In a quadtree, the nodes in the tree are squares, with the root node being the outermost square. Octrees are very similar, except that the nodes are cubes, and that each node has eight children rather than four. Insertion occurs from the root of the tree: If the data can fit completely into one of the child nodes of the current node, then the insertion continues in the child node, and otherwise the data is stored in the current node. Empty space is not partitioned since child nodes are created on demand.

2.4 Summary

To summarize, this is the behaviour of the standard photon mapping algorithm:

1. N_{scene} photons are emitted from the light sources in the scene. All photons are given the same radiant power, $\Phi_p = \Phi_{\text{scene}}/N_{\text{scene}}$, with the distinction being that more photons are emitted from bright light sources than dim light sources.

2. Each of the emitted photons is traced through the scene to simulate the scattering of light. Using russian roulette, only a single path is traced for each photon, and the radiant power of each photon stays constant for the entire length of its path. A photon is stored in the photon map every time it is reflected or refracted, i.e. in each point of intersection along its path until absorbed.

3. View rays are traced from the viewpoint and into the scene. For each ray-surface intersection, the k-nearest photons are gathered from the photon map and used as input to the radiance estimate. The result of the radiance estimate adds to the intensity of the corresponding pixel in the final image. Refracted rays are traced recursively to support transparent surfaces.

The next section describes a different variant of photon mapping, introduced by Schjøth et al. [8], in which photons are replaced by photon differentials. Unlike photons, photon differentials have size and shape associated with them, and this allows for a redefinition of the radiance estimate which preserves the shape of the emitted light.

3 Photon differentials

With photon mapping, recall that the propagation and distribution of light is approximated by emitting and tracing a finite number of photons from each light source. Because the resolution of light is infinite, each of the emitted photons can also be interpreted as the center of a beam whose size and shape changes as it propagates through the scene.

By keeping track of the beam surrounding a given photon, it is possible to approximate the shape of the light that reaches a given surface without having to increase the number of photons. Essentially, when a photon hits a surface, the projection of the surrounding beam onto the tangential surface in the point of intersection yields a footprint, and this footprint is an approximation of the shape of the light that reaches the intersecting surface.

Schjøth et al. [8] propose a variant of the standard photon mapping algorithm, in which each photon is associated with a *ray differential* to describe and keep track of the surrounding beam as it propagates through the scene. They refer to this combination of photons and ray differentials simply as *photon differentials*.

More specifically, a photon differential represents a beam of light whose size and shape changes as it propagates through the scene. It is defined by some radiant power Φ_{pd}, a position P and a direction V that describe the position and direction of the center of the beam, as well as a ray differential (dP, dV) that describes the positional and directional offsets of two imaginary adjacent rays that approximate the size and shape of the beam.

Although photon differentials completely replace photons in the variant of photon mapping proposed by Schjøth et al. [8], most of the basic aspects of the algorithm are indifferent from standard photon mapping, with the most essential differences being that the differentials dP and dV have to be initialized and updated during emission and scattering, as well as the radiance estimate, which is redefined to take into account the size and shape of the footprint.

First, section 3.1 describes the fundamental theory of ray differentials, as well as the basic operations for tracing them analytically. Then, based on the theory of ray differentials, section 3.2 describes the necessary changes to emission and scattering, while the redefined radiance estimate is described in section 3.3.

For a detailed evaluation of how photon mapping with photon differentials performs in comparison to standard photon mapping, I refer the interested reader to the original paper on photon differentials by Schjøth et al. [8].

3.1 Ray differentials

A ray can be defined by its position P and a direction V. If a ray with position P is known to pass through a point x, then its direction V can also be written as a function of the vector from P to x, i.e.:

$$V = \frac{v}{\|v\|} \tag{12}$$

where

$$v = x - P \tag{13}$$

A ray differential is a first order approximation of the change in a ray's position and direction with respect to one or more of the initial parameters of the ray. In other words, a ray differential consists of two differentials, dP and dV, which can be used to infer the position and direction of an imaginary adjacent ray.

For example, given a ray whose position P is constant and whose direction V therefore only depends on x, the differential dV approximates the change in V for some change in x. Figure 6 illustrates the relationship between the change in x, the differential dV, and the direction $V + dV$ of the imaginary adjacent ray.

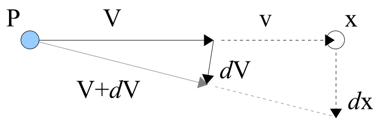


Figure 6: Illustrates the relationship between the differential dV of a ray's direction V for some change in x, dx, as well as the inferred direction $V + dV$ of the resulting imaginary adjacent ray.

Ray differentials were first introduced by Igehy [3], who originally used them to perform anisotropic texture filtering. More recently, Sporring et al. [9] described them in a more general form, using the notation and algebra of Magnus and Neudecker [7] to support concatenation of operations by matrix multiplication, and also extended them to the temporal domain.

A very nice property of ray differentials is that they can be propagated through the scene alongside the original ray, simply by evaluating the differentials of the same operations used to propagate the original ray. Figure 7 illustrates the relationship between a ray and the imaginary adjacent ray represented by a ray differential as both propagate through the scene.

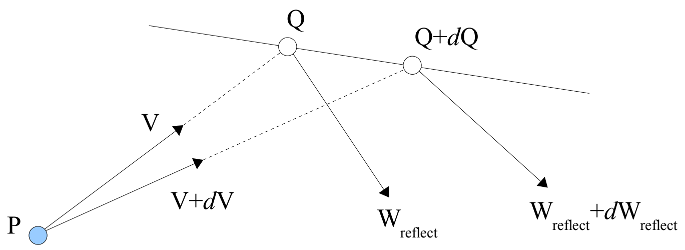


Figure 7: The relationship between a ray and the imaginary adjacent ray represented by a ray differential as both propagate through the scene. In this example, the original ray is transferred to an intersecting surface and then reflected. P is the initial position of the ray, while Q is its position following the transfer. Similarly, V is the initial direction of the ray, while W_{reflect} is its direction following the reflection. The position and direction of the imaginary adjacent ray is inferred at each step by taking the sum of the ray's position and direction and the respective components of the ray differential. The operations for transfer and reflection are derived in sections 3.1.2 and 3.1.3.

Igehy [3] describes the necessary operations for transfer, reflection, and refraction, but skips over some detail in the derivations of these. On the other hand, Sporring et al. [9] go into great detail in the derivations, but the general form of their equations requires a bit more of the reader.

The remainder of this section describes the initialization of a ray differential and derives the operations for transfer, reflection, and refraction. I will be using the notation of Sporring et al. [9], while following the assumptions of Igehy [3] to simplify some of the terms.

3.1.1 Initialization

Given a ray with position P and direction V, the ray differential (dP, dV) with respect to the parameters of P or V can be found by applying the differential operator directly to P and V.

Starting with dV, note that $\|v\| = \sqrt{v^T v} = (v^T v)^{\frac{1}{2}}$, and that V therefore also can be written:

$$V = v(v^T v)^{-\frac{1}{2}} \tag{14}$$

Applying the differential operator to the above definition of V yields:

$$\begin{aligned}
dV &= d\left(v(v^T v)^{-\frac{1}{2}}\right) \\
&= (dv)(v^T v)^{-\frac{1}{2}} + vd\left((v^T v)^{-\frac{1}{2}}\right) \\
&= (dv)(v^T v)^{-\frac{1}{2}} + v\left(-\frac{1}{2}(v^T v)^{-\frac{3}{2}} d(v^T v)\right) \\
&= (dv)(v^T v)^{-\frac{1}{2}} + v\left(-\frac{1}{2}(v^T v)^{-\frac{3}{2}} \left((dv^T)v + v^T dv\right)\right) \\
&= (dv)(v^T v)^{-\frac{1}{2}} + v\left(-\frac{1}{2}(v^T v)^{-\frac{3}{2}} 2v^T dv\right) \\
&= (dv)(v^T v)^{-\frac{1}{2}} - v(v^T v)^{-\frac{3}{2}} v^T dv
\end{aligned} \tag{15}$$

To continue the derivation, observe how dv can move freely in the left term because $(v^T v)$ is a scalar. However, also note that moving out dv requires the introduction of the identity matrix I, since one cannot subtract a matrix from a scalar. The final steps follow:

$$\begin{aligned}
dV &= \left((v^T v)^{-\frac{1}{2}} I - v(v^T v)^{-\frac{3}{2}} v^T\right) dv \\
&= \frac{(v^T v)^{\frac{3}{2}}}{(v^T v)^{\frac{3}{2}}} \left((v^T v)^{-\frac{1}{2}} I - v(v^T v)^{-\frac{3}{2}} v^T\right) dv \\
&= \frac{1}{(v^T v)^{\frac{3}{2}}} \left((v^T v)I - vv^T\right) dv \\
&= \frac{(v^T v)I - vv^T}{(v^T v)^{\frac{3}{2}}} dv
\end{aligned} \tag{16}$$

Note that expressing the differential dV with respect to dv results in the Jacobian of V, which is a matrix containing the partial derivatives of V with respect to each component of v:

$$\frac{dV}{dv} = \frac{(v^T v)I - vv^T}{(v^T v)^{\frac{3}{2}}} \tag{17}$$

where

$$\left[\frac{dV}{dv}\right]_{i,j} = \frac{\partial V_i}{\partial v_j} \tag{18}$$

Finally, from Equation 13 it is obvious that $dv = (dx - dP)$. Thus, under the assumption that the ray's position P has no dependencies, it follows that the differential dP of the ray's position is zero, and that the differential dV of the ray's direction only depends on the change in x, dx:

$$\begin{aligned} dP &= 0 \qquad (19)\\ dV &= \frac{(v^Tv)I - vv^T}{(v^Tv)^{\frac{3}{2}}}dx \qquad (20) \end{aligned}$$

Once initialized, the ray differential (dP, dV) can be traced alongside the original ray using the operations for transfer, reflection and refraction.

3.1.2 Transfer

The transfer operation shifts a ray's position to its point of intersection with a surface in the scene. Given a ray with position P and an intersection point Q_0, the ray's transferred position Q can be written as:

$$Q = P + sV \tag{21}$$

where s is the distance to the surface along the ray's direction V.

Applying the differential operator to Equation 21 yields the formula for transferring the corresponding ray differential. Given a ray differential (dP, dV), the differential dQ of the ray's transferred position can be written as:

$$dQ = dP + (ds)V + sdV \tag{22}$$

To actually solve Equation 22 it is necessary to derive ds. Given the surface normal N in the point of intersection Q_0, s can be expressed as the ratio between the two dot products:

$$s = \frac{(Q_0 - P)^TN}{V^TN} \tag{23}$$

Applying the differential operator to the above definition of s yields:

$$\begin{aligned} ds &= d\left(\frac{(Q_0 - P)^TN}{V^TN}\right)\\ &= \frac{(V^TN)d\left((Q_0 - P)^TN\right) - \left(d(V^TN)\right)(Q_0 - P)^TN}{(V^TN)^2}\\ &= \frac{(dQ_0 - dP)^TN}{V^TN} + \frac{(Q_0 - P)^TdN}{V^TN} - \frac{sd(V^TN)}{V^TN}\\ &= \frac{(dQ_0 - dP)^TN}{V^TN} + \frac{(Q_0 - P)^TdN}{V^TN} - \frac{sN^TdV}{V^TN} - \frac{sV^TdN}{V^TN}\\ &= \frac{N^T}{V^TN}dQ_0 - \frac{N^T}{V^TN}dP + \frac{(Q_0 - P)^T - sV^T}{V^TN}dN - \frac{sN^T}{V^TN}dV \qquad (24) \end{aligned}$$

Sporring et al. [9] proceeds with this general form of ds while Igehy [3] makes the assumption that the intersection point Q_0 is fixed and that surface normal N is constant over the intersecting surface. By zeroing out these terms in Equation 24, it is possible to verify that the result is equal to that of Igehy [3]:

$$\begin{aligned} ds &= -\frac{N^T}{V^T N} dP - \frac{sN^T}{V^T N} dV \\ &= -\frac{N^T dP + sN^T dV}{V^T N} \\ &= -\frac{(dP^T + sdV^T)N}{V^T N} \end{aligned} \tag{25}$$

Following Igehy's assumption of a fixed intersection point and constant surface normal, i.e. plugging Equation 25 into Equation 22, a readily solvable formula for the differential dQ of the ray's transferred position is found to be:

$$\begin{aligned} dQ &= dP - \left(\frac{N^T}{V^T N} dP + \frac{sN^T}{V^T N} dV \right) V + sdV \\ &= dP - \frac{N^T dP}{V^T N} V - \frac{sN^T dV}{V^T N} V + sdV \\ &= dP - \frac{VN^T}{V^T N} dP - \frac{sVN^T}{V^T N} dV + sdV \\ &= \left(I - \frac{VN^T}{V^T N} \right) dP - \left(\frac{VN^T + I}{V^T N} \right) sdV \end{aligned} \tag{26}$$

Refer to Sporring et al. [9] for an elaborate definition of dQ that does not assume constant intersection point nor constant surface normal. Figure 8 illustrates the transfer operation geometrically using the simplified formulation of ds.

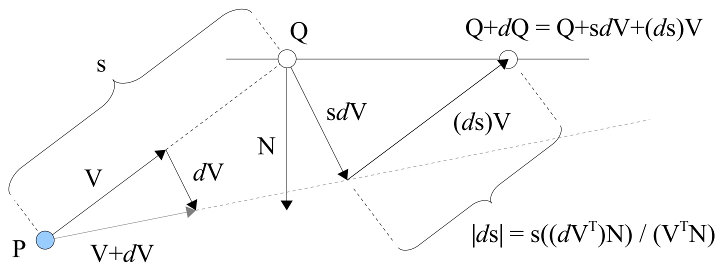


Figure 8: An illustration of the involved terms in the transfer of a ray differential, assuming that $dP = 0$, and assuming a flat surface using the simplified formulation of ds shown in Equation 25. The proportions are approximate but roughly correct. Notice how this shows how dQ only approximates the offset to the intersection point between the imaginary adjacent ray (grey dotted line inferred by $V + dV$) and the surface.

3.1.3 Reflection

The reflection operation reflects a ray's direction based on its incident angle to an intersecting surface. It assumes that the ray has already been transferred to the surface in question. Given a ray with direction V and the surface normal N at the intersection point, the ray's reflected direction W_{reflect} can be written as:

$$W_{\text{reflect}} = V - 2(V^T N)N \tag{27}$$

Applying the differential operator to Equation 27 yields the basic formula for reflecting the corresponding ray differential. Given a ray differential (dP, dV), the differential dW_{reflect} of the ray's reflected direction can be written as:

$$\begin{aligned} dW_{\text{reflect}} &= dV - 2d\left((V^T N)N\right) \\ &= dV - 2\left((d(V^T N))N + (V^T N)dN\right) \\ &= dV - 2(V^T N)dN - 2\left((dV^T)N + V^T dN\right)N \\ &= dV - 2(V^T N)dN - 2(N^T dV)N - 2(V^T dN)N \\ &= dV - 2(V^T N)dN - (2NN^T)dV - (2NV^T)dN \\ &= (I - 2NN^T)dV - 2(V^T NI - NV^T)dN \end{aligned} \tag{28}$$

Evaluating dW_{reflect} requires the differential dN of the surface normal N. This derivation is outside the scope of this report, but one may refer to [3] for a discussion of the concept of using a so-called shape operator, or Sporring et al. [9] who derive dN in full for phong shaded surfaces with vertex-interpolated surface normals.

3.1.4 Refraction

The refraction operation refracts a ray's direction based on its incident angle to an intersecting surface using Snell's law of refraction [3][9]. Just like the reflection operation, this operation also assumes that the ray has already been transferred to the surface in question. Given a ray with direction V and the intersecting surface's surface normal N, the ray's refracted direction W_{refract} can be written as:

$$W_{\text{refract}} = \eta V - \mu N \tag{29}$$

where η is the ratio of refraction, and where:

$$\mu = \eta V^T N + \sqrt{\varepsilon} \tag{30}$$

$$\varepsilon = 1 - \eta^2\left(1 - (V^T N)^2\right) \tag{31}$$

Applying the differential operator to Equation 29 yields the formula for refracting the corresponding ray differential. Given a ray differential (dP, dV), the differential dW_{refract} of the ray's refracted direction can be written as:

$$\begin{aligned} dW_{\text{refract}} &= d\eta V + \eta dV - d\mu N - \mu dN \\ &= V d\eta + \eta dV - N d\mu - \mu dN \end{aligned} \tag{32}$$

To actually evaluate the above definition of dW_{refract} it is necessary to derive $d\eta$, $d\mu$, and dN. For flat surfaces separating two homogeneous media, the surface normal N and the refraction ratio η are constants across the intersecting surface, which implies that the corresponding differentials are zero, i.e. $dN = 0$ and $d\eta = 0$. This assumption leaves only the derivation of $d\mu$, which relies on $d\varepsilon$.

Applying the differential operator to Equation 31 yields $d\eta$:

$$\begin{aligned} d\varepsilon &= 0 - \left(d(\eta^2)\right)\left(1-(V^TN)^2\right) - \eta^2 d\left(1-(V^TN)^2\right) \\ &= -2\eta(d\eta)\left(1-(V^TN)^2\right) - \eta^2\left(0 - 2V^TN((dV^T)N + V^TdN)\right) \\ &= -2\eta\left(1-(V^TN)^2\right)d\eta + 2\eta^2 V^TN(N^TdV + V^TdN) \end{aligned} \tag{33}$$

Similarly, given $d\varepsilon$, $d\mu$ is derived from Equation 30 as follows:

$$\begin{aligned} d\mu &= (d\eta)V^TN + \eta\left((dV^T)N + V^TdN\right) + d\sqrt{\varepsilon} \\ &= V^TNd\eta + \eta(N^TdV + V^TdN) + \frac{1}{2}\varepsilon^{-\frac{1}{2}}d\varepsilon \\ &= V^TNd\eta + \eta(N^TdV + V^TdN) + \frac{d\varepsilon}{2\sqrt{\varepsilon}} \\ &= V^TNd\eta - \frac{2\eta\left(1-(V^TN)^2\right)}{2\sqrt{\varepsilon}}d\eta + \\ &\qquad \eta(N^TdV + V^TdN) + \frac{2\eta^2V^TN(N^TdV + V^TdN)}{2\sqrt{\varepsilon}} \\ &= \left(V^TN - \frac{1-\varepsilon}{\eta\sqrt{\varepsilon}}\right)d\eta + \\ &\qquad \eta\left(1 + \frac{\eta V^TN}{\sqrt{\varepsilon}}\right)N^TdV + \eta\left(1 + \frac{\eta V^TN}{\sqrt{\varepsilon}}\right)V^TdN \end{aligned} \tag{34}$$

And finally, substituting the above definition of $d\mu$ into Equation 32 yields the full definition of dW_{refract}:

$$\begin{aligned} dW_{\text{refract}} &= Vd\eta - N\left(V^TN - \frac{1-\varepsilon}{\eta\sqrt{\varepsilon}}\right)d\eta + \\ &\qquad \eta dV - \eta N\left(1 + \frac{\eta V^TN}{\sqrt{\varepsilon}}\right)N^TdV - \\ &\qquad \mu dN - \eta N\left(1 + \frac{\eta V^TN}{\sqrt{\varepsilon}}\right)V^TdN \\ &= \left(V - N\left(V^TN - \frac{1-\varepsilon}{\eta\sqrt{\varepsilon}}\right)\right)d\eta + \\ &\qquad \left(\eta I - \eta N\left(1 + \frac{\eta V^TN}{\sqrt{\varepsilon}}\right)N^T\right)dV - \\ &\qquad \left(\mu I - \eta N\left(1 + \frac{\eta V^TN}{\sqrt{\varepsilon}}\right)V^T\right)dN \end{aligned} \tag{35}$$

where the first and third terms evaluate to zero for flat surfaces separating two homogeneous media. Akin to the derivation of the reflection operation in section 3.1.3, refer to Igehy [3] and Sporring et al. [9] for details on the derivation of dN for non-flat surfaces.

3.2 Emission and scattering

Based on the properties of the emitting light source, each photon differential is assigned a position P, a direction V, and some radiant power Φ_{pd}. The initialization of these properties follow the same principles as when regular photons are emitted as described in section 2.1.1, with the addition being the initialization of (dP, dV). Using the definition of V from Equation 12 in section 3.1.1:

$$\begin{aligned} V &= \frac{v}{\|v\|} \\ &= \frac{x-P}{\|x-P\|} \end{aligned}$$

and assuming that P is constant, dP and dV are given by:

$$\begin{aligned} dP &= 0 \\ dV &= \frac{(v^T v)I - vv^T}{(v^T v)^{\frac{3}{2}}} dx \end{aligned}$$

For a photon differential, dx is a 3×2 matrix whose columns describe the changes in x that govern the initial change in V for each of two imaginary adjacent rays. Using α to denote the vector parameter of the first imaginary ray, and β to denote the vector parameter of the second imaginary ray, dx can be defined as:

$$dx = \begin{bmatrix} \alpha & \beta \end{bmatrix} \tag{36}$$

Schjøth et al. [8] suggest that, ideally, for each photon differential emitted by a given light source, α and β should be chosen such that the footprint of the resulting beam represents a fraction of the area of the unit sphere proportional to the number of photon differentials emitted by that particular light source.

Based on the definition of the area of the footprint given in section 3.2.1, and assuming that $v = V$, which implies that x is a point on the unit sphere around P, this is possible by choosing α and β such that these describe two adjacent sides of a square in the tangent plane, whose center is x, and whose area is a fraction of the total surface area 4π of the unit sphere.

Basically, pick two orthogonal unit vectors in the tangent plane in x, and then scale them such that they span a quarter of the desired area. More formally, first pick a random unit vector u where $|u \cdot V| \neq 1$, i.e. where u is *not* parallel to V. Then, using u to obtain the vectors in the tangent plane, α and β are given by:

$$\alpha = \frac{1}{2}\sqrt{\frac{4\pi}{N}}\,(V \times u) \tag{37}$$

$$\beta = \frac{1}{2}\sqrt{\frac{4\pi}{N}}\,(V \times (V \times u)) \tag{38}$$

where N is the number of photon differentials emitted by the light source.

After emission, each photon differential is treated as a ray that is traced through the scene until absorbed. This procedure is mostly similar to that of a normal photon as described in section 2.1.2, also using russian roulette, but with the addition that (dP, dV) is updated in each point of intersection Q. This is accomplished using the operations for transfer, reflection and refraction as described in sections 3.1.2 through 3.1.4.

Updating (dP, dV) is done in two steps. First the transfer operation is used to transfer dP to the intersecting surface, yielding dQ. Then, based on the outcome of the russian roulette selection scheme, dV is either reflected or refracted by the intersecting surface, yielding either dW_{reflect} or dW_{refract}.

When (dQ, dW) has been found, then the photon differential is stored in the photon map. More specifically, the photon map stores the photon differential in the point of intersection Q, along with its radiant power Φ_{pd}, the incident direction V, and its footprint on the tangential surface in the point of intersection Q. The definition of the footprint is given in section 3.2.1.

Finally, (dQ, dW) replaces (dP, dV), i.e.:

$$\begin{aligned} dP &\leftarrow dQ \\ dV &\leftarrow dW \end{aligned}$$

and the trace continues.

3.2.1 Footprint of photon differential in Q

The footprint of a photon differential on the tangential surface in an intersection point Q is defined by the positional offsets of the two imaginary adjacent rays described by (dQ, dW). The positional offsets are given by the differential of Q for each of the initial emission parameters α and β. Schjøth et al. [8] refer to these offsets as the differential position vectors. Based on the definition of dx in Equation 36, they can be obtained directly from the columns of dQ as follows:

$$\begin{bmatrix} d_\alpha Q & d_\beta Q \end{bmatrix} = dQ \tag{39}$$

where $d_\alpha Q$ denotes the differential position vector of the first imaginary ray, and $d_\beta Q$ denotes the differential position vector of the second. See Figure 9.

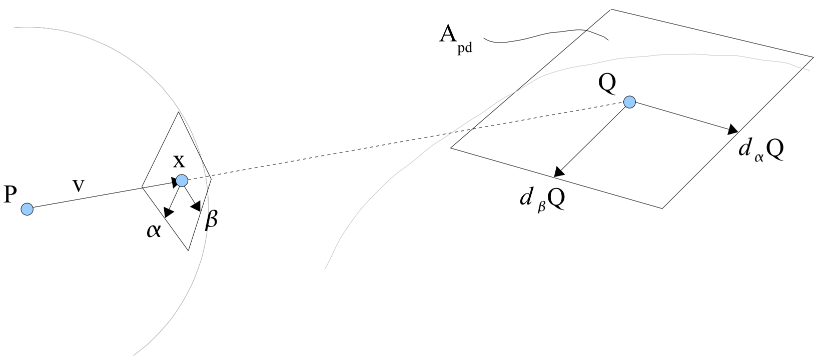


Figure 9: Illustration of the footprint of a photon differential in Q after emission from a light source in P followed by the initial trace to the first intersecting surface. Note the relationship between the plane spanned by the differential position vectors $d_\alpha Q$ and $d_\beta Q$ and the projected area A_{pd}.

Schjøth et al. [8] define the area A_{pd} of the footprint as the area of the parallelogram spanned by the differential position vectors $d_\alpha Q$ and $d_\beta Q$. Under the assumption that Q is the center of the photon differential on the tangential surface, arguably the parallelogram should be spanned by $2d_\alpha Q$ and $2d_\beta Q$, and offset by $[\ -d_\alpha Q \quad -d_\beta Q\ ]^T$, in order to respect Q as the center.

Thus, noting that the area of a parallelogram is the length of the cross product of two of its adjacent sides, A_{pd} is given by:

$$A_{pd} = \|\ (2d_\alpha Q) \times (2d_\beta Q)\ \| \tag{40}$$

As mentioned in section 3.2, the footprint $(d_\alpha Q, d_\beta Q)$ is stored in the photon map alongside the other properties of a photon differential. This is in order to make $(d_\alpha Q, d_\beta Q)$, and thereby also A_{pd}, available to the redefined radiance estimate, which is described in the following section.

3.3 Computing the reflected radiance

Schjøth et al. [8] redefine the radiance estimate such that it takes into account the size and shape of the footprint $(d_\alpha Q, d_\beta Q)$ of each of the involved photon differentials. In brief, the size of the footprint is used in the approximation of the irradiance in the sampling point x, while the shape of the footprint is used to define an anisotropic filter kernel centered around the point x_{pd} where the photon differential was stored in the photon map.

Recall from Equation 8 in section 2.2.1 that the reflected radiance is given by:

$$L_r(x,\omega) \quad = \quad \int_{\Omega_x} f_r(x,\omega',\omega)\frac{d^2\Phi_i(x,\omega')}{dA_i}$$

As described in section 2.2.1, in the radiance estimate based on regular photons it was necessary to approximate the projected area dA_i by the projected area of the sphere encompassing the k-nearest photons, simply because regular photons do not have any size or shape.

Given a photon differential on the other hand, the area A_{pd} of its footprint $(d_\alpha Q, d_\beta Q)$ is a direct approximation of the projected area affected by its radiant power Φ_{pd}. These quantities are related to the irradiance in the point x_{pd} where the photon differential was stored in the photon map, which prompts a different solution to the above equation.

Jensen [5] defines the irradiance in some point x as the incident radiant power Φ_i per unit projected area dA_i:

$$E(x) \quad = \quad \frac{d\Phi_i(x)}{dA_i} \tag{41}$$

Schjøth et al. [8] remark that by adding a directional dependency ω' to the incident radiant power Φ_i, the result is the irradiance in x due to the incident radiant power from that particular direction:

$$E(x,\omega') \quad = \quad \frac{d\Phi_i(x,\omega')}{dA_i} \tag{42}$$

and this is exactly what is provided by a photon differential for $x = x_{pd}$, noting that $\Phi_{pd} \approx d\Phi(x_{pd}, \omega')$ and $A_{pd} \approx dA_i$. Substitution of this definition in Equation 8 results in the following definition of the reflected radiance L_r:

$$L_r(x, \omega) \quad = \quad \int_{\Omega_x} f_r(x, \omega', \omega) dE(x, \omega') \tag{43}$$

Under the assumption that $x = x_{pd}$, the continuous integral in Equation 43 can be approximated by the following sum over the k-nearest photons:

$$\begin{aligned} L_r(x, \omega) &\approx \sum_{pd=1}^{k} f_r(x, \omega_{pd}, \omega) \Delta E(x_{pd}, \omega_{pd}) \\ &= \sum_{pd=1}^{k} f_r(x, \omega_{pd}, \omega) \frac{\Phi_{pd}}{A_{pd}} \end{aligned} \tag{44}$$

Obviously, the above approximation becomes less and less accurate as x diverges from x_{pd}. Similar to the filtered radiance estimate in the standard photon mapping algorithm, see Equation 11 in section 2.2.1, Schjøth et al. [8] choose to weigh the contribution of each photon differential by its distance to the sampling point x. However, instead of simply using the distance $\|x_{pd} - x\|$, the contribution of a photon differential is based on a function of the vector from x_{pd} to x that takes into account the shape of the footprint.

The footprint $(d_\alpha Q, d_\beta Q)$ of the photon differential can be interpreted as two axis of a skew ellipsoid that defines *filter space* with respect to world space. Thus, the matrix M_{pd} that transforms from world space to filter space is given by:

$$M_{pd} \quad = \quad \begin{bmatrix} d_\alpha Q & d_\beta Q & \dfrac{d_\alpha Q \times d_\beta Q}{\|d_\alpha Q \times d_\beta Q\|} \end{bmatrix}^{-1} \tag{45}$$

By applying M_{pd} to the vector from the photon differential to the sampling point, $x - x_{pd}$, the result is a vector in anisotropic filter space, $M_{pd}(x - x_{pd})$, the length of which can be used as input to an isotropic filter kernel. Putting it all together, the filtered radiance estimate for photon differentials is defined as:

$$L_r(x, \omega) \quad \approx \quad \sum_{pd=1}^{k} f_r(x, \omega_{pd}, \omega) \frac{\Phi_{pd}}{A_{pd}} K(\|M_{pd}(x - x_{pd})\|) \tag{46}$$

where K defines the filter kernel. Because the input to K is given with respect to the anisotropic filter space defined by the footprint of the photon differential, K essentially behaves like an anisotropic filter kernel defined by the footprint of the photon differential.

Finally, Schjøth et al. [8] also specifically mention rejecting those of the k-nearest photon differentials whose footprints do not overlap the sampling point x, excluding these from the radiance estimate.

Note that this is a departure from the filtered radiance estimate in the standard photon mapping algorithm, see Equation 11 in section 2.2.1, where all of the k-nearest photons contribute to the reflected radiance in some capacity, though far-away photons may contribute very little.

4 Incorporating the view ray differential

With both standard photon mapping and photon mapping with photon differentials, see sections 2 and 3, the rendering stage traditionally revolves around tracing infinitesimal view rays from the viewpoint. For each intersection between a view ray and a surface, the radiance estimate is based on just the intersection point x, also sometimes referred to as the sampling point, and the view ray's incident direction ω.

However, like photons in the transition to photon differentials, a view ray can also be interpreted as part of a ray bundle or beam. Specifically, because a finite number of view rays are traced for each pixel in the final image, and because each pixel spans an area on the view plane which has infinite resolution, each view ray can also be interpreted as the center of a beam whose size and shape changes as it propagates through the scene.

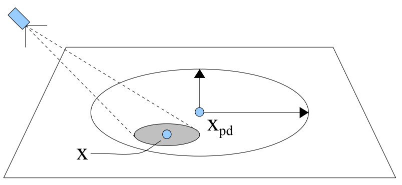
Consequently, a radiance estimate which is based on just the intersection point x essentially assumes that the corresponding beam is very small, i.e. that a large number of view rays are traced into the scene for every pixel in the final image. Thus, the accuracy of the estimate can be improved by increasing the number of view rays per pixel, but this is also quite costly in terms of required rendering time since these are typically proportional.

Conversely, by taking into account the size and shape of the beam in each point of intersection, it should be possible to improve the accuracy of the radiance estimate without increasing the number of view rays per pixel, the resolution of the photon map, or the number of evalauted photon differentials in each point of intersection.

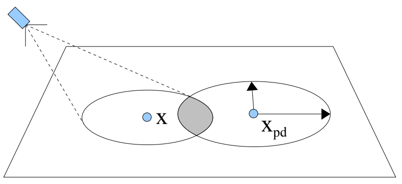
By associating each view ray with a ray differential, tracing the ray differential alongside the view ray as it propagates through the scene, it is possible to approximate the size and shape of the corresponding beam in each point of intersection. I refer to the association of view rays and ray differentials as *view ray differentials*. Similar to a photon differential, a view ray differential defines the footprint of the beam in each point of intersection, and this footprint can be taken into account in the radiance estimate.

For photon mapping with photon differentials, the inclusion of the footprint of the view ray differential prompts the idea that the contribution of each photon differential could be computed based on the intersection between the pair of footprints, rather than just the filter space distance to the sampling point x. Figure 10 on page 23 highlights three different scenarios, two of which should benefit from taking into account the intersection between the involved footprints.

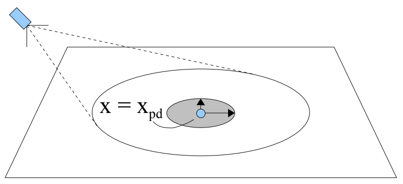
This section explores two different approaches which take into account the intersection of the involved footprints. Both approaches define a new radiance estimate based on the filtered radiance estimate for photon mapping with photon differentials, see section 3.3. As a precursor, the initialization of a view ray differential is described in section 4.1.



(a) The sampling point x lies within the photon differential's range of influence, and the footprint of the view ray differential is comparatively small.



(b) The sampling point x lies outside the photon differential's range of influence, but the intersection between the involved footprints lies *inside* this range.



(c) The sampling point x lies in the center of the photon differential, but the footprint of the view ray differential also spans a much larger area than the area of the intersection.

Figure 10: Three example scenarios of the footprint of a view ray differential overlapping the footprint of a photon differential. In these examples the footprints are coplanar, but note that this is just for the sake of the simplicity of the illustration. The footprint of the view ray differential is illustrated by the ellipse at the end of the beam extending from the viewpoint, with x being the sampling point, while the footprint of the photon differential is illustrated by the ellipse with center in x_{pd}. The filled grey area marks the intersection between them. In (a), the footprint of the view ray differential is relatively small, and in this case the filtered radiance estimate based on just the sampling point x is an acceptable approximation. However, observe how in (b), the filtered radiance estimate based on just the sampling point x will result in zero contribution, despite part of the beam represented by the view ray differential actually overlapping the photon differential's range of influence. Also observe how in (c), the filtered radiance estimate will result in full contribution, despite the footprint of the photon differential covering only a fraction of the footprint of the view ray differential, thus affecting only a fraction of the beam and therefore only a fraction of the pixel in the final image.

The first approach, described in section 4.2, explores the possibility of approximating the coverage and centroid of the intersection based on a two-dimensional reconstruction of the geometry of the involved footprints, using the coverage and centroid of the intersection to redefine the contribution of a photon differential in the radiance estimate.

The second approach, described in section 4.3, explores the possibility of multisampling the photon differential, using the footprint of the view ray differential to define the sample distribution, thus implicitly taking into account the intersection between the involved footprints.

4.1 Initialization of view ray differential

A view ray is a ray whose starting point P is the viewpoint, and whose direction V describes a trajectory that intersects the view plane in some point x which corresponds to the center of a pixel in the final image. Note that x in the definition of V is not to be confused with the variable of the same name in the radiance estimate. Using the definition of V from Equation 12 in section 3.1:

$$\begin{aligned} V &= \frac{v}{\|v\|} \\ &= \frac{x - P}{\|x - P\|} \end{aligned}$$

Similar to the initialization of a photon differential, section 3.2, assuming that P is constant, the ray differential (dP, dV) of a view ray differential is given by:

$$\begin{aligned} dP &= 0 \\ dV &= \frac{(v^T v)I - vv^T}{(v^T v)^{\frac{3}{2}}} dx \end{aligned}$$

where dx is a 3×2 matrix whose columns describe the changes in x that govern the initial change in V for each of two imaginary adjacent rays. Using α to denote the vector parameter of the first imaginary ray, and β to denote the vector parameter of the second imaginary ray, dx can be defined as:

$$dx = \begin{bmatrix} \alpha & \beta \end{bmatrix} \tag{47}$$

To choose α and β, observe that the view plane can be divided into segments, where each segment corresponds to a pixel in the final image. For the view ray differential to approximate a beam that occupies a full pixel in the final image, α and β should be chosen such that they describe the world space extents of the corresponding view plane segment. This is illustrated in Figure 11.

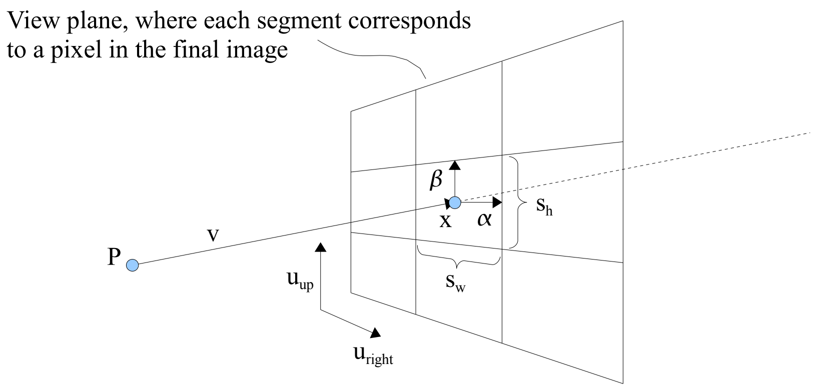

View plane, where each segment corresponds to a pixel in the final image
β
x
α
s_h
v
P
u_up
s_w
u_right

Figure 11: Illustration of the vector parameters α and β in relation to the view plane segment with center in x. Each segment on the view plane corresponds to a pixel in the final image, and α and β should be chosen such that they cover the extents of the segment.

Given the world space width and height of a view plane segment, s_{w} and s_{h}, as well as two unit vectors that describe the horizontal and vertical axis of the view plane in world space, u_{right} and u_{up}, α and β can be written as follows:

$$\alpha = \frac{1}{2} s_{\mathrm{w}} u_{\mathrm{right}} \tag{48}$$

$$\beta = \frac{1}{2} s_{\mathrm{h}} u_{\mathrm{up}} \tag{49}$$

The process of tracing a view ray differential is similar to that of tracing a photon differential. In each point of intersection Q, the ray differential (dP, dV) is first transferred, yielding dQ, and then either reflected or refracted, yielding dW. The operations for transfer, reflection and refraction are described in sections 3.1.3 through 3.1.4.

Perhaps most importantly, the transferred differential position vectors $d_\alpha Q$ and $d_\beta Q$ describe the footprint of the view ray differential on the tangential surface in each point of intersection Q, thus allowing the footprint of the view ray differential to be taken into account in the radiance estimate, noting that $x = Q$ in the radiance estimate.

4.2 Coplanar intersection-weighted photon differentials

Given a view ray differential with footprint $(d_\alpha Q_{vd}, d_\beta Q_{vd})$ in the sampling point x, the surface normal n_x in x, and the direction ω of the path towards the view-point, the intersection with a photon differential with footprint $(d_\alpha Q_{pd}, d_\beta Q_{pd})$ in x_{pd} can be approximated by the two-dimensional intersection between the shapes inferred by the coplanar projections of the two footprints.

This section describes an approach which takes into account the intersection between the involved footprints based two properties of their coplanar intersection. Specifically, the coplanar intersection can be used to obtain the photon differential's coverage of the view ray differential, as well as an approximation of the centroid of the actual intersection, both of which are quantities that can be used to refine the radiance estimate in terms of handling the scenarios shown in Figure 10 on page 23.

4.2.1 Projection and rotation

Each footprint spans a completely arbitrary plane of some offset in space. Thus, to be able to compute the coplanar intersection, the first step is to project them onto a common plane. Because the goal is to compute the visible intersection between the involved footprints, the direction of the projection is given by ω, which is the direction of the first leg of the path towards the viewpoint for the current view ray.

In principle, given ω as the direction of projection, any plane that intersects ω may be chosen as the plane onto which both footprints are projected. However, because more than one photon differential will likely be considered for each view ray differential, it makes sense to let the footprint of the view ray differential define the common plane of projection. This way, the projection of the view ray differential is already given, and the plane of projection remains the same for each of the considered photon differentials.

Thus, noting that x is a point in the plane spanned by the footprint of the view ray differential, and that n_x is its surface normal, the projection g of some point p along ω is given by:

$$g(p) \quad = \quad p - \omega \left(\frac{n_x \cdot (p - x)}{n_x \cdot \omega} \right) \qquad (50)$$

Applying g to the position x_{pd} of the photon differential yields its projected position x'_{pd}:

$$x'_{pd} \quad = \quad g(x_{pd}) \tag{51}$$

The projected footprint $(d_\alpha Q'_{pd}, d_\beta Q'_{pd})$, illustrated in Figure 12, is obtained by projecting the corresponding translations of the original position x_{pd} and then subtracting the projected position x'_{pd}:

$$d_\alpha Q'_{pd} \quad = \quad g(x_{pd} + d_\alpha Q_{pd}) - x'_{pd} \tag{52}$$
$$d_\beta Q'_{pd} \quad = \quad g(x_{pd} + d_\beta Q_{pd}) - x'_{pd} \tag{53}$$

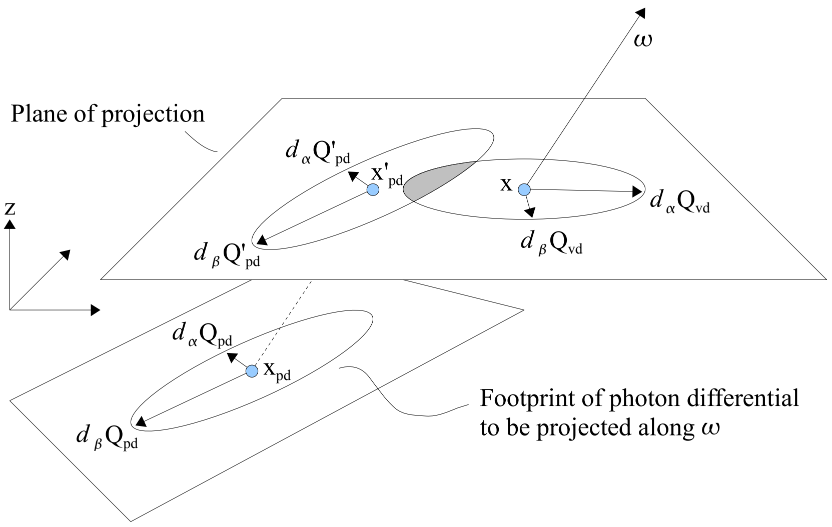


Figure 12: Illustrates the projection of the footprint and position of a photon differential, $(d_\alpha Q_{pd}, d_\beta Q_{pd})$ and x_{pd}, onto the plane of projection as defined by the footprint of the view ray differential, $(d_\alpha Q_{vd}, d_\beta Q_{vd})$. The direction of the projection is given by ω, and the result is the projected footprint and projected position of the photon differential, $(d_\alpha Q_{pd}, d_\beta Q_{pd})$ and x'_{pd}.

Computing the intersection between two shapes in the same plane is essentially a two-dimensional problem. However, because the orientation of the plane of projection is arbitrary, it cannot be treated as a two-dimensional problem until the plane is aligned with two axis of the root coordinate system, essentially removing the influence of the third. Aligning the plane with two axis is synonymous with performing a rotation such that the surface normal becomes parallel with the third axis.

The matrix R that defines this transformation is given by two unit vectors in the plane as well as its surface normal. More specifically, knowing that $d_\alpha Q_{vd}$ is a vector in the plane, R can be defined as:

$$R \quad = \quad \begin{bmatrix} \dfrac{d_\alpha Q_{vd}}{\|d_\alpha Q_{vd}\|} & \left(n_x \times \dfrac{d\alpha Q_{vd}}{\|d\alpha Q_{vd}\|}\right) & n_x \end{bmatrix}^T \tag{54}$$

Noting that R is defined by the transpose of three orthogonal axis, it should be clear that R defines a rotation *from* the plane of projection and *into* the plane spanned by the first and second axis of the root coordinate system.

Then, using the position x of the view ray differential as the origin of rotation, the projected and rotated position x''_{pd} of the photon differential is given by:

$$x''_{pd} = R(x'_{pd} - x) + x \tag{55}$$

The projected and rotated footprint of the photon differential is easier to compute, since the differential position vectors can be interpreted as directions and therefore do not require translation:

$$d_\alpha Q''_{pd} = R d_\alpha Q'_{pd} \tag{56}$$
$$d_\beta Q''_{pd} = R d_\beta Q'_{pd} \tag{57}$$

Because the footprint of the view ray differential already lies in the plane of projection, the projected and rotated footprint of the view ray differential is obtained simply by rotating the original footprint:

$$d_\alpha Q''_{vd} = R d_\alpha Q_{vd} \tag{58}$$
$$d_\beta Q''_{vd} = R d_\beta Q_{vd} \tag{59}$$

Finally, since the rotation occurs around x, which is also a point in the plane of projection, the projected and rotated position x'' of the view ray differential is:

$$x'' = x \tag{60}$$

Notice how, by letting the footprint of the view ray differential define the plane of projection, R only needs to be computed once for each view ray differential. The same is true for the projected and rotated footprint of the view ray differential.

4.2.2 Computing the intersection

Given the projected and rotated footprints $(d_\alpha Q''_{vd}, d_\beta Q''_{vd})$ and $(d_\alpha Q''_{pd}, d_\beta Q''_{pd})$ and their offsets x'' and x''_{pd}, the next step is to compute the intersection between the inferred two-dimensional shapes, noting that the third axis can be disregarded since the rotated plane of projection is parallel to the plane spanned by the first and second axis of the root coordinate system.

The shape of a footprint can be interpreted in a variety of ways. For example, as mentioned in section 3.2.1, Schjøth et al. [8] describe the area of a footprint as the area of a the parallelogram spanned by its differential position vectors. Another interpretation is that of a skew ellipse, which is the two-dimensional representation of the skew ellipsoid used in the definition of filter space in the radiance estimate, see Equation 45 and Equation 46 in section 3.3.

How to compute the intersection between two shapes depends on how the shapes are defined. For example, the intersection between two parallelograms can be found by treating them as convex polygons and using a polygon clipping algorithm. However, to analytically compute the intersection between two skew ellipses is more complicated and a topic in its own right.

Instead of defining specialized methods for computing the intersection between specific types of shapes, observe that both types used by Schjøth et al. [8], the paralellogram and the skew ellipse, are convex. In general, under the assumption that the shapes of the two involved footprints are convex, the intersection between them can be approximated by discretizing the shapes into convex polygons and then computing the intersection between these convex polygons.

As such, the intersection computation is generalized to the intersection between convex polygons of any number of vertices, and it is the chosen discretization of the shapes of the footprints that define how these are interpreted. The following describes how to obtain the two-dimensional vertices of a convex polygon for either a parallelogram or a skew ellipse in the plane of projection.

Parallelogram Given a projected and rotated footprint $(d_\alpha Q'', d_\beta Q'')$, the $N = 4$ vertices of a convex polygon C describing a parallelogram in the plane of projection are given by:

$$\begin{aligned} p^1 &= d_\alpha Q'' + d_\beta Q'' \\ p^2 &= -d_\alpha Q'' + d_\beta Q'' \\ p^3 &= -d_\alpha Q'' - d_\beta Q'' \\ p^4 &= d_\alpha Q'' - d_\beta Q'' \end{aligned} \tag{61}$$

Skew ellipse Generating the N vertices $(p^1, \ldots, p^N)$ of a convex polygon C approximating a skew ellipse is possible by selecting N points on the unit circle with respect to the first two axis of the projected and rotated footprint $(d_\alpha Q'', d_\beta Q'')$. More formally, the i'th vertex is given by:

$$p^i = \begin{bmatrix} d_\alpha Q''_1 & d_\beta Q''_1 \\ d_\alpha Q''_2 & d_\beta Q''_2 \end{bmatrix} \begin{bmatrix} \cos\left((i-1)\frac{2\pi}{N}\right) \\ \sin\left((i-1)\frac{2\pi}{N}\right) \end{bmatrix} \tag{62}$$

where $i \in [1, 2, \ldots, N]$. Note that, by selecting the vertices such that they have uniform distribution on the unit circle, the resolution of the resulting ellipse is greatest in regions with high curvature.

A variety of methods exist for computing the intersection between two convex polygons. One such method is the classic polygon clipping algorithm by Sutherland and Hodgman [10], which, given two polygons, one describing a convex clip frame and one describing the subject, yields the vertices of a third polygon which describes the subject within the clip frame. For two convex polygons, the clip frame and the subject are interchangeable, and the result is always a third convex polygon describing the intersection between them. The Sutherland-Hodgman algorithm is described in further detail by Foley et al. [2].

Using C_{vd} and C_{pd} to denote the convex polygons describing the footprints of the view ray differential and the photon differential, respectively, the intersection between them can be computed by applying the Sutherland-Hodgman algorithm with the parameters in any order. The result is a third convex polygon $C_{vd \cap pd}$ that describes the two-dimensional intersection between the two footprints in the rotated plane of projection.

4.2.3 Coverage and centroid

Using p^i to denote the i'th vertex of a convex planar polygon C, the area of the polygon C can be found by using the formula for the area of a planar polygon:

$$A_C = \left| \frac{1}{2} \sum_{i=1}^{N-1} p_1^i p_2^{i+1} - p_1^{i+1} p_2^i \right| \tag{63}$$

Given the convex polygons that describe the footprint of the view ray differential as well as its intersection with the photon differential, C_{vd} and $C_{vd \cap pd}$, the *coverage* of the photon differential, i.e. the fraction that the photon differential covers of the view ray differential, can be expressed as follows:

$$w_{vd \cap pd} = \frac{A_{C_{vd \cap pd}}}{A_{C_{vd}}} \tag{64}$$

The coverage of the photon differential can be employed directly as a weight in the radiance estimate to handle scenarios like the one shown in Figure 10(c) on page 23, where the view ray differential encompasses an area larger than its intersection with the photon differential, and therefore the photon differential only contributes to a fraction of the beam represented by the view ray differential.

Another useful property is the centroid of the intersection in the rotated plane of projection. This is a point which can be obtained by averaging the positions of the vertices of the convex polygon $C_{vd \cap pd}$ describing the intersection, i.e.:

$$x_{C_{vd \cap pd}} = \frac{1}{N} \sum_{i=1}^{N} p^i \tag{65}$$

The centroid in the rotated plane of projection can be used to approximate the centroid of the actual three-dimensional intersection between the two footprints. In turn, the centroid of the three-dimensional intersection can be used to handle the scenario shown in Figure 10(b) on page 23, where the sampling point x lies outside the photon differential's range of influence while the centroid of the intersection does not.

To approximate the centroid of the three-dimensional intersection, first express the centroid $x_{C_{vd \cap pd}}$ of the two-dimensional intersection as a linear combination of the first two axis of the projected and rotated footprint of the view ray differential, making sure the third component is zero, and then treat it as a linear combination of the three axis of the original footprint of the view ray differential to obtain a point in the root coordinate system. More formally, the centroid of the three-dimensional intersection can be approximated as follows:

$$\begin{aligned} x_{vd \cap pd} \approx\ & x + \begin{bmatrix} d_\alpha Q_{vd} & d_\beta Q_{vd} & n_x \end{bmatrix} \\ & \begin{bmatrix} \begin{bmatrix} (d_\alpha Q''_{vd})_1 & (d_\beta Q''_{vd})_1 \\ (d_\alpha Q''_{vd})_2 & (d_\beta Q''_{vd})_2 \end{bmatrix}^{-1} & \begin{matrix} 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & \quad 0 \end{matrix} & 0 \end{bmatrix} \\ & \left(x_{C_{vd \cap pd}} - x'' \right) \end{aligned} \tag{66}$$

By replacing the sampling point x with $x_{vd\cap pd}$ in the input to the filter kernel K, the radiance estimate will effectively weigh each photon differential by its filter space distance to the centroid of the intersection with the view ray differential, rather than just the filter space distance to the sampling point x, which may lie outside the photon differential's range of influence, as shown in Figure 10(b).

Notice how one could also have chosen to approximate $x_{vd\cap pd}$ by expressing $x_{C_{vd\cap pd}}$ with respect to the footprint of the photon differential. However, because the two-dimensional intersection is just an approximation based on the projection of either footprint along ω, this could potentially result in a point outside the footprint of the view ray differential. Conversely, by choosing to express $x_{C_{vd\cap pd}}$ with respect to the footprint of the view ray differential, the approximation of the three-dimensional intersection may not always fall within range of the photon differential, but in worst case this just results in zero contribution, which is no worse than if the sampling point x is outside the range of influence.

4.2.4 Computing the reflected radiance

Using the definitions of coverage $w_{vd\cap pd}$ and approximate centroid of intersection $x_{vd\cap pd}$ from the previous section, the filtered radiance estimate defined by Schjøth et al. [8], see section 3.3, can be refined as follows:

$$L_r(x,\omega) \approx \sum_{pd=1}^{k} f_r(x,\omega_{pd},\omega)\frac{\Phi_{pd}}{A_{pd}}K(\|M_{pd}(x_{vd\cap pd}-x_{pd})\|)w_{vd\cap pd} \quad (67)$$

In the above definition of the radiance estimate, each photon differential is weighed according to the filter space distance to the approximate centroid $x_{vd\cap pd}$ of the intersection between the involved footprints, rather than according to the filter space distance to the sampling point x. Also, the filtered contribution of a photon differential is scaled by the fraction that the intersection covers of the view ray differential. These refinements handle the scenarios illustrated in Figure 10(b) and Figure 10(c), respectively.

Because both the coverage and the centroid are properties derived from the coplanar intersection between the involved footprints, I refer to this approach as *coplanar intersection-weighted photon differentials.*

4.3 Multi-sampled photon differentials

Section 4.2 described an approach, in which the intersection between the involved footprints was computed explicitly, using the coverage and centroid of the intersection to refine the radiance estimate. However, it is also possible to take into account the intersection implicitly.

This section describes a different approach, in which the footprint of the view ray differential is used to define a set of sampling points used to multi-sample the photon differential, thereby implicitly taking into account the intersection between them. This approach can also be regarded as a direct approximation of simply tracing more view rays per pixel.

4.3.1 Generating the sampling points

Given the footprint of the view ray differential, any point in the same plane can be expressed as a linear combination of the differential position vectors $d_\alpha Q_{vd}$ and $d_\beta Q_{vd}$, and the surface normal n_x of the tangential surface in the intersection point x.

Using s and t to describe the offsets along $d_\alpha Q_{vd}$ and $d_\beta Q_{vd}$, respectively, the definition of a sampling point for the current view ray differential can be written:

$$x_{s,t} \quad = \quad x + \begin{bmatrix} d_\alpha Q_{vd} & d_\beta Q_{vd} & n_x \end{bmatrix} \begin{bmatrix} s \\ t \\ 0 \end{bmatrix} \tag{68}$$

The offsets s and t for the generated set of points can be chosen in infinitely many ways, but in general they should be chosen such that the set of points approximates the shape of the footprint of the view ray differential. Obviously, this depends on how the shape of the footprint of the view ray differential is interpreted, as well as the desired number of sampling points. For simplicity, the remainder of this section will interpret the shape of the footprint of the view ray differential as that of a parallelogram.

Using a parallelogram to define the shape of a footprint, the offsets for the set of sampling points can be defined using an $N \times N$ grid, where N is the desired number of indices along each side of the parallelogram. Specifically, the offsets s and t specified for grid element i, j can be defined as:

$$s_{i,j} = \quad = \quad -1 + (i-1)\left(\frac{2}{N-1}\right) \tag{69}$$

$$t_{i,j} = \quad = \quad -1 + (j-1)\left(\frac{2}{N-1}\right) \tag{70}$$

where $i, j \in [1 \ldots N]$ and $N \geq 2$. By iterating over the elements of the grid and applying the offsets to Equation 68, the result is a set of sampling points that outline the parallelogram, also filling its interior if $N \geq 3$. Figure 13 illustrates the sample distribution using a 3×3 grid.

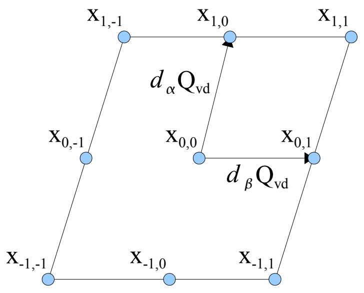


Figure 13: The distribution of the generated sampling points using a 3×3 grid for the parallelogram spanned by $d_\alpha Q_{vd}$ and $d_\alpha Q_{vd}$. In this example, the set X of sampling points is given by $\{ x_{-1,-1}, x_{-1,0}, x_{-1,1}, x_{0,-1}, x_{0,0}, x_{0,1}, x_{1,-1}, x_{1,0}, x_{1,1} \}$.

Note that a slight inflexibility of using a grid-based distribution is the number of generated sampling points, which is always N squared. Since $N \geq 2$, this means that the minimum number of sampling points is 4. This not an issue in practice, however, since at least 4 sampling points are necessary regardless, in order to define a symmetric distribution that represents more than just a straight line.

4.3.2 Computing the reflected radiance

Given a set X of sampling points distributed in the plane spanned by the footprint of the view ray differential, the filtered radiance estimate defined by Schjøth et al. [8], see section 3.3, can be refined as follows:

$$L_r(x,\omega) \approx \sum_{pd=1}^{k} f_r(x,\omega_{pd},\omega)\frac{\Phi_{pd}}{A_{pd}}\left(\frac{1}{N^2}\sum_{i=1}^{N^2} K(\|M_{pd}(X_i - x_{pd})\|)\right) \quad (71)$$

where N^2 is the number of sampling points in the set, and where X_i denotes the i'th sampling point from the set.

In the above definition of the radiance estimate, each photon differential is weighed by multi-sampling the filter kernel K, one sample for each of the sampling points in the set X, using the average of the results to determine the contribution with respect to the sample distribution.

Because the sample distribution approximates the shape of the footprint of the view ray differential, and because the filter space transformation M_{pd} is based on the footprint of the photon differential, see Figure 14, the above definition of the radiance estimate implicitly takes into account the intersection between the involved footprints, and inherently handles the two scenarios shown in Figure 10(b) and Figure 10(c), both on page 23.

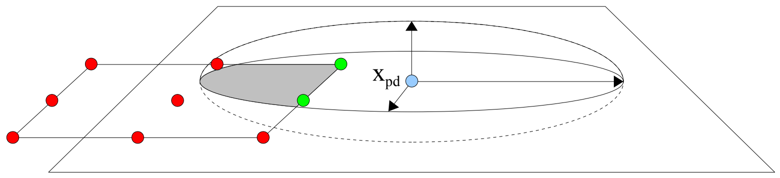


Figure 14: Illustration of the intersection between a set of sampling points and the skew ellipsoid that represents the world space boundaries of filter space for a given photon differential. The filter space transformation M_{pd} is based on the footprint of the photon differential, and a more formal definition is given in section 3.3. In this example, the red dots denote sampling points that lie outside the photon differential's range of influence, which therefore receive zero contribution, while the green dots denote sampling points that lie inside the range of influence. Note that the sampling points do *not* have to be coplanar with the footprint of the photon differential, and that this is simply the case in this example for the simplicity of the illustration.

Finally, because the filter kernel K is multi-sampled in filter space, and because filter space is defined individually for each photon differential, I refer to this approach as *multi-sampled photon differentials*.

In section 5, both multi-sampled photon differentials and coplanar intersection-weighted photon differentials will be evaluated in comparison to the filtered radiance estimate, which both approaches strive to improve.

5 Implementation and evaluation

Section 4 explores and defines two different approaches to incorporating the view ray differential in the radiance estimate, namely *coplanar intersection-weighted photon differentials* and *multi-sampled photon differentials*.

Both methods were defined based on the hypothesis that, by taking into account the view ray differential, it should be possible to surpass the accuracy of the regular filtered radiance estimate, defined by Schjøth et al. [8] and described in section 3.3, without increasing the number of view rays per pixel, the resolution of the photon map, or the number of gathered photon differentials in each point of intersection.

In this section, a number of comparisons are made to determine whether the above hypothesis holds true for either of the two new methods. Specifically, a number of comparisons are made between images produced using the regular filtered radiance estimate and images produced using each of the two new methods, in order to determine if either of the two new methods produce images that represent a more accurate result.

To avoid relying on a subjective measure for the comparisons, it is necessary to define what constitutes a more accurate – a better – result.

As discussed in section 4, the accuracy of the regular filtered radiance estimate increases with the number of view rays per pixel. Thus, with the baseline being the regular filtered radiance estimate with one view ray per pixel, a better result can be defined as an image that approximates the image produced using the regular filtered radiance estimate with *more than one* view ray per pixel.

Notice how this measure of success requires that the chosen test case clearly shows the discrepancy between the images produced using the regular filtered radiance estimate with one and more than one view ray per pixel, respectively.

With the purpose of being able to apply the new methods and the regular filtered radiance estimate to the exact same test case, using the same parameters for the photon map as well as the same distribution for the stored photon differentials, I chose to implement the two new methods in an existing rendering framework, which already contains a working reference implementation of photon mapping with photon differentials written by Jeppe Revall Frisvad, who is one of the co-authors of the paper on photon differentials by Schjøth et al. [8].

The framework itself is written in C++ and consists of several classes that define camera, scene graph, loading a scene from disk, ray tracing operations, building the photon map, finding the k-nearest photons, and so forth. My implementation consists of the following additions to the framework:

- A new rendering class, which associates each view ray with a ray differential, and which parameterizes the radiance estimate by exposing it as a function parameter.
- Two functors that implement the methods described in section 4, each of which can then be passed as an argument to the new rendering class.

- Because ray propagation during the rendering stage is handled by the shaders for the surface materials along a given path, my implementation also extends a selection of the shaders in the framework to support the propagation of a ray differential alongside the original view ray.

The source code to the implementation can be found in Appendix A.

Prior to defining the test case and carrying out the actual comparisons, it is important to note that the rendering framework also implements the practical optimizations discussed in section 2.2.2.

Most importantly, during the photon tracing stage the photon differentials are stored in two seperate photon maps. One of them is the caustic photon map, which stores caustic photon differentials, and the other is the global photon map, which stores the rest. Caustic photon differentials are photon differentials whose first intersection with the scene was with a specular surface.

Furthermore, during rendering it is possible to select a shader that results in only the caustic photon differentials being considered in the radiance estimate. The result is an image in which only the caustic photon differentials apply light to the surfaces in the scene. Because caustics have a tendency to form well defined contours, considering only the caustic photon differentials results in high contrast between lit and unlit surfaces in the resulting image.

This is quite useful in terms of the evaluation, since it makes differences in the results easier to discern. For this reason, I chose to base my comparisons on renders where only the caustic photon differentials are taken into consideration in the radiance estimate.

To summarize the plan for determining whether either of the two new methods produce better results than the regular filtered radiance estimate:

- Comparisons will be made between a number of images based on a common test case.
- Specifically, the images produced using the two new methods will be compared to the images produced using the regular filtered radiance estimate.
- The measure of success is the image produced using the regular filtered radiance estimate with more than one view ray per pixel.

The performance of the two new methods is evaluated in section 5.2. Using the rendering times obtained when generating the images for the test case, the performance evaluation provides the basis for a discussion of whether either of the two new methods are usable in practice.

5.1 Image comparisons

For the common test case I chose a scene which consists of a reflective metal ring placed on a planar diffuse surface. A render of this scene is shown in Figure 15 on page 35. Using 100000 and 20000 as the thresholds for emission of global and caustic photon differentials, respectively, 100036 photons differentials were stored in the global photon map, while 4978 photon differentials were stored in the caustic photon map.

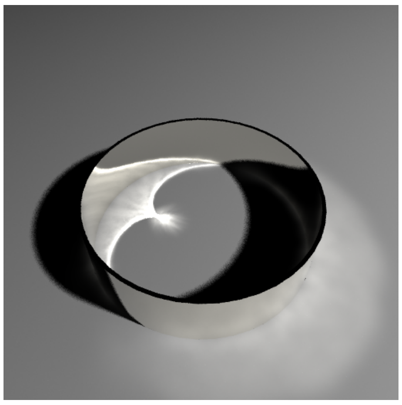


Figure 15: A render of the scene that constitutes the common test case. The ring is a specular surface which reflects the incoming light onto the planar diffuse surface beneath it, thus forming a caustic that is visible both in the planar surface beneath the ring, and in the reflective surface of the inner band of the ring.

As shown in Figure 15, the chosen scene exhibits a caustic with a clearly defined contour. A clearly defined contour is prone to undersampling artefacts as a result of tracing too few view rays per pixel, or not taking into account the view ray differential, and this is what makes this particular scene a good test case. Note that the caustic photon differentials are all located on the planar surface, but because the inner band of the ring is reflective, and because the view rays are reflected onto the planar surface, the caustic is also visible in the inner band of the ring.

In the comparisons that follow, the regular filtered radiance estimate is referred to as the regular method. I chose the regular method with 3×3 view rays per pixel as the measure of success. By the end of this section, it should be clear that this is an acceptable choice, due to the visible discrepancy between the images produced using the regular method with one and 3×3 view rays per pixel, respectively.

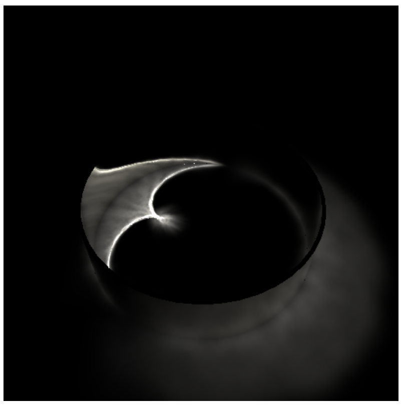
The two new methods use just one view ray per pixel, but on the other hand they depend on the initial configuration of the view ray differential. The initialization of the view ray differential such that it overlaps a single pixel on the viewport is described in section 4.1, and I use this configuration in the comparison.

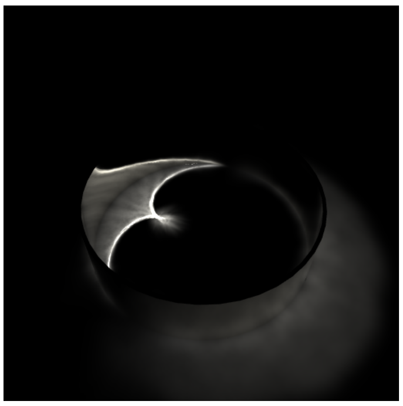
For coplanar intersection-weighted photon differentials, there are no additional parameters. For multi-sampled photon differentials, however, the results will depend on the number of generated sampling points. As described in section 4.3.1, the number of generated sampling points is given by $N \times N$, where N is a parameter of the method. To avoid undersampling, I chose a relatively high number of sampling points, i.e. $N = 8$.

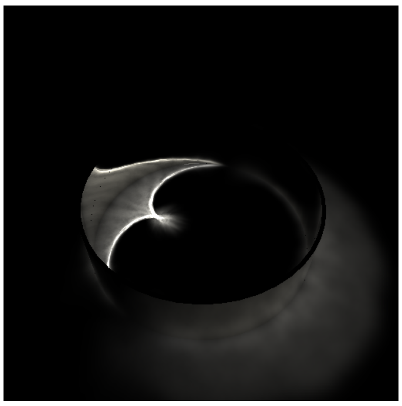
With the method-specific parameters in place, the four different methods to be used in the comparison are: The regular method, the regular method with 3×3 view rays per pixel, the coplanar intersection-weighted method, and the multi-sampled method with 8×8 sampling points.

To make it easier to discern the differences between the resulting images, the scene is rendered using only the photon differentials from the caustic photon map as input to each method.

Figure 16 shows the images produced by each of the four methods when rendering the scene using only the caustic photon differentials. For each method, the maximum number of photon differentials to be considered in each point of intersection is set to 50, i.e. $k = 50$.


(a) Regular, 1×1 view rays/pixel


(b) Regular, 3×3 view rays/pixel


(c) Coplanar intersection-weighted

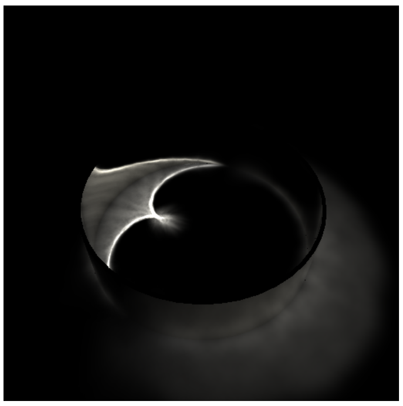
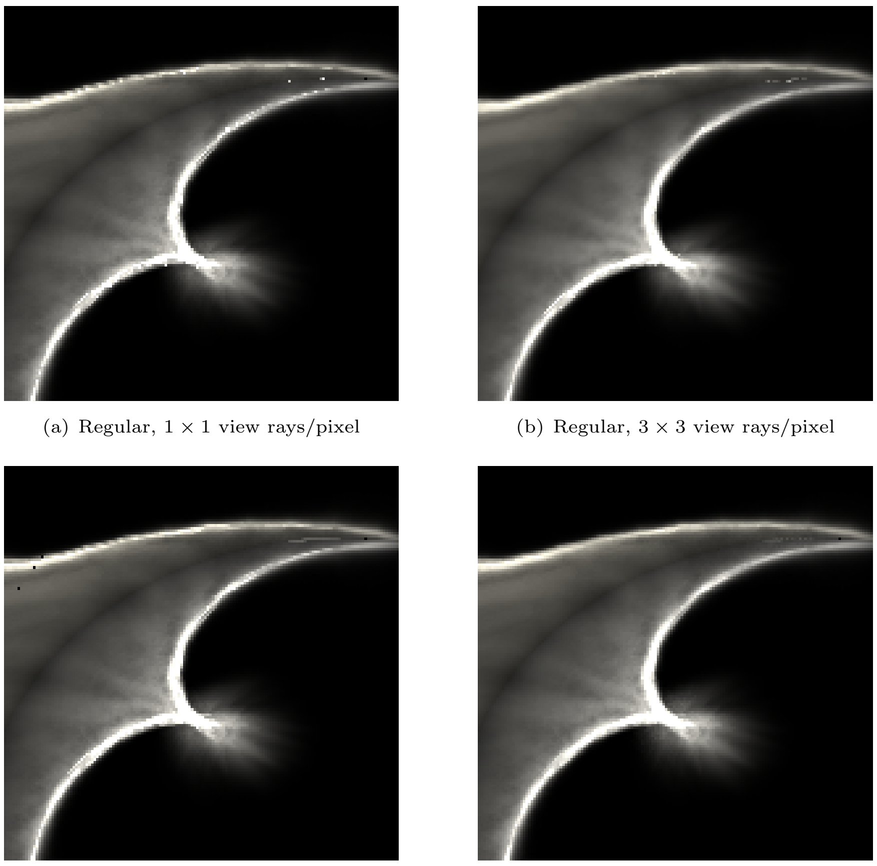

(d) Multi-sampled, 8×8 sampling points

Figure 16: The images produced using each of the four methods when considering only the caustic photon differentials. Without magnification it can be difficult to identify the differences between them. The most notable differences appear in the clearly defined contours of the caustic, and Figure 17 on page 37 illustrates these more clearly by enlarging the region containing these contours.

At a first glance, the resulting images in Figure 16 appear to be mostly identical. However, this is not true, but some magnification is necessary in order to visually identify the differences between them.

With the purpose of illustrating the differences more clearly, knowing that the most notable differences appear in the contours of the caustic that is visible in and near the inner band of the ring, I enlarge the results and crop them to the region which contains just these contours of the caustic. The enlarged and cropped results are shown in Figure 17 on page 37.


(a) Regular, 1 × 1 view rays/pixel

(b) Regular, 3 × 3 view rays/pixel

(c) Coplanar intersection-weighted

(d) Multi-sampled, 8 × 8 sampling points

Figure 17: Enlarged and cropped results based on the images shown in Figure 16 on page 36. These images illustrate the quality of the clearly defined contours of the caustic as rendered using each of the four methods.

From the results in Figure 17 it is possible to make several observations:

1. First of all, the image produced by the regular method with one view ray per pixel, see Figure 17(a), exhibits noticeable undersampling artefacts along the contour of the caustic. This is expected, and underlines the need to either trace more view rays per pixel or take into account the view ray differential.

2. When comparing the images produced using the regular method with one and 3×3 view rays per pixel, Figure 17(a) and Figure 17(b), it is immediately apparent that the regular method with 3×3 view rays per pixel produces a smoother contour with much fewer aliasing artefacts. This is also expected, and validates the choice of treating the regular method using 3×3 view rays per pixel as the measure of success.

3. Comparing the image produced using the coplanar intersection-weighted method, Figure 17(c), to the images produced using both instances of the regular method, Figure 17(a) and Figure 17(b), it should be quite

clear, based on the contour of the caustic, that the coplanar intersection-weighted method results in at least some improvement over the regular method.

4. More specifically, using the coplanar intersection-weighted method, many of the undersampling artefacts exhibited by the regular method with one view ray per pixel, see Figure 17(a), are eliminated, and as a result the contour itself is more continuous, thus approximating the smooth contour of the regular method with 3×3 view rays per pixel, Figure 17(b).

5. Unfortunately, the coplanar intersection-weighted method also introduces some new artefacts. Looking at the left region of Figure 17(c), notice the three black dots near the upper part of the caustic. These specific artefacts are a result of not being able to compute the inverse of the submatrix in Equation 66 in section 4.2, thereby causing the method to abort. In the framework used for my implementation, computing the inverse of a 2×2 matrix relies on the determinant, and when the determinant is too small, then the function that computes the inverse throws an error. In other words, when the area of the footprint of the view ray differential is too small, then the framework does not allow the computation of the inverse of the submatrix in Equation 66 in section 4.2.

6. Comparing the image produced using the multi-sampled method, Figure 17(d), to the images produced using both instances of the regular method, Figure 17(a) and Figure 17(b), observe how the multi-sampled method, quite simply, produces a result which is completely superior to that of the regular method with one view ray per pixel.

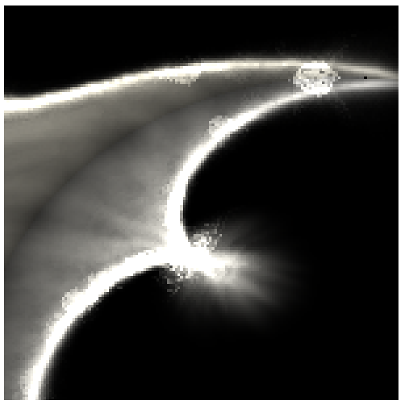
7. In fact, the contour in Figure 17(d) is even smoother and more continuous than the contour in Figure 17(b), while still being equally well-defined. This implies that, in terms of rendering caustics, the multi-sampled method actually produces *better* results than the regular method with 3×3 view rays per pixel. In other words, the multi-sampled method both reaches and *surpasses* the measure of success for this evaluation.

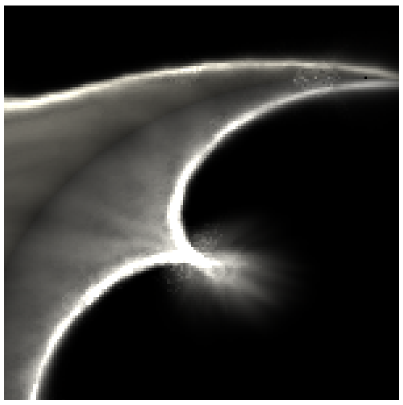
Based on these observations, I can conclude that both new methods, namely coplanar intersection-weighted photon differentials and multi-sampled photon differentials, are capable of producing better results than the regular method with one view ray per pixel. This is because they both approximate the results produced by the regular method with 3×3 view rays per pixel.

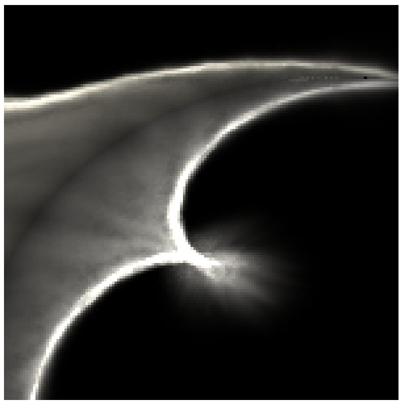
For coplanar intersection-weighted photon differentials, note that the results can only be considered marginally better based on the current implementation. As mentioned in observation 5, this is due to the fact that a few new artefacts are introduced in the resulting image, simply because the matrix library in the rendering framework does not allow computing the inverse of 2×2 matrices with a very small determinant. This limitation is obviously meant as a precaution against numerical error, which is a valid concern. In principle, however, the problem can be fixed by replacing the built-in function for the 2×2 inverse.

For multi-sampled photon differentials with 8×8 sampling points, the results are very good, and as mentioned in observation 7, this method actually produces better results than the regular method with 3×3 view rays per pixel.

However, when using multi-sampled photon differentials, one should also note that choosing a relatively high number of sampling points is essential to obtaining stable results. The effects of varying the number of sampling points are shown in Figure 18.


(a) Multi-sampled, 2×2 sampling points


(b) Multi-sampled, 4×4 sampling points


(c) Multi-sampled, 8×8 sampling points

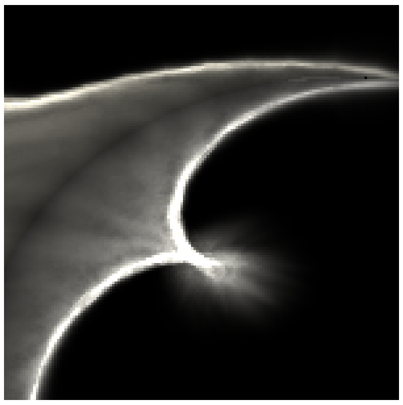

(d) Multi-sampled, 16×16 sampling points

Figure 18: The results produced using multi-sampled photon differntials for different choices N, i.e. varying the number of sampling points. Notice the artefacts in (a) and (b), where $N < 8$, and how the result changes very little in the jump from $N = 8$ (c) to $N = 16$ (d).

To understand why artefacts appear in certain regions when the number of sampling points is low, recall from section 3.3 that the raw irradiance of a photon differential is its radiant power divided by the area of its footprint. As such, when sampling a photon differential whose area is very small, the resulting contribution has the potential to be very large, depending on the filter space distance to the sampling point.

With multi-sampled photon differentials, the contribution of a photon differential is always an average of the contribution measured for each of the $N \times N$ sampling points. If the contribution for one sampling point is very large, and the number of sampling points is relatively low, then the contribution will still be noticable even after averaging the results. As shown in Figure 18, the method becomes stable at $N = 8$ as used in the evaluation.

5.2 Performance discussion

To evaluate the performance of the two new methods in comparison to both configurations of the regular filtered radiance estimate, I compare the rendering times obtained for each of the four methods when rendering the images shown in Figure 16 on page 36. Table 1 shows the results.

Method	Rendering time in seconds
Regular, 1×1 view rays/pixel	20.637
Regular, 3×3 view rays/pixel	183.255
Coplanar intersection-weighted	380.045
Multi-sampled, 8×8 sampling points	63.671

Table 1: Time in seconds for the four methods to render the images shown in Figure 16 on page 36.

From Table 1 it is easy to make the following observations:

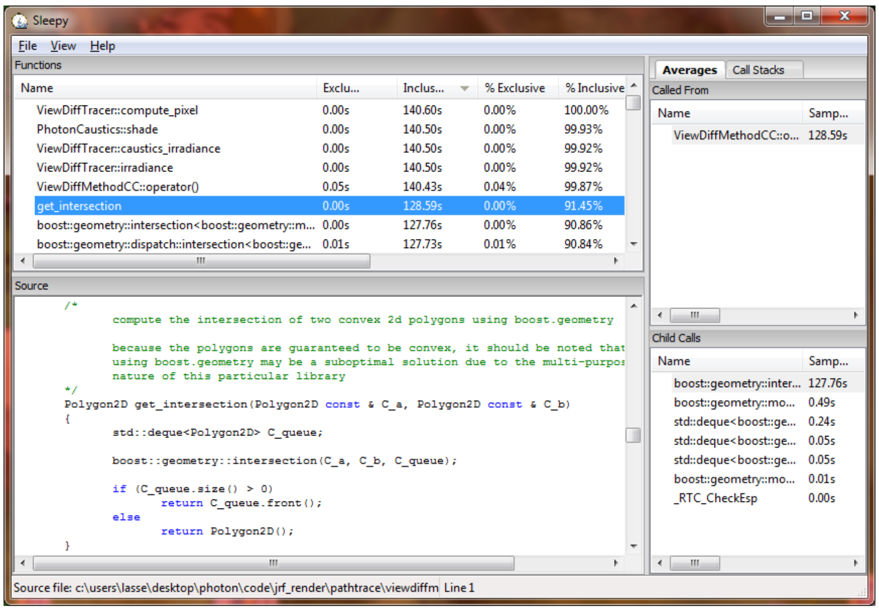
1. Increasing the number of view rays per pixel results in a nearly linear increase in rendering time.
2. Using coplanar intersection-weighted photon differentials, the rendering time is about 19 times longer than when using the regular filtered radiance estimate with one view ray per pixel, and more than 2 times longer than when using the filtered radiance estimate with 3×3 view rays per pixel.
3. Using multi-sampled photon differentials with 8×8 sampling points, the rendering time is about 3 times longer than when using the regular filter radiance estimate with one view ray per pixel. However, the rendering time is also about 3 times shorter than when using the regular filtered radiance estimate with 3×3 times per pixel.

For coplanar intersection-weighted photon differentials, the long rendering time, combined with the results of the image comparison in section 5.1, is reason enough to disqualify this particular method for practical purposes, at least in its current implementation. More specifically, because the method does not rival the accuracy of the regular filtered radiance estimate with 3×3 view rays per pixel, although it is an acceptable approximation, and because it is slower, there is no reason to actually use it in practice, unless the rendering time can be more than halved through optimizations.

To identify what causes the huge gap in performance for coplanar intersection-weighted photon differentials, I examined the code using a freeware profiler known as *Sleepy*[1]. Essentially, Sleepy identifies how much time is spent in each part of the code based on a number of process samples taken over time. Figure 19 on page 41 shows the result of profiling the rendering process while rendering an image using coplanar intersection-weighted photon differentials.

Using my current implementation of coplanar intersection-weighted photon differentials, it is clear from from Figure 19 that approximately 91.5% of the

[1]http://www.codersnotes.com/sleepy



Functions

Name	Exclu...	Inclus...	% Exclusive	% Inclusive
ViewDiffTracer::compute_pixel	0.00s	140.60s	0.00%	100.00%
PhotonCaustics::shade	0.00s	140.50s	0.00%	99.93%
ViewDiffTracer::caustics_irradiance	0.00s	140.50s	0.00%	99.92%
ViewDiffTracer::irradiance	0.00s	140.50s	0.00%	99.92%
ViewDiffMethodCC::operator()	0.05s	140.43s	0.04%	99.87%
get_intersection	0.00s	128.59s	0.00%	91.45%
boost::geometry::intersection<boost::geometry::m...	0.00s	127.76s	0.00%	90.86%
boost::geometry::dispatch::intersection<boost::ge...	0.01s	127.73s	0.01%	90.84%

Called From

Name	Samp...
ViewDiffMethodCC::o...	128.59s

Child Calls

Name	Samp...
boost::geometry::inter...	127.76s
boost::geometry::mo...	0.49s
std::deque<boost::ge...	0.24s
std::deque<boost::ge...	0.05s
std::deque<boost::ge...	0.05s
boost::geometry::mo...	0.01s
_RTC_CheckEsp	0.00s

Source

```
/*
        compute the intersection of two convex 2d polygons using boost.geometry

        because the polygons are guaranteed to be convex, it should be noted that
        using boost.geometry may be a suboptimal solution due to the multi-purpos
        nature of this particular library
*/
Polygon2D get_intersection(Polygon2D const & C_a, Polygon2D const & C_b)
{
        std::deque<Polygon2D> C_queue;

        boost::geometry::intersection(C_a, C_b, C_queue);

        if (C_queue.size() > 0)
                return C_queue.front();
        else
                return Polygon2D();
}
```

Source file: c:\users\lasse\desktop\photon\code\jrf_render\pathtrace\viewdiffm Line 1

Figure 19: The result of using Sleepy to profile the rendering process during rendering using coplanar intersection-weighted photon differentials. Notice how approximately 91.5% of the rendering time is spent in `get_intersection`.

time is spent in the function `get_intersection`, which returns the intersection between the convex polygons describing the involved footprints. In turn, `get_intersection` relies on the intersection function in the boost geometry library to compute the intersection.

Considering the fact that boost geometry library supports more than just convex polygons, it is possible that the use of their intersection function adds a significant amount of overhead in comparison to a case-specific implementation of Sutherland-Hodgman, for example. Unfortunately, it is not possible to estimate the potential gain of replacing the intersection function of boost geometry with a different algorithm, since the documentation of this function does not specify which underlying algorithm is used for a particular set of inputs.

For multi-sampled photon differentials, however, the rendering times are very acceptable. With 8×8 sampling points, recall from section 5.1 that multi-sampled photon differentials actually surpasses the results produced using the regular radiance estimate with 3×3 view rays per pixel. Thus, considering the rendering times in Table 1 on page 40, multi-sampled photon differentials provide a real and practical alternative to tracing more view rays per pixel.

Though not strictly necessary considering the results, it should also be possible to optimize the implementation of multi-sampled photon differentials. Specifically, instead of applying the filter space transformation M_{pd} to the translation of each sampling point such as done in Equation 71, M_{pd} could enter the definition of a sampling point, see Equation 68, in order to express the sampling points directly with respect to the footprint of the view ray differential in filter space. This would result in not having having to perform a matrix multiplication for each sampling point, potentially increasing the speed of the method.

6 Conclusion

In the introductory chapters of this report, I have introduced the basics of photon mapping, going into moderate detail with both stages of the basic algorithm, as well as described the concepts and theory of photon mapping with photon differentials, including the general theory of ray diffentials. Regarding the theory of ray differentials, I have also derived the necessary operations for tracing a ray differential alongside a ray that propagates through the scene.

Based on the hypothesis that the accuracy of the radiance estimate can be improved by also taking into account the footprint of the view ray differential, I have explored two different approaches to incorporating the footprint of the view ray differential in the filtered radiance estimate, defined by Schjøth et al [8], for photon mapping with photon differentials.

This has led to two new methods for computing the reflected radiance, one which explicitly computes the intersection between the coplanar footprints, and one which uses the footprint of the view ray differential to multi-sample the photon differential. I have named these methods *coplanar intersection-weighted photon differentials* and *multi-sampled photon differentials*, respectively.

In order to determine whether the hypothesis holds true for either of the two new methods, I implemented both of them in an existing rendering framework, which already provided an implementation of photon mapping with photon differentials, including the regular filtered radiance estimate. Using this framework, I applied each method to a common test case and then compared the results.

By letting the results of the regular filtered radiance estimate with 3×3 view rays per pixel define the direction of the measure of success, I found that both new methods produce more accurate results than the filtered radiance estimate with one view ray per pixel, and that one of them is superior.

Using coplanar intersection-weighted photon differentials, the results are slightly better than the regular filtered radiance estimate, but unfortunately the required rendering time is also much too high in comparison. By profiling the method, I found that the culprit of the low performance is the intersection computation, which accounts for approximately 91.5% of the rendering time. As a result, this method is not very useful in practice without further optimization.

Using multi-sampled photon differentials, the results are much better than those of the regular filtered radiance estimate. In fact, with 8×8 sampling points, the resulting images clearly rival the images produced using the regular filtered radiance estimate with 3×3 view rays per pixel. In terms of performance, the method requires more time to render an image than the regular filtered radiance estimate with one view ray per pixel. However, even with 8×8 sampling points, it is three times faster than the regular filtered radiance estimate with 3×3 view rays per pixel.

I can conclude that, while coplanar intersection-weighted photon differentials falls a bit short, *multi-sampled photon differentials* present a practical and relatively fast alternative to tracing more view rays per pixel. With 8×8 sampling points, the results of the method clearly rival those of the regular filtered radiance estimate with 3×3 view rays per pixel, and it is also three times faster.

References

[1] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974. 10.1007/BF00288933.

[2] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[3] H. Igehy. Tracing ray differentials. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 179–186, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[4] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[5] H. W. Jensen. *Realistic image synthesis using photon mapping.* A. K. Peters, Ltd., Natick, MA, USA, 2001.

[6] J. T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[7] J. R. Magnus and H. Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics.* John Wiley & Sons, 1988.

[8] L. Schjøth, J. R. Frisvad, K. Erleben, and J. Sporring. Photon differentials. In *Proceedings of GRAPHITE 2007*, December 2007.

[9] J. Sporring, L. Schjøth, and K. Erleben. Spatial and temporal ray differentials. Technical report, Dept. of Computer Science, University of Copenhagen, 2009.

[10] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17:32–42, January 1974.

[11] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23:343–349, June 1980.

A Source code

The files `ViewDiffTracer.h` through `ViewDiffMethodMS.cpp` contain the new classes that define the view ray differential enabled renderer, as well as the two methods described in section 4. The rest of the files, starting with `Shader.h`, contain shader classes, which have been extended to support the propagation of a ray differential along with the original view ray.

`ViewDiffTracer.h` 44

`ViewDiffTracer.cpp` 45

`ViewDiffMethod.h` 46

`ViewDiffMethodCC.h` 47

`ViewDiffMethodCC.cpp` 47

`ViewDiffMethodMS.h` 51

`ViewDiffMethodMS.cpp` 51

`Shader.h` 52

`PhotonCaustics.h` 53

`PhotonCaustics.cpp` 53

`PhotonLambertian.h` 54

`PhotonLambertian.cpp` 54

`Transparent.h` 56

`Transparent.cpp` 56

`Mirror.h` 57

`Mirror.cpp` 58

`Metal.h` 58

`Metal.cpp` 59

ViewDiffTracer.h

```cpp
/* ViewDiffTracer.h

    Lasse Jon Fuglsang Pedersen <fuglsang@diku.dk>
    14-10-2011

*/

#ifndef __VIEWDIFFTRACER_H__
#define __VIEWDIFFTRACER_H__

#include "DifferentialTracer.h"
#include "ViewDiffMethod.h"

class ViewDiffTracer : public DifferentialTracer
{
private:
    ViewDiffMethod const & f;

public:
    ViewDiffTracer(
        unsigned int w, unsigned int h, Scene * s,
        unsigned int max_no_of_particles, float diff_smoothing = 10.0f,
        unsigned int pixel_subdivs = 1,
        ViewDiffMethod const & f = ViewDiffMethod())
        :
    DifferentialTracer(w, h, s, max_no_of_particles, diff_smoothing,
        pixel_subdivs), f(f)
    {
    }

    /*
        modified version of compute_pixel, which associates each view ray with
        a ray differential, passing both to the selected shader for the first
        intersection, which then takes care of any further propagation
        (based on lines 165 through 189 of ParticleTracer.cpp)
    */
    CGLA::Vec3f compute_pixel(unsigned int x, unsigned int y) const;

    /*
        needed for tracing ray differentials through diffuse reflections
        (based on lines 331 through 337 of DifferentialTracer.cpp)
    */
    bool trace_cosine_weighted(Geometry::Ray const & in, Geometry::Ray & out,
        RayDiff & dr) const;

    /*
        estimate irradiance of photon differentials in given photon map
    */
    CGLA::Vec3f irradiance(PhotonDiffMap const & map, Geometry::Ray const & r,
        RayDiff const & dr, float max_distance, int no_of_particles) const;

    /*
        estimate irradiance of caustic photon differentials
    */
    CGLA::Vec3f caustics_irradiance(Geometry::Ray const & r,
        RayDiff const & dr, float max_distance, int no_of_particles) const;

    /*
        estimate irradiance of global photon differentials
    */
    CGLA::Vec3f global_irradiance(Geometry::Ray const & r,
        RayDiff const & dr, float max_distance, int no_of_particles) const;
};

#endif //__VIEWDIFFTRACER_H__
```

ViewDiffTracer.cpp

```
/* ViewDiffTracer.cpp

    Lasse Jon Fuglsang Pedersen <fuglsang@diku.dk>
    14-10-2011

*/

#include "ViewDiffTracer.h"
#include "sampler.h"

CGLA::Vec3f ViewDiffTracer::compute_pixel(unsigned int x, unsigned int y) const
{
    CGLA::Vec3f result(0.0);

    CGLA::Vec2f vp_pos = CGLA::Vec2f(x, y)*win_to_vp - lower_left;
    CGLA::Vec2f vp_pos_du = CGLA::Vec2f(x+0.5, y)*win_to_vp - lower_left;
    CGLA::Vec2f vp_pos_dv = CGLA::Vec2f(x, y+0.5)*win_to_vp - lower_left;
    CGLA::Vec2f vp_pos_jit;

    Geometry::Ray r;
    RayDiff dr;

    for (unsigned int ky = 0; ky < subdivs; ky++)
    {
        for (unsigned int kx = 0; kx < subdivs; kx++)
        {
            vp_pos_jit = vp_pos + jitter[ky*subdivs + kx];
            r = scene->get_camera()->get_ray(vp_pos_jit);

            dr.dpos = Differential(0.0f, 0.0f);
            dr.ddir = Differential(0.0f, 0.0f);

            vp_pos_jit = vp_pos_du + jitter[ky*subdivs + kx];
            dr.ddir.du = scene->get_camera()->get_ray_dir(vp_pos_jit);
            dr.ddir.du = normalize(dr.ddir.du) - r.direction;

            vp_pos_jit = vp_pos_dv + jitter[ky*subdivs + kx];
            dr.ddir.dv = scene->get_camera()->get_ray_dir(vp_pos_jit);
            dr.ddir.dv = normalize(dr.ddir.dv) - r.direction;

            if (scene->intersect(r))
            {
                dr.transfer(r);

                Shader const * s = get_shader(r);
                if (s)
                    result += s->shade(r, dr);
                if (render_tex.has_texture())
                {
                    CGLA::Vec2f pixel_pos = vp_pos + lower_left;
                    result += render_tex.sample_nearest(
                        pixel_pos[0], 1.0f - pixel_pos[1]
                    );
                }
            }
            else
            {
                result += get_background(r.direction);
            }
        }
    }

    return result/static_cast<double>(subdivs*subdivs);
}

bool ViewDiffTracer::trace_cosine_weighted(Geometry::Ray const & in,
                                           Geometry::Ray & out,
                                           RayDiff & dr) const
{
    static const double M_2SQRT2 = 2.0*sqrt(2.0);

    CGLA::Vec3f d = sample_cosine_weighted(in.hit_normal);
    CGLA::Quatf q;

    q.make_rot(d, in.direction);
    dr.ddir.du = q.apply_unit(dr.ddir.du);
    dr.ddir.dv = q.apply_unit(dr.ddir.dv);
    dr.dpos *= M_2SQRT2;

    out.origin = in.hit_pos;
    out.direction = d;
    out.trace_depth = in.trace_depth + 1;
    out.did_hit_diffuse = in.did_hit_diffuse;

    return trace(out, dr);
}

CGLA::Vec3f ViewDiffTracer::irradiance(PhotonDiffMap const & map,
                                       Geometry::Ray const & r,
                                       RayDiff const & dr,
                                       float max_distance,
                                       int no_of_particles) const
{
    NearestPhotons<PhotonDiff> np;

    float vd_radius2 = std::max(
        CGLA::sqr_length(dr.dpos.du),
        CGLA::sqr_length(dr.dpos.dv)
    );

    np.dist2     = (float *)alloca(sizeof(float) * (no_of_particles+1));
    np.index     = (const PhotonDiff **)
                     alloca(sizeof(PhotonDiff *) * (no_of_particles+1));
    np.pos[0]    = r.hit_pos[0];
    np.pos[1]    = r.hit_pos[1];
    np.pos[2]    = r.hit_pos[2];
    np.max       = no_of_particles;   // k
    np.found     = 0;
    np.got_heap  = 0;
    np.dist2[0]  = vd_radius2 + map.get_longest_dpos();

    map.locate_photons(&np, 1);

    return f(np, map, r, dr);
}

CGLA::Vec3f ViewDiffTracer::caustics_irradiance(Geometry::Ray const & r,
                                                RayDiff const & dr,
                                                float max_distance,
                                                int no_of_particles) const
{
    return irradiance(caustic_diffs, r, dr, max_distance, no_of_particles);
}

CGLA::Vec3f ViewDiffTracer::global_irradiance(Geometry::Ray const & r,
                                              RayDiff const & dr,
                                              float max_distance,
                                              int no_of_particles) const
{
    return irradiance(global_diffs, r, dr, max_distance, no_of_particles);
}
```

`ViewDiffMethod.h`

```cpp
/* ViewDiffMethod.h

    Lasse Jon Fuglsang Pedersen <fuglsang@diku.dk>
    14-10-2011

*/

#ifndef __VIEWDIFFMETHOD_H__
#define __VIEWDIFFMETHOD_H__

#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "PhotonDiff.h"

class ViewDiffMethod
{
public:
    /*
        operator() in the derived class should estimate the contribution of
        the k-nearest photon differentials 'np' from the photon map 'map',
        given a ray 'r' with associated ray differential 'dr'
    */
    virtual CGLA::Vec3f operator()(
        NearestPhotons<PhotonDiff> const & np, PhotonDiffMap const & map,
        Geometry::Ray const & r, RayDiff const & dr) const
    {
        return CGLA::Vec3f(0.0);
    }
};

#endif//__VIEWDIFFMETHOD_H__
```

ViewDiffMethodCC.h

```cpp
/* ViewDiffMethodCC.h

    Lasse Jon Fuglsang Pedersen <fuglsang@diku.dk>
    14-10-2011

*/

#ifndef __VIEWDIFFMETHODCC_H__
#define __VIEWDIFFMETHODCC_H__

#include "ViewDiffMethod.h"

class ViewDiffMethodCC : public ViewDiffMethod
{
public:
    CGLA::Vec3f operator()(
        NearestPhotons<PhotonDiff> const & np, PhotonDiffMap const & map,
        Geometry::Ray const & r, RayDiff const & dr) const;
};

#endif//__VIEWDIFFMETHODCC_H__
```

ViewDiffMethodCC.cpp

```cpp
/* ViewDiffMethodCC.cpp

    Lasse Jon Fuglsang Pedersen <fuglsang@diku.dk>
    14-10-2011

*/

#include "ViewDiffMethodCC.h"
#include "CGLA/Vec2f.h"
#include "CGLA/Mat2x2f.h"
#include "CGLA/Mat3x3f.h"
#include "boost/geometry.hpp"
#include "boost/geometry/geometries/point_xy.hpp"
#include "boost/geometry/geometries/polygon.hpp"
#include "boost/foreach.hpp"

typedef boost::geometry::model::d2::point_xy<float>      Point2D;
typedef boost::geometry::model::ring<Point2D, false>     Polygon2D;

/*
    projects a point 'p' onto the plane of projection defined by the tangential
    surface in the point of intersection for the current view ray 'r', using
    the direction of the view ray as the direction of projection
```

```
*/
CGLA::Vec3f project(Geometry::Ray const &r, CGLA::Vec3f const &p)
{
    CGLA::Vec3f x = r.hit_pos;
    CGLA::Vec3f n = r.hit_normal;
    CGLA::Vec3f omega = -CGLA::normalize(r.direction);

    return (p - omega * (CGLA::dot(n, p - x) / CGLA::dot(n, omega)));
}

/*
    projects the differential position vectors of a footprint 'd' onto the
    plane of projection, also given its position 'p' and projected position
    'p_proj'
*/
Differential project(Geometry::Ray const &r, Differential const &d,
                     CGLA::Vec3f const &p, CGLA::Vec3f const &p_proj)
{
    Differential d_proj;

    d_proj.du = project(r, p + d.du) - p_proj;
    d_proj.dv = project(r, p + d.dv) - p_proj;

    return d_proj;
}

/*
    constructs a matrix that defines a rotation from the plane of projection
    defined by the current view ray 'r' and a footprint 'd', and into a plane
    parallel to the x- and y-axis of the root coordinate frame
*/
CGLA::Mat3x3f get_rotation(Geometry::Ray const &r, Differential const &d)
{
    CGLA::Vec3f du = CGLA::normalize(d.du);
    /*
        note that the params of the Mat3x3f constructor are the rows of the
        matrix, which means that the returned matrix == R^T (see section 4.2)
    */
    return CGLA::Mat3x3f(du, CGLA::cross(r.hit_normal, du), r.hit_normal);
}

/*
    applies a rotation 'R' to a point 'p', using 'x' as the center of rotation
*/
CGLA::Vec3f rotate(CGLA::Mat3x3f const &R, CGLA::Vec3f const &p,
                   CGLA::Vec3f const &x)
{
    return R*(p - x) + x;
}

/*
    applies a rotation 'R' to a footprint 'd'
*/
Differential rotate(CGLA::Mat3x3f const &R, Differential const &d)
{
    return Differential(R*d.du, R*d.dv);
}

/*
    constructs a 2d parallelogram from a projected and rotated footprint
    'd_proj' and its projected position 'p_proj', assuming prior rotation of
    both parameters into a plane parallel to the x- and y-axis of the root
    coordinate system since the third component is ignored
*/
Polygon2D get_parallelogram(Differential const &d_proj,
                            CGLA::Vec3f const &p_proj)
{
    Polygon2D   C;
    CGLA::Vec3f q1;
    CGLA::Vec3f q2;
    CGLA::Vec3f q3;
    CGLA::Vec3f q4;

    /*
        vertices must be added in counter-clockwise order so we have to check
        the determinant of the 2d differential position vectors to decide the
        primary axis
    */
    if (d_proj.du[0] * d_proj.dv[1] -
        d_proj.dv[0] * d_proj.du[1] < 0.0)
    {
        // negative => ccw using dv as primary axis
        q1 = p_proj + d_proj.dv + d_proj.du;
        q2 = p_proj - d_proj.dv + d_proj.du;
        q3 = p_proj - d_proj.dv - d_proj.du;
        q4 = p_proj + d_proj.dv - d_proj.du;
    }
    else
    {
        // positive => ccw using du as primary axis
        q1 = p_proj + d_proj.du + d_proj.dv;
        q2 = p_proj - d_proj.du + d_proj.dv;
        q3 = p_proj - d_proj.du - d_proj.dv;
        q4 = p_proj + d_proj.du - d_proj.dv;
    }

    boost::geometry::append(C, Point2D(q1[0], q1[1]));
    boost::geometry::append(C, Point2D(q2[0], q2[1]));
    boost::geometry::append(C, Point2D(q3[0], q3[1]));
    boost::geometry::append(C, Point2D(q4[0], q4[1]));
    boost::geometry::append(C, Point2D(q1[0], q1[1]));

    return C;
}

/*
    constructs a 2d skew ellipse from a projected footprint 'd_proj' and its
    projected position 'p_proj', using the specified number of vertices
    'num_vertices' (defaults to 20), and assuming prior rotation of both
    parameters into 2d as the third component is ignored
*/
Polygon2D get_skew_ellipse(Differential const & d_proj,
                           CGLA::Vec3f const & p_proj, int num_vertices = 20)
{
    Polygon2D       C;
    CGLA::Mat2x2f   M_skew_ellipse;
    CGLA::Vec2f     q;
    CGLA::Vec2f     q0(p_proj[0], p_proj[1]);
    float           t;

    /*
        vertices must be added in counter-clockwise order so we have to check
        the determinant of the 2d differential position vectors to decide the
        primary axis
    */
    if (d_proj.du[0] * d_proj.dv[1] -
        d_proj.dv[0] * d_proj.du[1] < 0.0)
    {
        // negative => ccw using dv as primary axis
        M_skew_ellipse = CGLA::Mat2x2f(
            CGLA::Vec2f(d_proj.dv[0], d_proj.du[0]),
            CGLA::Vec2f(d_proj.dv[1], d_proj.du[1])
        );
    }
    else
    {
        // positive => ccw using du as primary axis
        M_skew_ellipse = CGLA::Mat2x2f(
            CGLA::Vec2f(d_proj.du[0], d_proj.dv[0]),
            CGLA::Vec2f(d_proj.du[1], d_proj.dv[1])
        );
    }

    // sample points on unit circle, transforming them before appending them
    for (int i = 0; i < num_vertices; i++)
    {
        t = ((float)i / (float)num_vertices) * (2.0 * M_PI);
        q = q0 + M_skew_ellipse * CGLA::Vec2f(cos(t), sin(t));
        boost::geometry::append(C, Point2D(q[0], q[1]));
    }

    t = 0.0;
    q = q0 + M_skew_ellipse * CGLA::Vec2f(cos(t), sin(t));
    boost::geometry::append(C, Point2D(q[0], q[1]));

    return C;
}

/*
    compute the intersection of two convex 2d polygons using boost.geometry

    because the polygons are guaranteed to be convex, it should be noted that
    using boost.geometry may be a suboptimal solution due to the multi-purpose
    nature of this particular library
*/
Polygon2D get_intersection(Polygon2D const & C_a, Polygon2D const & C_b)
{
    std::deque<Polygon2D> C_queue;

    boost::geometry::intersection(C_a, C_b, C_queue);

    if (C_queue.size() > 0)
        return C_queue.front();
    else
        return Polygon2D();
}

/*
    compute the area of a convex 2d polygon using boost.geometry
*/
float area(Polygon2D const & C)
{
    return std::abs(boost::geometry::area(C));
}

CGLA::Vec3f centroid(Polygon2D const & C)
{
    Point2D centroid = boost::geometry::return_centroid<Point2D, Polygon2D>(C);
    return CGLA::Vec3f(centroid.x(), centroid.y(), 0.0);
}

CGLA::Vec3f ViewDiffMethodCC::operator()(NearestPhotons<PhotonDiff> const & np,
                                         PhotonDiffMap const & map,
                                         Geometry::Ray const & r,
                                         RayDiff const & dr) const
{
    CGLA::Vec3f     irrad(0.0f);

    CGLA::Vec3f       x_vd = r.hit_pos;
    CGLA::Vec3f       n_vd = r.hit_normal;

    CGLA::Mat3x3f     R_vd = get_rotation(r, dr.dpos);
    Differential      d_vd = rotate(R_vd, dr.dpos);
    Polygon2D         C_vd = get_parallelogram(d_vd, x_vd);
    float             A_vd = area(C_vd);

    CGLA::Vec3f       x_pd;
    Differential      d_pd;
    Polygon2D         C_pd;
    float             A_pd;
    float             w_pd;
    CGLA::Mat3x3f     M_pd;

    CGLA::Vec3f       x_is;
    Polygon2D         C_is;
    float             A_is;
    float             w_is;
    CGLA::Mat2x2f     M_is_sub;
    CGLA::Mat3x3f     M_is;

    // generate matrix that transforms the centroid of the intersection in 2d
    // into 3d by expressing the 2d centroid with respect to the axis of the
    // projected and rotated view ray differential, and then applying the axis
    // of the original footprint (see section 4.2.3 for more information)
    try
    {
        M_is_sub = CGLA::invert(CGLA::Mat2x2f(
            CGLA::Vec2f(d_vd.du[0], d_vd.dv[0]),
            CGLA::Vec2f(d_vd.du[1], d_vd.dv[1])
        ));
    }
    catch (CGLA::Mat2x2fException e)
    {
        return CGLA::Vec3f(0.0);    // abort
    }

    M_is = CGLA::Mat3x3f(
        CGLA::Vec3f(dr.dpos.du[0], dr.dpos.dv[0], n_vd[0]),
        CGLA::Vec3f(dr.dpos.du[1], dr.dpos.dv[1], n_vd[1]),
        CGLA::Vec3f(dr.dpos.du[2], dr.dpos.dv[2], n_vd[2])
    ) * CGLA::Mat3x3f(
        CGLA::Vec3f(M_is_sub[0][0], M_is_sub[0][1], 0.0),
        CGLA::Vec3f(M_is_sub[1][0], M_is_sub[1][1], 0.0),
        CGLA::Vec3f(0.0, 0.0, 0.0)
    );

    // sum irradiance of k-nearest photons
    for (int i = 1; i <= np.found; i++)
    {
        PhotonDiff const * pd = np.index[i];

        // skip if not incident on tangential surface
        if (dot(map.photon_dir(pd), n_vd) <= 0.0f)
               continue;

        // compute area of the footprint in local coordinates
        // (based on line 70 of PhotonDiff.cpp)
        A_pd = cross(pd->dpos.du, pd->dpos.dv).length()*M_PI;

        // skip if area is very small
        if (A_pd < 1.0e-12f)
            continue;

        // compute world space -> filter space transformation
        try
        {
            CGLA::Vec3f const & du_pd = pd->dpos.du;
            CGLA::Vec3f const & dv_pd = pd->dpos.dv;

            M_pd = CGLA::invert(CGLA::Mat3x3f(
                CGLA::Vec3f(du_pd[0], dv_pd[0], n_vd[0]),
                CGLA::Vec3f(du_pd[1], dv_pd[1], n_vd[1]),
                CGLA::Vec3f(du_pd[2], dv_pd[2], n_vd[2])
            ));
        }
        catch (CGLA::Mat3x3fSingular e)
        {
            continue;
        }

        // project
        x_pd = project(r, pd->pos);                   // projected position
        d_pd = project(r, pd->dpos, pd->pos, x_pd); // projected footprint

        // rotate into 2d
        x_pd = rotate(R_vd, x_pd, x_vd);
        d_pd = rotate(R_vd, d_pd);

        // reconstruct geometry
        C_pd = get_parallelogram(d_pd, x_pd);

        // compute coplanar intersection
        C_is = get_intersection(C_vd, C_pd);
        A_is = area(C_is);

        // skip if area of intersection is zero
        if (A_is <= 0.0)
```

```
                continue;

            // compute coverage of coplanar intersection
            w_is = A_is / A_vd;

            // approximate centroid of actual intersection
            x_is = x_vd + M_is * (centroid(C_is) - x_vd);

            // compute sampling point in filter space
            CGLA::Vec3f x_is_filter = M_pd * (x_is - pd->pos);

            // compute squared distance in filter space
            float x_is_filter_dist2 = CGLA::sqr_length(x_is_filter);

            // skip if outside range of influence
            if (x_is_filter_dist2 > 1.0f || CGLA::isnan(x_is_filter_dist2))
                continue;

            // use the Silverman kernel (same as PhotonDiff.cpp line 92)
            w_pd = CGLA::sqr(1.0f - x_is_filter_dist2);

            // compute irradiance
            irrad += (pd->power / A_pd) * w_pd * w_is;
        }

        // done
        return (3.0 * irrad);
}
```

ViewDiffMethodMS.h

```
/* ViewDiffMethodMS.h

    Lasse Jon Fuglsang Pedersen <fuglsang@diku.dk>
    14-10-2011

*/

#ifndef __VIEWDIFFMETHODMS_H__
#define __VIEWDIFFMETHODMS_H__

#include "ViewDiffMethod.h"

class ViewDiffMethodMS : public ViewDiffMethod
{
private:
    unsigned int    N;
    float           N_step;
    CGLA::Vec3f *   X;

public:
    ViewDiffMethodMS(unsigned int sqrt_num_samples = 3);

    CGLA::Vec3f operator()(
        NearestPhotons<PhotonDiff> const & np, PhotonDiffMap const & map,
        Geometry::Ray const & r, RayDiff const & dr) const;
};

#endif//__VIEWDIFFMETHODMS_H__
```

ViewDiffMethodCC.cpp

```
/* ViewDiffMethodMS.cpp

    Lasse Jon Fuglsang Pedersen <fuglsang@diku.dk>
    14-10-2011

*/

#include "ViewDiffMethodMS.h"
#include "CGLA/Mat3x3f.h"

ViewDiffMethodMS::ViewDiffMethodMS(unsigned int sqrt_num_samples)
{
    N       = std::max(static_cast<unsigned int>(2), sqrt_num_samples);
    N_step  = 2.0 / static_cast<float>(N - 1);
    X       = new CGLA::Vec3f[N * N];
}

CGLA::Vec3f ViewDiffMethodMS::operator()(NearestPhotons<PhotonDiff> const & np,
                                          PhotonDiffMap const & map,
                                          Geometry::Ray const & r,
                                                RayDiff const & dr) const
{
    CGLA::Vec3f       irrad(0.0f);

    CGLA::Vec3f       x_vd = r.hit_pos;
    CGLA::Vec3f       n_vd = r.hit_normal;
    CGLA::Mat3x3f     M_vd;

    float             A_pd;
    CGLA::Mat3x3f     M_pd;
    float             w_pd = 0.0;

    // generate sampling points
    M_vd = CGLA::Mat3x3f(
        CGLA::Vec3f(dr.dpos.du[0], dr.dpos.dv[0], n_vd[0]),
        CGLA::Vec3f(dr.dpos.du[1], dr.dpos.dv[1], n_vd[1]),
        CGLA::Vec3f(dr.dpos.du[2], dr.dpos.dv[2], n_vd[2])
    );

    for (unsigned int i = 0; i < N; i++)
    {
        for (unsigned int j = 0; j < N; j++)
        {
            float s = static_cast<float>(i) * N_step - 1.0;
            float t = static_cast<float>(j) * N_step - 1.0;

            X[i*N + j] = x_vd + M_vd * CGLA::Vec3f(s, t, 0.0);
        }
    }

    // sum irradiance of k-nearest photons
    for (int i = 1; i <= np.found; i++)
    {
        PhotonDiff const * pd = np.index[i];

        // skip if not incident on tangential surface
        if (dot(map.photon_dir(pd), n_vd) <= 0.0f)
              continue;

        // compute area of the footprint in local coordinates
        // (based on line 70 of PhotonDiff.cpp)
        A_pd = cross(pd->dpos.du, pd->dpos.dv).length()*M_PI;

        // skip if area is very small
        if (A_pd < 1.0e-12f)
            continue;

        // compute world space -> filter space transformation
        try
        {
            CGLA::Vec3f const & du_pd = pd->dpos.du;
            CGLA::Vec3f const & dv_pd = pd->dpos.dv;

            M_pd = CGLA::invert(CGLA::Mat3x3f(
                CGLA::Vec3f(du_pd[0], dv_pd[0], n_vd[0]),
                CGLA::Vec3f(du_pd[1], dv_pd[1], n_vd[1]),
                CGLA::Vec3f(du_pd[2], dv_pd[2], n_vd[2])
            ));
        }
        catch (CGLA::Mat3x3fSingular e)
        {
            continue;
        }

        // loop over sampling points
        for (unsigned int j = 0; j < N * N; j++)
        {
            // compute sampling point in filter space
            CGLA::Vec3f x_st_filter = M_pd * (X[j] - pd->pos);

            // compute squared distance in filter space
            float x_st_filter_dist2 = CGLA::sqr_length(x_st_filter);

            // skip if outside range of influence
            if (x_st_filter_dist2 > 1.0f || CGLA::isnan(x_st_filter_dist2))
                continue;

            // use the Silverman kernel (same as PhotonDiff.cpp line 92)
            w_pd += CGLA::sqr(1.0f - x_st_filter_dist2);
        }

        // average weights
        w_pd /= static_cast<float>(N * N);

        // compute irradiance
        irrad += (pd->power / A_pd) * w_pd;
    }

    // done
    return (3.0 * irrad);
}
```

Shader.h

```cpp
#ifndef SHADER_H
#define SHADER_H

#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "RayDiff.h"

class Shader
{
public:
    virtual CGLA::Vec3f shade(Geometry::Ray& r, bool emit = true) const = 0;

    /* fuglsang begin */
    virtual CGLA::Vec3f shade(Geometry::Ray& r, RayDiff& dr, bool emit = true) ↩
        const
    {
        /*std::cout
            << "shade() cannot propagate ray differential, aborting"
            << std::endl;*/

        return CGLA::Vec3f(0.0, 0.0, 0.0);
    }
    /* fuglsang end */
};

#endif // SHADER_H
```

PhotonCaustics.h

```cpp
#ifndef PHOTONCAUSTICS_H
#define PHOTONCAUSTICS_H

#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "ParticleTracer.h"
#include "MCLambertian.h"
#include "RayDiff.h"

class PhotonCaustics : public MCLambertian
{
public:
  PhotonCaustics(ParticleTracer* particle_tracer,
                 const std::vector<Light*>& light_vector,
                 float max_distance_in_estimate,
                 int no_of_photons_in_estimate)
    : MCLambertian(particle_tracer, light_vector),
      tracer(particle_tracer),
      max_dist(max_distance_in_estimate),
      photons(no_of_photons_in_estimate)
  { }

  virtual CGLA::Vec3f shade(Geometry::Ray& r, bool emit = true) const;
  /* fuglsang begin */
  virtual CGLA::Vec3f shade(Geometry::Ray &r, RayDiff &dr, bool emit = true) ↩
      const;
  /* fuglsang end */

protected:
  ParticleTracer* tracer;
  float max_dist;
  int photons;
};

#endif // PHOTONCAUSTICS_H
```

PhotonCaustics.cpp

```cpp
#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "PhotonCaustics.h"

/* fuglsang begin */
#include "ViewDiffTracer.h"
/* fuglsang end */

using namespace CGLA;
using namespace Geometry;

Vec3f PhotonCaustics::shade(Ray& r, bool emit) const
{
  Vec3f result = tracer->caustics_irradiance(r, max_dist, photons);
  result *= get_diffuse(r)/M_PI;
  return result + Emission::shade(r, emit); // + MCLambertian::shade(r, emit);
}

/* fuglsang begin */
CGLA::Vec3f PhotonCaustics::shade(Geometry::Ray & r, RayDiff & dr,
                                  bool emit) const
{
    ViewDiffTracer * vdtracer = static_cast<ViewDiffTracer *>(tracer);
    if (!vdtracer)
        return Shader::shade(r, dr, emit);

    CGLA::Vec3f result = vdtracer->caustics_irradiance(
        r, dr, max_dist, photons);

    result *= get_diffuse(r)/M_PI;
    return result + Emission::shade(r, emit);
}
/* fuglsang end */
```

PhotonLambertian.h

```
#ifndef PHOTONLAMBERTIAN_H
#define PHOTONLAMBERTIAN_H

#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "ParticleTracer.h"
#include "PhotonCaustics.h"
#include "RayDiff.h"

class PhotonLambertian : public PhotonCaustics
{
public:
  PhotonLambertian(ParticleTracer* particle_tracer,
                   const std::vector<Light*>& light_vector,
                   float max_distance_in_estimate,
                   int no_of_photons_in_estimate,
                   bool use_final_gathering = true,
                   unsigned int no_of_samples = 1,
                   const PhotonCaustics* caustics_shader = 0)
    : PhotonCaustics(particle_tracer, light_vector, max_distance_in_estimate, ↩
        no_of_photons_in_estimate),
      gather(use_final_gathering),
      samples(no_of_samples),
      caustics(caustics_shader)
  { }

  virtual CGLA::Vec3f shade(Geometry::Ray& r, bool emit = true) const;
  /* fuglsang begin */
  virtual CGLA::Vec3f shade(Geometry::Ray & r, RayDiff & dr, bool emit = true) ↩
      const;
  /* fuglsang end */

  bool is_gathering() const { return gather; }
  void toggle_final_gathering() { gather = !gather; }

protected:
  CGLA::Vec3f split_shade(Geometry::Ray& r, bool emit) const;
  CGLA::Vec3f shade_new_ray(Geometry::Ray& r) const;
  /* fuglsang begin */
  CGLA::Vec3f split_shade(Geometry::Ray & r, RayDiff & dr, bool emit) const;
  CGLA::Vec3f shade_new_ray(Geometry::Ray & r, RayDiff & dr) const;
  /* fuglsang end */

  bool gather;
  unsigned int samples;
  const PhotonCaustics* caustics;
};

#endif // PHOTONLAMBERTIAN_H
```

PhotonLambertian.cpp

```
#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "PhotonLambertian.h"

/* fuglsang begin */
#include "ViewDiffTracer.h"
/* fuglsang end */

```

```
using namespace CGLA;
using namespace Geometry;

Vec3f PhotonLambertian::shade(Ray& r, bool emit) const
{
  if(gather && !r.did_hit_diffuse)
    return split_shade(r, emit);

  Vec3f result = tracer->global_irradiance(r, max_dist, photons);
  result *= get_diffuse(r)/static_cast<float>(M_PI);
  return result + Emission::shade(r, !gather);
}

Vec3f PhotonLambertian::split_shade(Ray& r, bool emit) const
{
  Vec3f result = Lambertian::shade(r, emit);
  Vec3f rho = get_diffuse(r);
  if(rho[0] + rho[1] + rho[2] > 0.0)
  {
    Vec3f indirect(0.0f);
    for(unsigned int i = 0; i < samples; ++i)
    {
      Ray new_ray;
      tracer->trace_cosine_weighted(r, new_ray);
      indirect += shade_new_ray(new_ray);
    }
    result += indirect*rho/static_cast<double>(samples);
    result += caustics
      ? caustics->shade(r, false)
      : (rho/static_cast<float>(M_PI))*tracer->caustics_irradiance(r, max_dist, ↩
           photons);
  }
  return result;
}

Vec3f PhotonLambertian::shade_new_ray(Ray& r) const
{
  if(r.has_hit)
  {
    r.did_hit_diffuse = true;
    const Shader* s = tracer->get_shader(r);
    if(s)
      return s->shade(r, false);
  }
  else
    return tracer->get_background(r.direction);

  return Vec3f(0.0f);
}

/* fuglsang begin */
CGLA::Vec3f PhotonLambertian::shade(Geometry::Ray &r, RayDiff &dr,
                                    bool emit) const
{
    if (gather && !r.did_hit_diffuse)
        return split_shade(r, emit);

    ViewDiffTracer * vdtracer = static_cast<ViewDiffTracer *>(tracer);
    if (!vdtracer)
        return Shader::shade(r, dr, emit);

    CGLA::Vec3f result = vdtracer->global_irradiance(r, dr, max_dist, photons);

    result *= get_diffuse(r)/static_cast<float>(M_PI);
    return result + Emission::shade(r, !gather);
}

CGLA::Vec3f PhotonLambertian::split_shade(Geometry::Ray &r, RayDiff &dr,
                                          bool emit) const
{
    ViewDiffTracer * vdtracer = static_cast<ViewDiffTracer *>(tracer);
    if (!vdtracer)
        return Shader::shade(r, dr, emit);

    CGLA::Vec3f result = Lambertian::shade(r, emit);
    CGLA::Vec3f rho = get_diffuse(r);

    if (rho[0] + rho[1] + rho[2] > 0.0)
    {
        CGLA::Vec3f indirect(0.0f);

        for (unsigned int i = 0; i < samples; ++i)
        {
            Geometry::Ray r_split;
            RayDiff dr_split = dr;
            vdtracer->trace_cosine_weighted(r, r_split, dr_split);
            indirect += shade_new_ray(r_split, dr_split);
        }

        result += indirect*rho/static_cast<double>(samples);
        result += caustics
                ? caustics->shade(r, dr, false)
                : (rho/static_cast<float>(M_PI))
                * vdtracer->caustics_irradiance(r, dr, max_dist, photons);
    }

    return result;
}

```

```
Vec3f PhotonLambertian::shade_new_ray(Geometry::Ray & r, RayDiff & dr) const
{
    if (r.has_hit)
    {
        r.did_hit_diffuse = true;
        Shader const * s = tracer->get_shader(r);
        if (s)
            return s->shade(r, dr, false);
    }
    else
        return tracer->get_background(r.direction);

    return Vec3f(0.0f);
}
/* fuglsang end */
```

Transparent.h

```
#ifndef TRANSPARENT_H
#define TRANSPARENT_H

#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "Geometry/Material.h"
#include "PathTracer.h"
#include "Mirror.h"

class Transparent : public Mirror
{
public:
  Transparent(PathTracer* pathtracer, int no_of_splits = 1, int max_trace_depth = ↩
      20)
    : Mirror(pathtracer, max_trace_depth),
      splits(no_of_splits)
  { }

  virtual CGLA::Vec3f shade(Geometry::Ray& r, bool emit = true) const;
  /* fuglsang begin */
  virtual CGLA::Vec3f shade(Geometry::Ray & r, RayDiff & dr, bool emit = true) ↩
      const;
  /* fuglsang end */

protected:
  CGLA::Vec3f split_shade(Geometry::Ray& r, bool emit) const;
  /* fuglsang begin */
  CGLA::Vec3f split_shade(Geometry::Ray & r, RayDiff & dr, bool emit) const;
  /* fuglsang end */

  int splits;
};

#endif // TRANSPARENT_H
```

Transparent.cpp

```
#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "mt_random.h"
#include "Transparent.h"

/* fuglsang begin */
#include "ViewDiffTracer.h"
/* fuglsang end */

using namespace CGLA;
using namespace Geometry;

Vec3f Transparent::shade(Ray& r, bool emit) const
{
  if(r.trace_depth >= max_depth)
    return Vec3f(0.0f);
  if(r.trace_depth < splits)
    return split_shade(r, emit);

  double R;
  Ray refracted;
  tracer->trace_refracted(r, refracted, R);

  double xi = mt_random();
  if(xi < R)
  {
    Ray reflected;
    tracer->trace_reflected(r, reflected);
    return shade_new_ray(reflected);
  }
  return shade_new_ray(refracted);
}

Vec3f Transparent::split_shade(Ray& r, bool emit) const
{
  double R;
  Ray reflected, refracted;
  tracer->trace_reflected(r, reflected);
  tracer->trace_refracted(r, refracted, R);
  return R*shade_new_ray(reflected) + (1.0 - R)*shade_new_ray(refracted);
}

/* fuglsang begin */
CGLA::Vec3f Transparent::shade(Geometry::Ray & r, RayDiff & dr,
                               bool emit) const
{
    if (r.trace_depth >= max_depth)
        return Vec3f(0.0f);
    if (r.trace_depth < splits)
        return split_shade(r, dr, emit);

    ViewDiffTracer * vdtracer = static_cast<ViewDiffTracer *>(tracer);
    if (!vdtracer)
        return Shader::shade(r, dr, emit);

    double R;
    Geometry::Ray r_refracted;
    RayDiff dr_refracted = dr;
    vdtracer->trace_refracted(r, r_refracted, dr_refracted, R);

    double xi = mt_random();
    if (xi < R)
    {
        Geometry::Ray r_reflected;
        RayDiff dr_reflected = dr;
        vdtracer->trace_reflected(r, r_reflected, dr_reflected);
        return shade_new_ray(r_reflected, dr_reflected);
    }

    return shade_new_ray(r_refracted, dr_refracted);
}

CGLA::Vec3f Transparent::split_shade(Geometry::Ray & r, RayDiff & dr,
                                     bool emit) const
{
    ViewDiffTracer * vdtracer = static_cast<ViewDiffTracer *>(tracer);
    if (!vdtracer)
        return Shader::shade(r, dr, emit);

    double R;
    Geometry::Ray r_reflected, r_refracted;
    RayDiff dr_reflected, dr_refracted;

    dr_reflected = dr;
    dr_refracted = dr;

    vdtracer->trace_reflected(r, r_reflected, dr_reflected);
    vdtracer->trace_refracted(r, r_refracted, dr_refracted, R);

    return (shade_new_ray(r_reflected, dr_reflected)*R +
            shade_new_ray(r_refracted, dr_refracted)*(1.0 - R));
}
/* fuglsang end */
```

Mirror.h

```
#ifndef MIRROR_H
#define MIRROR_H

#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "PathTracer.h"
#include "Collimated.h"
#include "Shader.h"
#include "RayDiff.h"

class Mirror : virtual public Shader
{
public:
  Mirror(PathTracer* pathtracer, int max_trace_depth = 20)
    : tracer(pathtracer), max_depth(max_trace_depth), laser(0)
  { }

  virtual CGLA::Vec3f shade(Geometry::Ray& r, bool emit = true) const;
  /* fuglsang begin */
  virtual CGLA::Vec3f shade(Geometry::Ray & r, RayDiff & dr, bool emit = true) ↩
      const;
  /* fuglsang end */

  void set_laser(Collimated* collimated) { laser = collimated; }

protected:
  CGLA::Vec3f shade_new_ray(Geometry::Ray& r) const;
  /* fuglsang begin */
  CGLA::Vec3f shade_new_ray(Geometry::Ray & r, RayDiff & dr) const;
  /* fuglsang end */

  Collimated* laser;
  PathTracer* tracer;
  int max_depth;
};

#endif // MIRROR_H
```

Mirror.cpp

```
#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "Mirror.h"

/* fuglsang begin */
#include "ViewDiffTracer.h"
/* fuglsang end */

using namespace CGLA;
using namespace Geometry;

Vec3f Mirror::shade(Ray& r, bool emit) const
{
  if(r.trace_depth >= max_depth)
    return Vec3f(0.0f);

  Ray reflected;
  tracer->trace_reflected(r, reflected);
  return shade_new_ray(reflected);
}

Vec3f Mirror::shade_new_ray(Ray& r) const
{
  if(r.has_hit)
  {
    const Shader* s = tracer->get_shader(r);
    if(s)
      return s->shade(r, true);
  }
  else
    return tracer->get_background(r.direction);

  return Vec3f(0.0f);
}

/* fuglsang begin */
CGLA::Vec3f Mirror::shade(Geometry::Ray & r, RayDiff & dr, bool emit) const
{
    if (r.trace_depth >= max_depth)
        return Vec3f(0.0f);

    ViewDiffTracer * vdtracer = static_cast<ViewDiffTracer *>(tracer);
    if (!vdtracer)
        return Shader::shade(r, dr, emit);

    Geometry::Ray r_reflected;
    vdtracer->trace_reflected(r, r_reflected, dr);
    return shade_new_ray(r_reflected, dr);
}

Vec3f Mirror::shade_new_ray(Geometry::Ray & r, RayDiff & dr) const
{
    if (r.has_hit)
    {
        Shader const * s = tracer->get_shader(r);
        if (s)
            return s->shade(r, dr, true);
    }
    else
        return tracer->get_background(r.direction);

    return Vec3f(0.0f);
}
/* fuglsang end */
```

Metal.h

```cpp
#ifndef METAL_H
#define METAL_H

#include <climits>
#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "PathTracer.h"
#include "Mirror.h"

class Metal : public Mirror
{
public:
  Metal(PathTracer* pathtracer, int no_of_splits = 1)
    : Mirror(pathtracer, INT_MAX), splits(no_of_splits)
  { }

  virtual CGLA::Vec3f shade(Geometry::Ray& r, bool emit = true) const;
  /* fuglsang begin */
  virtual CGLA::Vec3f shade(Geometry::Ray &r, RayDiff &dr, bool emit = true) ↩
      const;
  /* fuglsang end */

protected:
  int splits;
};

#endif // METAL_H
```

Metal.cpp

```cpp
#include "CGLA/Vec3f.h"
#include "Geometry/Ray.h"
#include "mt_random.h"
#include "Metal.h"

/* fuglsang begin */
#include "ViewDiffTracer.h"
/* fuglsang end */

using namespace CGLA;
using namespace Geometry;

Vec3f Metal::shade(Ray& r, bool emit) const
{
  Vec3f result(0.0f);
  Vec3f R;
  Ray reflected;
  tracer->trace_reflected(r, reflected, R);

  if(r.trace_depth >= splits)
  {
    double prob = (R[0] + R[1] + R[2])/3.0;
    double xi = mt_random();
    if(xi < prob)
      result += R*shade_new_ray(reflected)/prob;
  }
  else
    result += R*shade_new_ray(reflected);
  return result;
}

/* fuglsang begin */
CGLA::Vec3f Metal::shade(Geometry::Ray &r, RayDiff &dr, bool emit) const
{
    CGLA::Vec3f result(0.0f);
    CGLA::Vec3f R;
    Geometry::Ray r_reflected;
    RayDiff dr_reflected = dr;

    ViewDiffTracer * vdtracer = static_cast<ViewDiffTracer *>(tracer);
    if (!vdtracer)
        return Shader::shade(r, dr, emit);

    vdtracer->trace_reflected(r, r_reflected, dr_reflected, R);

    if (r.trace_depth >= splits)
    {
        double prob = (R[0] + R[1] + R[2])/3.0;
        double xi = mt_random();
        if (xi < prob)
            result += R*shade_new_ray(r_reflected, dr_reflected) / prob;
    }
    else
        result += R*shade_new_ray(r_reflected, dr_reflected);

    return result;
}
/* fuglsang end */
```